Reference: PLT Scheme

Version 3.99.0.23

April 30, 2008

This manual defines the core PLT Scheme language and describes its most prominent libraries. The companion manual §"**Guide**: PLT Scheme" provides a friendlier (though less precise and less complete) overview of the language.

```
#lang scheme/base
#lang scheme
```

Unless otherwise noted, the bindings defined in this manual are exported by the `scheme/base` and `scheme` languages, where `scheme` includes all of `scheme/base`.

Contents

1 Language Model

1.1 Evaluation Model

Scheme evaluation can be viewed as the simplification of expressions to obtain values. For example, just as an elementary-school student simplifies

```
1 + 1 = 2
```

Scheme evaluation simplifies

```
(+ 1 1) → 2
```

The arrow → above replaces the more traditional = to emphasize that evaluation proceeds in a particular direction towards simpler expressions. In particular, a *value* is an expression that evaluation simplifies no further, such as the number `2`.

1.1.1 Sub-expression Evaluation and Continuations

Some simplifications require more than one step. For example:

```
(- 4 (+ 1 1)) → (- 4 2) → 2
```

An expression that is not a value can always be partitioned into two parts: a *redex*, which is the part that changed in a single-step simplification (show in blue), and the *continuation*, which is the surrounding expression context. In `(- 4 (+ 1 1))`, the redex is `(+ 1 1)`, and the continuation is `(- 4 [])`, where [] takes the place of the redex. That is, the continuation says how to "continue" after the redex is reduced to a value.

Before some things can be evaluated, some sub-expressions must be evaluated; for example, in the application `(- 4 (+ 1 1))`, the application of `-` cannot be reduced until the sub-expression `(+ 1 1)` is reduced.

Thus, the specification of each syntactic form specifies how (some of) its sub-expressions are evaluated, and then how the results are combined to reduce the form away.

The *dynamic extent* of an expression is the sequence of evaluation steps during which an expression contains the redex.

1.1.2 Tail Position

An expression *`expr1`* is in *tail position* with respect to an enclosing expression *`expr2`* if, whenever *`expr1`* becomes a redex, its continuation is the same as was the enclosing *`expr2`*'s continuation.

For example, the `(+ 1 1)` expression is *not* in tail position with respect to `(- 4 (+ 1 1))`. To illustrate, we use the notation `C[expr]` to mean the expression that is produced by substituting `expr` in place of [] in the continuation `C`:

```
C[(- 4 (+ 1 1))] → C[(- 4 2)]
```

In this case, the continuation for reducing `(+ 1 1)` is `C[(+ 4 [])]`, not just `C`.

In contrast, `(+ 1 1)` is in tail position with respect to `(if (zero? 0) (+ 1 1) 3)`, because, for any continuation `C`,

```
C[(if (zero? 0) (+ 1 1) 3)] → C[(if #t (+ 1 1) 3)] → C[(+ 1 1)]
```

The steps in this reduction sequence are driven by the definition of `if`, and they do not depend on the continuation `C`. The "then" branch of an `if` form is always in tail position with respect to the `if` form. Due to a similar reduction rule for `if` and `#f`, the "else" branch of an `if` form is also in tail position.

Tail-position specifications provide a guarantee about the asymptotic space consumption of a computation. In general, the specification of tail positions goes with each syntactic form, like `if`.

1.1.3 Multiple Return Values

A Scheme expression can evaluate to *multiple values*, in the same way that a procedure can accept multiple arguments.

Most continuations expect a particular number of result values. Indeed, most continuations, such as `(+ [] 1)` expect a single value. The continuation `(let-values ([(x y) []]) expr)` expects two result values; the first result replaces `x` in the body `expr`, and the second replaces `y` in `expr`. The continuation `(begin [] (+ 1 2))` accepts any number of result values, because it ignores the result(s).

In general, the specification of a syntactic form inidicates the number of values that it produces and the number that it expects from each of its sub-expression. In addtion, some procedures (notably `values`) produce multiple values, and some procedures (notably `call-with-values`) create continuations internally that accept a certain number of values.

1.1.4 Top-Level Variables

Given

```
x = 10
```

then an algebra student simplifies `x + 1` as follows:

```
x + 1 = 10 + 1 = 11
```

Scheme works much the same way, in that a set of top-level variables are available for substitutions on demand during evaluation. For example, given

```
(define x 10)
```

then

`(+ x 1)` → `(+ 10 1)` → `11`

In Scheme, the way definitions appear is just as important as the way that they are used. Scheme evaluation thus keeps track of both definitions and the current expression, and it extends the set of definitions in response to evaluating forms such as `define`.

Each evaluation step, then, takes the current set of definitions and program to a new set of definitions and program. Before a `define` can be moved into the set of definitions, its right-hand expression must be reduced to a value.

```
 defined:
 evaluate:(begin (define x (+ 9 1)) (+ x 1))
→defined:
 evaluate:(begin (define x 10) (+ x 1))
→defined: (define x 10)
 evaluate:(begin (void) (+ x 1))
→defined: (define x 10)
 evaluate:(+ x 1)
→defined: (define x 10)
 evaluate:(+ 10 1)
→defined: (define x 10)
 evaluate:11
```

Using `set!`, a program can change the value associated with an existing top-level variable:

```
 defined: (define x 10)
 evaluate:(begin (set! x 8) x)
→defined: (define x 8)
 evaluate:(begin (void) x)
→defined: (define x 8)
 evaluate:x
→defined: (define x 8)
 evaluate:8
```

1.1.5 Objects and Imperative Update

In addition to `set!` for imperative update of top-level variables, various procedures enable the modification of elements within a compound data structure. For example, `vector-set!` modifies the content of a vector.

To allow such modifications to data, we must distingiush between values, which are the results of expressions, and *objects*, which hold the data referenced by a value.

A few kinds of objects can serve directly as values, including booleans, `(void)`, and small exact integers. More generally, however, a value is a reference to an object. For example, a value can be a reference to a particular vector that currently holds the value `10` in its first slot. If an object is modified, then the modification is visible through all copies of the value that reference the same object.

In the evaluation model, a set of objects must be carried along with each step in evaluation, just like the definition set. Operations that create objects, such as `vector`, add to the set of objects:

```
  objects:
  defined:
  evaluate:(begin (define x (vector 10 20))
                  (define y x)
                  (vector-set! x 0 11)
                  (vector-ref y 0))
→objects: (define <o1> (vector 10 20))
  defined:
  evaluate:(begin (define x <o1>)
                  (define y x)
                  (vector-set! x 0 11)
                  (vector-ref y 0))
→objects: (define <o1> (vector 10 20))
  defined: (define x <o1>)
  evaluate:(begin (void)
                  (define y x)
                  (vector-set! x 0 11)
                  (vector-ref y 0))
→objects: (define <o1> (vector 10 20))
  defined: (define x <o1>)
  evaluate:(begin (define y x)
                  (vector-set! x 0 11)
                  (vector-ref y 0))
→objects: (define <o1> (vector 10 20))
  defined: (define x <o1>)
```

```
   evaluate: (begin (define y <o1>)
                    (vector-set! x 0 11)
                    (vector-ref y 0))
→objects:  (define <o1> (vector 10 20))
   defined: (define x <o1>)
            (define y <o1>)
   evaluate: (begin (void)
                    (vector-set! x 0 11)
                    (vector-ref y 0))
→objects:  (define <o1> (vector 10 20))
   defined: (define x <o1>)
            (define y <o1>)
   evaluate: (begin (vector-set! x 0 11)
                    (vector-ref y 0))
→objects:  (define <o1> (vector 10 20))
   defined: (define x <o1>)
            (define y <o1>)
   evaluate: (begin (vector-set! <o1> 0 11)
                    (vector-ref y 0))
→objects:  (define <o1> (vector 11 20))
   defined: (define x <o1>)
            (define y <o1>)
   evaluate: (begin (void)
                    (vector-ref y 0))
→objects:  (define <o1> (vector 11 20))
   defined: (define x <o1>)
            (define y <o1>)
   evaluate: (vector-ref y 0)
→objects:  (define <o1> (vector 11 20))
   defined: (define x <o1>)
            (define y <o1>)
   evaluate: (vector-ref <o1> 0)
→objects:  (define <o1> (vector 11 20))
   defined: (define x <o1>)
            (define y <o1>)
   evaluate: 11
```

The distinction between a top-level variable and an object reference is crucial. A top-level variable is not a value; each time a variable expression is evaluated, the value is extracted from the current set of definitions. An object reference, in contrast is a value, and therefore needs no further evaluation. The model evaluation steps above use angle-bracketed `<o1>` for an object reference to distinguish it from a variable name.

A direct object reference can never appear in a text-based source program. A program representation created with datum->syntax-object, however, can embed direct references to existing objects.

1.1.6 Object Identity and Comparisons

The eq? operator compares two values, returning #t when the values refer to the same object. This form of equality is suitable for comparing objects that support imperative update (e.g., to determine that the effect of modifying an object through one reference is visible through another reference). Also, an eq? test evaluates quickly, and eq?-based hashing is more lightweight than equal?-based hashing in hash tables.

In some cases, however, eq? is unsuitable as a comparison operator, because the generation of objects is not clearly defined. In particular, two applications of + to the same two exact integers may or may not produce results that are eq?, although the results are always equal?. Similarly, evaluation of a lambda form typically generates a new procedure object, but it may re-use a procedure object previously generated by the same source lambda form.

The behavior of a datatype with respect to eq? is generally specified with the datatype and its associated procedures.

1.1.7 Garbage Collection

In the program state

```
objects: (define <o1> (vector 10 20))
         (define <o2> (vector 0))
defined: (define x <o1>)
evaluate:(+ 1 x)
```

evaluation cannot depend on <o2>, because it is not part of the program to evaluate, and it is not referenced by any definition that is accessible in the program. The object <o2> may therefore be removed from the evaluation by *garbage collection*.

A few special compound datatypes hold *weak references* to objects. Such weak references are treated specially by the garbage collector in determining which objects are reachable for the remainder of the computation. If an object is reachable only via a weak reference, then the object can be reclaimed, and the weak reference is replaced by a different value (typically #f).

1.1.8 Procedure Applications and Local Variables

Given

```
f(x) = x + 10
```

then an algebra student simplifies `f(1)` as follows:

```
f(7) = 7 + 10 = 17
```

The key step in this simplification is take the body of the defined function `f`, and then replace each `x` with the actual value `1`.

Scheme procedure application works much the same way. A procedure is an object, so evaluating `(f 7)` starts with a variable lookup:

```
  objects: (define <p1> (lambda (x) (+ x 10)))
  defined: (define f <p1>)
  evaluate:(f 7)
→objects: (define <p1> (lambda (x) (+ x 10)))
  defined: (define f <p1>)
  evaluate:(<p1> 7)
```

Unlike in algebra, however, the value associated with an argument can be changed in the body of a procedure by using `set!`, as in the example `(lambda (x) (begin (set! x 3) x))`. Since the value associated with `x` can be changed, an actual value for cannot be substituted for `x` when the procedure is applied.

Instead, a new *location* is created for each variable on each application. The argument value is placed in the location, and each instance of the variable in the procedure body is replaced with the new location:

```
  objects: (define <p1> (lambda (x) (+ x 10)))
  defined: (define f <p1>)
  evaluate:(<p1> 7)
→objects: (define <p1> (lambda (x) (+ x 10)))
  defined: (define f <p1>)
           (define xloc 7)
  evaluate:(+ xloc 10)
→objects: (define <p1> (lambda (x) (+ x 10)))
  defined: (define f <p1>)
           (define xloc 7)
  evaluate:(+ 7 10)
→objects: (define <p1> (lambda (x) (+ x 10)))
  defined: (define f <p1>)
           (define xloc 7)
  evaluate:17
```

A location is the same as a top-level variable, but when a location is generated, it (conceptually) uses a name that has not been used before and that cannot not be generated again or accessed directly.

Generating a location in this way means that `set!` evaluates for local variables in the same way as for top-level variables, because the local variable is always replaced with a location by the time the `set!` form is evaluated:

```
  objects: (define <p1> (lambda (x) (begin (set! x 3) x)))
  defined: (define f <p1>)
  evaluate:(f 7)
→objects: (define <p1> (lambda (x) (begin (set! x 3) x)))
  defined: (define f <p1>)
  evaluate:(<p1> 7)
→objects: (define <p1> (lambda (x) (begin (set! x 3) x)))
  defined: (define f <p1>)
           (define xloc 7)
  evaluate:(begin (set! xloc 3) xloc)
→objects: (define <p1> (lambda (x) (begin (set! x 3) x)))
  defined: (define f <p1>)
           (define xloc 3)
  evaluate:(begin (void) xloc)
→objects: (define <p1> (lambda (x) (begin (set! x 3) x)))
  defined: (define f <p1>)
           (define xloc 3)
  evaluate:xloc
→objects: (define <p1> (lambda (x) (begin (set! x 3) x)))
  defined: (define f <p1>)
           (define xloc 3)
  evaluate:3
```

The substition and location-generation step of procedure application requires that the argument is a value. Therefore, in `((lambda (x) (+ x 10)) (+ 1 2))`, the `(+ 1 2)` subexpression must be simplified to the value `3`, and then `3` can be placed into a location for `x`. In other words, Scheme is a *call-by-value* language.

Evaluation of a local-variable form, such as `(let ([x (+ 1 2)]) expr)`, is the same as for a procedure call. After `(+ 1 2)` produces a value, it is stored in a fresh location that replaces every instance of `x` in `expr`.

1.1.9 Variables and Locations

A *variable* is a placeholder for a value, and an expressions in an initial program refer to variables. A *top-level variable* is both a variable and a location. Any other variable is always replaced by a location at run-time, so that evaluation of expressions involves only locations. A single *local variable* (i.e., a non-top-level, non-module-level variable), such as a procedure argument, can correspond to different locations through different instantiations.

For example, in the program

```
(define y (+ (let ([x 5]) x) 6))
```

both `y` and `x` are variables. The `y` variable is a top-level variable, and the `x` is a local variable. When this code is evaluated, a location is created for `x` to hold the value `5`, and a location is also created for `y` to hold the value `6`.

The replacement of a variable with a location during evaluation implements Scheme's *lexical scoping*. For example, when a procedure-argument variable `x` is replaced by the location `xloc`, then it is replaced throughout the body of the procedure, including with any nested `lambda` forms. As a result, future references of the variable always access the same location.

1.1.10 Modules and Module-Level Variables

Most definitions in PLT Scheme are in modules. In terms of evaluation, a module is essentially a prefix on a defined name, so that different modules can define the name. That is, a *module-level variable* is like a top-level variable from the perspective of evaluation.

One difference between a module an a top-level definition is that a module can be declared without instantiating its module-level definitions. Evaluation of a `require` *instantiates* (i.e., triggers the *instantiation* of) a declared module, which creates variables that correspond to its module-level definitions.

For example, given the module declaration

```
(module m mzscheme
  (define x 10))
```

the evaluation of `(require m)` creates the variable `x` and installs `10` as its value. This `x` is unrelated to any top-level definition of `x`.

[

Module Phases]Module Phases

A module can be instantiated in multiple *phases*. A phase is an integer that, again, is effectively a prefix on the names of module-level definitions. A top-level `require` instantiates a module at phase 0, if the module is not already instantiated at phase 0. A top-level `require-for-syntax` instantiates a module at phase 1 (if it is not already instantiated at that level); a `require-for-syntax` also has a different binding effect on further program parsing, as described in §1.2.3.4 "Introducing Bindings".

Within a module, some definitions are shifted by a phase already; the `define-for-syntax` form is like `define`, but it defines a variable at relative phase 1, instead of relative phase 0. Thus, if the module is instantiated at phase 1, the variables for `define-for-syntax` are created at phase 2, and so on. Moreover, this relative phase acts as another layer of prefixing, so that a `define` of `x` and a `define-for-syntax` of `x` can co-exist in a module without colliding. Again, the higher phases are mainly related to program parsing, instead of normal evaluation.

If a module instantiated at phase *n* `require`s another module, then the `require`d module is first instantiated at phase *n*, and so on transitively. (Module `require`s cannot form cycles.) If a module instantiated at phase *n* `require-for-syntax`es another module, the other module is first instantiated at phase *n*+1, and so on. If a module instantiated at phase *n* for non-zero *n* `require-for-template`s another module, the other module is first instantiated at phase *n*-1, and so on.

A final distinction among module instantiations is that multiple instantiations may exist at phase 1 and higher. These instantiations are created by the parsing of module forms (see §1.2.3.8 "Module Phases"), and are, again, conceptually distinguished by prefixes.

[

Module Re-declarations]Module Re-declarations

When a module is declared using a name for which a module is already declared, the new declaration's definitions replace and extend the old declarations. If a variable in the old declaration has no counterpart in the new declaration, the old variable continues to exist, but its binding is not included in the lexical information for the module body. If a new variable definition has a counterpart in the old declaration, it effectively assigns to the old variable.

If a module is instantiated in any phases before it is re-declared, each re-declaration of the module is immediately instantiated in the same phases.

1.1.11 Continuation Frames and Marks

Every continuation C can be partitioned into *continuation frames* C_1, C_2, ..., C_n such that $C = C_1[C_2[...[C_n]]]$, and no frame C_i can be itself partitioned into smaller continuations. Evaluation steps add and remove frames to the current continuation, typically one at a time.

Each frame is conceptually annotated with a set of *continuation marks*. A mark consists of a key and its value; the key is an arbitrary value, and each frame includes at most one mark for any key. Various operations set and extract marks from continuations, so that marks can be used to attach information to a dynamic extent. For example, marks can be used to record information for a "stack trace" to be used when an exception is raised, or to implement dynamic scope.

1.1.12 Prompts, Delimited Continuations, and Barriers

A *prompt* is a special kind of continuation frame that is annotated with a specific *prompt-tag* (essentially a continuation mark). Various operations allow the capture of frames in the continuation from the redex position out to the nearest enclosing prompt with a particular prompt tag; such a continuation is sometimes called a *delimited continuation*. Other operations allow the current continuation to be extended with a captured continuation (specifically, a *composable continuation*). Yet other operations abort the computation to the nearest enclosing prompt with a particular tag, or replace the continuation to the nearest enclosing prompt with another one. When a delimited continuation is captured, the marks associated with the relevant frames are also captured.

A *continuation barrier* is another kind of continuation frame that prohibits certain replacements of the current continuation with another. Specifically, while an abort is allowed to remove a portion of the continuation containing a prompt, the continuation can be replaced by another only when the replacement also includes the continuation barrier. Certain operations install barriers automatically; in particular, when an exception handler is called, a continuation barrier prohibits the continuation of the handler from capturing the continuation past the exception point.

A *escape continuation* is essentially a derived concept. It combines a prompt for escape purposes with a continuation for mark-gathering purposes. as the name implies, escape continuations are used only to abort to the point of capture, which means that escape-continuation aborts can cross continuation barriers.

1.1.13 Threads

Scheme supports multiple, pre-emptive *threads* of evaluation. In terms of the evaluation model, this means that each step in evaluation actually consists of multiple concurrent expressions, rather than a single expression. The expressions all share the same objects and top-level variables, so that they can communicate through shared state. Most evaluation steps involve a single step in a single expression, but certain synchronization primitives require multiple threads to progress together in one step.

In addition to the state that is shared among all threads, each thread has its own private state that is accessed through *thread cells*. A thread cell is similar to a normal mutable object, but a change to the value inside a thread cell is seen only when extracting a value from the cell from the same thread. A thread cell can be *preserved*; when a new thread is created, the creating thread's value for a preserved thread cell serves as the initial value for the cell in the created thread. For a non-preserved thread cell, a new thread sees the same initial value (specified when the thread cell is created) as all other threads.

1.1.14 Parameters

Parameters are essentially a derived concept in Scheme; they are defined in terms of continuation marks and thread cells. However, parameters are also built in, in the sense that some primitive procedures consult parameter values. For example, the default output stream for primitive output operations is determined by a parameter.

A parameter is a setting that is both thread-specific and continuation-specific. In the empty continuation, each parameter corresponds to a preserved thread cell; a corresponding *parameter procedure* accesses and sets the thread cell's value for the current thread.

In a non-empty continuation, a parameter's value is determined through a *parameterization* that is associated with the nearest enclosing continuation frame though a continuation mark (whose key is not directly accessible). A parameterization maps each parameter to a preserved thread cell, and the combination of thread cell and current thread yields the parameter's value. A parameter procedure sets or accesses the relevant thread cell for its parameter.

Various operations, such as `parameterize` or `with-parameterization`, install a parameterization into the current continuation's frame.

1.1.15 Exceptions

Exceptions are essentially a derived concept in Scheme; they are defined in terms of continuations, prompts, and continuation marks. However, exceptions are also built in, in the sense that primitive forms and procedures may raise exceptions.

A handler for uncaught exceptions is designated through a built-in parameter. A handler to catch exceptions can be associated with a continuation frame though a continuation mark (whose key is not directly accessible). When an exception is raised, the current continuation's marks determine a chain of handler procedures that are consulted to handle the exception.

One potential action of an exception handler is to abort the current continuation up to an enclosing prompt with a particular tag. The default handler for uncaught exceptions, in particular, aborts to a particular tag for which a prompt is always present, because the prompt is installed in the outermost frame of the continuation for any new thread.

1.1.16 Custodians

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

In MrEd, custodians also manage eventspaces.

Except for the root custodian, every custodian itself it managed by a custodian, so that custodians form a hierarchy. Every object managed by a subordinate custodian is also managed by the custodian's owner.

When a custodian is shut down via `custodian-shutdown-all`, it forcibly and immediately closes the ports, TCP connections, etc. that it manages, as well as terminating (or suspending) its threads. A custodian that has been shut down cannot manage new objects. If the current custodian is shut down before a procedure is called to create a managed resource (e.g., `open-input-port`, `thread`), the `exn:fail:contract` exception is raised.

A thread can have multiple managing custodians, and a suspended thread created with `thread/suspend-to-kill` can have zero custodians. Extra custodians become associated with a thread through `thread-resume` (see §10.1.2 "Suspending, Resuming, and Killing Threads"). When a thread has multiple custodians, it is not necessarily killed by a `custodian-shutdown-all`, but shut-down custodians are removed from the thread's managing set, and the thread is killed when its managing set becomes empty.

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

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

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

1.2 Syntax Model

The syntax of a Scheme program is defined by

- a *read* phase that processes a character stream into a syntax object; and
- an *expand* phase that processes a syntax object to produce one that is fully parsed.

For details on the read phase, see §12.6 "The Reader". Source code is normally read in `read-syntax` mode, which produces a syntax object.

The expand phase recursively processes a syntax object to produce a complete parse of the program. Binding information in a syntax object drives the expansion process, and when the expansion process encounters a binding form, it extends syntax objects for sub-expression with new binding information.

1.2.1 Identifiers and Binding

§4.2 "Identifiers and Binding" in §"**Guide**: PLT Scheme" introduces binding.

An *identifier* is source-program entity. Parsing (i.e., expanding) a Scheme program reveals that some identifiers correspond to variables, some refer to syntactic forms, and some are quoted to produce a symbol or a syntax object.

An identifier *binds* another (i.e., it is a *binding*) when the former is parsed as a variable and the latter is parsed as a reference to the former; the latter is *bound*. The *scope* of a binding is the set of source forms to which it applies. The *environment* of a form is the set of bindings whose scope includes the form. A binding for a sub-expression *shadows* any bindings (i.e., it is *shadowing*) in its environment, so that uses of an identifier refer to the shadowing binding. A *top-level binding* is a binding from a definition at the top-level; a *module binding* is a binding from a definition in a module; and a *local binding* is another other kind of binding.

For example, as a bit of source, the text

```
(let ([x 5]) x)
```

includes two identifiers: `let` and `x` (which appears twice). When this source is parsed in a typical environment, `x` turns out to represent a variable (unlike `let`). In particular, the first `x` binds the second `x`.

Throughout the documentation, identifiers are typeset to suggest the way that they are parsed. A black, boldface identifier like `lambda` indicates as a reference to a syntactic form. A plain blue identifier like `x` is a variable or a reference to an unspecified top-level variable. A hyperlinked identifier `cons` is a reference to a specific top-level variable.

Every binding has a *phase level* in which it can be referenced, where a phase level normally corresponds to an integer (but the special *label phase level* does not correspond to an integer). Phase level 0 corresponds to the run time of the enclosing module (or the run time of top-level expression). Bindings in phase level 0 constitute the *base environment*. Phase level 1 corresponds to the time during which the enclosing module (or top-level expression) is expanded; bindings in phase level 1 constitute the *transformer environment*. Phase level

-1 corresponds to the run time of a different module for which the enclosing module is imported for use at phase level 1 (relative to the importing module); bindings in phase level -1 constitute the *template environment*. The label phase level does not correspond to any execution time; it is used to track bindings (e.g., to identifiers within documentation) without implying an execution dependency.

If an identifier has a local binding, then it is the same for all phase levels, though the reference is allowed only at a particular phase level. Attempting to reference a local binding in a different phase level than the binding's context produces a syntax error. If an identifier has a top-level binding or module binding, then it can have different such bindings in different phase levels.

1.2.2 Syntax Objects

A *syntax object* combines a simpler Scheme value, such as a symbol or pair, with *lexical information* about bindings, source-location information, syntax properties, and syntax certificates. In particular, an identifier is represented as a symbol object that combines a symbol and lexical and other information.

For example, a `car` identifier might have lexical information that designates it as the `car` from the `scheme/base` language (i.e., the built-in `car`). Similarly, a `lambda` identifier's lexical information may indicate that it represents a procedure form. Some other identifier's lexical information may indicate that it references a top-level variable.

When a syntax object represents a more complex expression than an identifier or simple constant, its internal components can be extracted. Even for extracted identifier, detailed information about binding is available mostly indirectly; two identifiers can be compared to see if they refer to the same binding (i.e., `free-identifier=?`), or whether each identifier would bind the other if one was in a binding position and the other in an expression position (i.e., `bound-identifier=?`).

For example, the when the program written as

```
(let ([x 5]) (+ x 6))
```

is represented as a syntax object, then two syntax objects can be extracted for the two `x`s. Both the `free-identifier=?` and `bound-identifier=?` predicates will indicate that the `x`s are the same. In contrast, the `let` identifier is not `free-identifier=?` or `bound-identifier=?` to either `x`.

The lexical information in a syntax object is independent of the other half, and it can be copied to a new syntax object in combination with an arbitrary other Scheme value. Thus, identifier-binding information in a syntax object is predicated on the symbolic name of the identifier as well as the identifier's lexical information; the same question with the same lexical information but different base value can produce a different answer.

For example, combining the lexical information from `let` in the program above to `'x` would not produce an identifier that is `free-identifier=?` to either `x`, since it does not appear in the scope of the `x` binding. Combining the lexical context of the `6` with `'x`, in contrast, would produce an identifier that is `bound-identifier=?` to both `x`s.

The `quote-syntax` form bridges the evaluation of a program and the representation of a program. Specifically, `(quote-syntax datum)` produces a syntax object that preserves all of the lexical information that *datum* had when it was parsed as part of the `quote-syntax` form.

1.2.3 Expansion (Parsing)

Expansion recursively processes a syntax object in a particular phase level, starting with phase level 0. Bindings from the syntax object's lexical information drive the expansion process, and cause new bindings to be introduced for the lexical information of sub-expressions. In some cases, a sub-expression is expanded in a deeper phase than the enclosing expression.

[

Fully Expanded Programs]Fully Expanded Programs

A complete expansion produces a syntax object matching the following grammar:

Beware that the symbolic names of identifiers in a fully expanded program may not match the symbolic names in the grammar. Only the binding (according to `free-identifier=?`) matters.

```
       top-level-form = general-top-level-form
                      | (#%expression expr)
                      | (module id name-id
                          (#%plain-module-begin
                           module-level-form ...))
                      | (begin top-level-form ...)

    module-level-form = general-top-level-form
                      | (#%provide raw-provide-spec ...)

general-top-level-form = expr
                      | (define-values (id ...) expr)
                      | (define-syntaxes (id ...) expr)
                      | (define-values-for-syntax (id ...) expr)
                      | (#%require raw-require-spec ...)

                 expr = id
                      | (#%plain-lambda formals expr ...+)
                      | (case-lambda (formals expr ...+) ...)
                      | (if expr expr expr)
```

```
          | (begin expr ...+)
          | (begin0 expr expr ...)
          | (let-values (((id ...) expr) ...)
              expr ...+)
          | (letrec-values (((id ...) expr) ...)
              expr ...+)
          | (set! id expr)
          | (quote datum)
          | (quote-syntax datum)
          | (with-continuation-mark expr expr expr)
          | (#%plain-app expr ...+)
          | (#%top . id)
          | (#%variable-reference id)
          | (#%variable-reference (#%top . id))

formals = (id ...)
          | (id ...+ . id)
          | id
```

A fully-expanded syntax object corresponds to a *parse* of a program (i.e., a *parsed* program), and lexical information on its identifiers indicates the parse.

More specifically, the typesetting of identifiers in the above grammar is significant. For example, the second case for *expr* is a syntax-object list whose first element is an identifier, where the identifier's lexical information specifies a binding to the `define-values` of the `scheme/base` language (i.e., the identifier is `free-identifier=?` to one whose binding is `define-values`). In all cases, identifiers above typeset as syntactic-form names refer to the bindings defined in §2 "Syntactic Forms".

Only phase levels 0 and 1 are relevant for the parse of a program (though the *datum* in a `quote-syntax` form preserves its information for all phase levels). In particular, the relevant phase level is 0, except for the *expr*s in a `define-syntax`, `define-syntaxes`, `define-for-syntax`, or `define-values-for-syntax` form, in which case the relevant phase level is 1 (for which comparisons are made using `free-transformer-identifier=?` instead of `free-identifier=?`).

[

Expansion Steps]Expansion Steps

In a recursive expansion, each single step in expanding a syntax object at a particular phase level depends on the immediate shape of the syntax object being expanded:

- If it is an identifier (i.e., a syntax-object symbol), then a binding is determined by the identifier's lexical information. If the identifier has a binding other than as a top-level variable, that binding is used to continue. If the identifier has no binding, a new syntax-object symbol `'#%top` is created using the lexical information of the identifier; if this `#%top` identifier has no binding (other than as a top-level variable), then parsing fails with an `exn:fail:syntax` exception. Otherwise, the new identifier is combined with the original identifier in a new syntax-object pair (also using the same lexical information as the original identifier), and the `#%top` binding is used to continue.

- If it is a syntax-object pair whose first element is an identifier, and if the identifier has a binding other than as a top-level variable, then the identifier's binding is used to continue.

- If it is a syntax-object pair of any other form, then a new syntax-object symbol `'#%app` is created using the lexical information of the pair. If the resulting `#%app` identifier has no binding, parsing fails with an `exn:fail:syntax` exception. Otherwise, the new identifier is combined with the original pair to form a new syntax-object pair (also using the same lexical information as the original pair), and the `#%app` binding is used to continue.

- If it is any other syntax object, then a new syntax-object symbol `'#%datum` is created using the lexical information of the original syntax object. If the resulting `#%datum` identifier has no binding, parsing fails with an `exn:fail:syntax` exception. Otherwise, the new identifier is combined with the original syntax object in a new syntax-object pair (using the same lexical information as the original pair), and the `#%datum` binding is used to continue.

Thus, the possibilities that do not fail lead to an identifier with a particular binding. This binding refers to one of three things:

- A transformer binding, such as introduced by `define-syntax` or `let-syntax`. If the associated value is a procedure of one argument, the procedure is called as a syntax transformer (described below), and parsing starts again with the syntax-object result. If the transformer binding is to any other kind of value, parsing fails with an `exn:fail:syntax` exception.

- A variable binding, such as introduced by a module-level `define` or by `let`. In this case, if the form being parsed is just an identifier, then it is parsed as a reference to the corresponding variable. If the form being parsed is a syntax-object pair, then an `#%app` is added to the front of the syntax-object pair in the same way as when the first item in the syntax-object pair is not an identifier (third case in the previous enumeration), and parsing continues.

- A core *syntactic form*, which is parsed as described for each form in §2 "Syntactic Forms". Parsing a core syntactic form typically involves recursive parsing of sub-forms, and may introduce bindings that determine the parsing of sub-forms.

[

Expansion Context]Expansion Context

Each expansion step occurs in a particular *context*, and transformers and core syntactic forms may expand differently for different contexts. For example, a `module` form is allowed only in a top-level context, and it fails in other contexts. The possible contexts are as follows:

- *top-level context* : outside of any module, definition, or expression, except that sub-expressions of a top-level `begin` form are also expanded as top-level forms.
- *module-begin context* : inside the body of a module, as the only form within the module.
- *module context* : in the body of a module (inside the module-begin layer).
- *internal-definition context* : in a nested context that allows both definitions and expressions.
- *expression context* : in a context where only expressions are allowed.

Different core syntactic forms parse sub-forms using different contexts. For example, a `let` form always parses the right-hand expressions of a binding in an expression context, but it starts parsing the body in an internal-definition context.

[

Introducing Bindings]Introducing Bindings

Bindings are introduced during expansion when certain core syntactic forms are encountered:

- When a `require` form is encountered at the top level or module level, all lexical information derived from the top level or the specific module's level are extended with bindings from the specified modules. If not otherwise indicated in the `require` form, bindings are introduced at the phase levels specified by the exporting modules: phase level 0 for each normal `provide`, phase level 1 for each `for-syntax` `provide`, and so on. The `for-meta` `provide` form allows exports at an arbitrary phase level (as long as a binding exists within the module at the phase level).

  A `for-syntax` sub-form within `require` imports similarly, but the resulting bindings have a phase level that is one more than the exported phase levels, when exports for the label phase level are still imported at the label phase level. More generally, a `for-meta` sub-form within `require` imports with the specified phase level shift; if the specified shift is `#f`, or if `for-label` is used to import, then all bindings are imported into the label phase level.

- When a `define`, `define-values`, `define-syntax`, or `define-syntaxes` form is encountered at the top level or module level, all lexical information derived from the top level or the specific module's level is extended with bindings for the specified identifiers at phase level 0 (i.e., the base environment is extended).

- When a `define-for-syntax` or `define-values-for-syntax` form is encountered at the top level or module level, bindings are introduced as for `define-values`, but at phase level 1 (i.e., the transformer environment is extended).

- When a `let-values` form is encountered, the body of the `let-values` form is extended (by creating new syntax objects) with bindings for the specified identifiers. The same bindings are added to the identifiers themselves, so that the identifiers in binding position are `bound-identifier=?` to uses in the fully expanded form, and so they are not `bound-identifier=?` to other identifiers. The bindings are available for use at the phase level at which the `let-values` form is expanded.

- When a `letrec-values` or `letrec-syntaxes+values` form is encountered, bindings are added as for `let-values`, except that the right-hand-side expressions are also extended with the bindings.

- Definitions in `internal-definition contexts` introduce bindings as described in §1.2.3.7 "Internal Definitions".

A new binding in lexical information maps to a new variable. The identifiers mapped to this variable are those that currently have the same binding (i.e., that are currently `bound-identifier=?`) to the identifier associated with the binding.

For example, in

```
(let-values ([(x) 10]) (+ x y))
```

the binding introduced for `x` applies to the `x` in the body, but not the `y` n the body, because (at the point in expansion where the `let-values` form is encountered) the binding `x` and the body `y` are not `bound-identifier=?`.

[

Transformer Bindings]Transformer Bindings

In a top-level context or module context, when the expander encounters a `define-syntaxes` form, the binding that it introduces for the defined identifiers is a *transformer binding*. The value of the binding exists at expansion time, rather than run time (though the two times can overlap), though the binding itself is introduced with phase level 0 (i.e., in the base environment).

The value for the binding is obtained by evaluating the expression in the `define-syntaxes` form. This expression must be expanded (i.e. parsed) before it can be evaluated, and it is expanded at phase level 1 (i.e., in the transformer environment) instead of phase level 0.

The if resulting `value` is a procedure of one argument or as the result of `make-set!-transformer` on a procedure, then is it used as a *syntax transformer* (a.k.a. *macro*). The procedure is expected to accept a syntax object and return a syntax object. A use of the binding (at phase level 0) triggers a call of the syntax transformer by the expander; see §1.2.3.2 "Expansion Steps".

Before the expander passes a syntax object to a transformer, the syntax object is extend with a *syntax mark* (that applies to all sub-syntax objects). The result of the transformer is similarly extended with the same syntax mark. When a syntax object's lexical information includes the same mark twice in a row, the marks effectively cancel. Otherwise, two identifiers are `bound-identifier=?` (that is, one can bind the other) only if they have the same binding and if they have the same marks—counting only marks that were added after the binding.

This marking process helps keep binding in an expanded program consistent with the lexical structure of the source program. For example, the expanded form of the program

```
(define x 12)
(define-syntax m
  (syntax-rules ()
    [(_ id) (let ([x 10]) id)]))
(m x)
```

is

```
(define x 12)
(define-syntax m
  (syntax-rules ()
    [(_ id) (let ([x 10]) id)]))
(let-values ([(x) 10]) x)
```

However, the result of the last expression is `12`, not `10`. The reason is that the transformer bound to `m` introduces the binding `x`, but the referencing `x` is present in the argument to the transformer. The introduced `x` is the one left with a mark, and the reference `x` has no mark, so the binding `x` is not `bound-identifier=?` to the body `x`.

The `set!` form and the `make-set!-transformer` procedure work together to support *assignment transformers* that transformer `set!` expression. Assignment transformers are applied by `set!` in the same way as a normal transformer by the expander.

The `make-rename-transformer` procedure creates a value that is also handled specially by the expander and by `set!` as a transformer binding's value. When *`id`* is bound to a *rename transformer* produced by `make-rename-transformer`, it is replaced with the identifier passed to `make-rename-transformer`. Furthermore, the binding is also specially handled by `syntax-local-value` as used by syntax transformers.

In addition to using marks to track introduced identifiers, the expander tracks the expansion history of a form through syntax properties such as `'origin`. See §11.6 "Syntax Object Properties" for more information.

Finally, the expander uses syntax certificates to control the way that unexported and protected module bindings are used. See §11.7 "Syntax Certificates" for more information on syntax certificates.

The expander's handling of `letrec-values+syntaxes` is similar to its handling of `define-syntaxes`. A `letrec-values+syntaxes` mist be expanded in an arbitrary phase level n (not just 0), in which case the expression for the transformer binding is expanded at phase level $n+1$.

The expression in a `define-for-syntax` or `define-values-for-syntax` form is expanded and evaluated in the same way as for `syntax`. However, the introduced binding is a variable binding at phase level 1 (not a transformer binding at phase level 0).

[

Partial Expansion]Partial Expansion

In certain contexts, such as an internal-definition context or module context, forms are partially expanded to determine whether they represent definitions, expressions, or other declaration forms. Partial expansion works by cutting off the normal recursion expansion when the relevant binding is for a primitive syntactic form.

As a special case, when expansion would otherwise add an `#%app`, `#%datum`, or `#%top` identifier to an expression, and when the binding turns out to be the primitive `#%app`, `#%datum`, or `#%top` form, then expansion stops without adding the identifier.

[

Internal Definitions]Internal Definitions

An internal-definition context corresponds to a partial expansion step (see §1.2.3.6 "Partial Expansion"). A form that supports internal definitions starts by expanding its first form in an internal-definition context, but only partially. That is, it recursively expands only until the form becomes one of the following:

- A `define-values` or `define-syntaxes` form, for any form other than the last one: The definition form is not expanded further. Instead, the next form is expanded partially, and so on. As soon as an expression form is found, the accumulated definition forms are converted to a `letrec-values` (if no `define-syntaxes` forms were found) or `letrec-syntaxes+values` form, moving the expression forms to the body to be expanded in expression context.

When a `define-values` form is discovered, the lexical context of all syntax objects for the body sequence is immediately enriched with bindings for the `define-values` form before expansion continues. When a `define-syntaxes` form is discovered, the right-hand side is expanded and evaluated (as for a `letrec-values+syntaxes` form_, and a transformer binding is installed for the body sequence before expansion continues.

- A primitive expression form other than `begin`: The expression is expanded in an expression context, along with all remaining body forms. If any definitions were found, this expansion takes place after conversion to a `letrec-values` or `letrec-syntaxes+values` form. Otherwise, the expressions are expanded immediately.
- A `begin` form: The sub-forms of the `begin` are spliced into the internal-definition sequence, and partial expansion continues with the first of the newly-spliced forms (or the next form, if the `begin` had no sub-forms).

If the last expression form turns out to be a `define-values` or `define-syntaxes` form, expansion fails with a syntax error.

[

Module Phases]Module Phases

A `require` form not only introduces bindings at expansion time, but also *visits* the referenced module when it is encountered by the expander. That is, the expander instantiates any `define-for-syntax`ed variables defined in the module, and also evaluates all expressions for `define-syntaxes` transformer bindings.

Module visits propagate through `require`s in the same way as module instantiation. Moreover, when a module is visited, any module that it `require-for-syntax`es is instantiated at phase 1, which the adjustment that `require-for-template` leading back to phase 0 causes the required module to be merely visited at phase 0, not instantiated.

When the expander encounters `require-for-syntax`, it immediately instantiates the required module at phase 1, in addition to adding bindings scheme phase level 1 (i.e., the transformer environment).

When the expander encounters `require` and `require-for-syntax` within a module context, the resulting visits and instantiations are specific to the expansion of the enclosing module, and are kept separate from visits and instantiations triggered from a top-level context or from the expansion of a different module.

1.2.4 Compilation

Before expanded code is evaluated, it is first *compiled*. A compiled form has essentially the same information as the corresponding expanded form, though the internal representation naturally dispenses with identifiers for syntactic forms and local bindings. One significant difference is that a compiled form is almost entirely opaque, so the information that it contains cannot be accessed directly (which is why some identifiers can be dropped). At the same time, a compiled form can be marshaled to and from a byte string, so it is suitable for saving and re-loading code.

Although individual read, expand, compile, and evaluate operations are available, the operations are often combined automatically. For example, the `eval` procedure takes a syntax object and expands it, compiles it, and evaluates it.

1.2.5 Namespaces

A *namespace* is a top-level mapping from symbols to binding information. It is the starting point for expanding an expression; a syntax object produced by `read-syntax` has no initial lexical context; the syntax object can be expanded after initializing it with the mappings of a particular namespace. A namespace is also the starting point evaluating expanded code, where the first step in evaluation is linking the code to specific module instances and top-level variables.

For expansion purposes, a namespace maps each symbol in each phase level to one of three possible bindings:

- a particular module binding from a particular module
- a top-level transformer binding named by the symbol
- a top-level variable named by the symbol

An “empty” namespace maps all symbols to top-level variables. Certain evaluations extend a namespace for future expansions; importing a module into the top-level adjusts the namespace bindings for all of the imported named, and evaluating a top-level `define` form updates the namespace’s mapping to refer to a variable (in addition to installing a value into the variable).

A namespace also has a *module registry* that maps module names to module declarations (see §1.1.10 “Modules and Module-Level Variables”). This registry is shared by all phase levels.

For evaluation, each namespace encapsulates a distinct set of top-level variables, as well as a potentially distinct set of module instances in each phase. That is, even though module declarations are shared for all phase levels, module instances are distinct for each phase.

After a namespace is created, module instances from existing namespaces can be attached to the new namespace. In terms of the evaluation model, top-level variables from different namespaces essentially correspond to definitions with different prefixes. Furthermore, the first step in evaluating any compiled expression is to link its top-level variable and module-level variable references to specific variables in the namespace.

At all times during evaluation, some namespace is designated as the *current namespace*. The current namespace has no particular relationship, however, with the namespace that was used to expand the code that is executing, or with the namespace that was used to link the compiled form of the currently evaluating code. In particular, changing the current namespace during evaluation does not change the variables to which executing expressions refer. The current namespace only determines the behavior of (essentially reflective) operations to expand code and to start evaluating expanded/compiled code.

Examples:

```
> (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-base-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
orignew
```

A namespace is purely a top-level entity, not to be confused with an environment. In particular, a namespace does not encapsulate the full environment of an expression inside local-binding forms.

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

Examples:

```
> (define x 5)
> (define (f) x)
> x
5
> (f)
5
> (define-syntax x (syntax-id-rules () [_ 10]))
> x
10
> (f)
5
```

```
> (define x 7)
> x
7
> (f)
7
> (module m mzscheme (define x 8) (provide x))
> (require 'm)
> x
8
> (f)
7
```

1.2.6 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, Scheme decides, at compile-time, to name as `'f` all procedures created by the `let`-bound `lambda`.

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

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

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

When an `'inferred-name` property is attached to a syntax object for an expression (see §11.6 "Syntax Object Properties"), the property value is used for naming the expression, and it overrides any name that was inferred from the expression's context.

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

2 Syntactic Forms

This section describes the core syntax forms that appear in a fully expanded expression, plus a many closely-related non-core forms. See §1.2.3.1 “Fully Expanded Programs” for the core grammar.

Notation

Each syntactic form is described by a BNF-like notation that describes a combination of (syntax-wrapped) pairs, symbols, and other data (not a sequence of characters). These grammatical specifications are shown as follows:

```
(some-form id ...)
```

Within such specifications,

- `...` indicates zero or more repetitions of the preceding datum.
- `...+` indicates one or more repetitions of the preceding datum.
- italic meta-identifiers play the role of non-terminals; in particular,
  - a meta-identifier that ends in *`id`* stands for an identifier.
  - a meta-identifier that ends in *`keyword`* stands for a keyword.
  - a meta-identifier that ends with *`expr`* stands for a sub-form that is expanded as an expression.
  - A meta-identifier that ends with *`body`* stands for a sub-form that is expanded in an internal-definition context (see §1.2.3.7 “Internal Definitions”).

2.1 Literals: `quote` and `#%datum`

Many forms are implicitly quoted (via `#%datum`) as literals. See §1.2.3.2 “Expansion Steps” for more information.

§4.10 “Quoting: `quote` and `'`” in §“**Guide**: PLT Scheme” introduces `quote`.

```
(quote datum)
```

Produces a constant value corresponding to *`datum`* (i.e., the representation of the program fragment) without its lexical information, source location, etc. Quoted pairs, vectors, and boxes are immutable.

Examples:

```
> (quote x)
x
> (quote (+ 1 2))
(+ 1 2)
> (+ 1 2)
3
```

`(#%datum . datum)`

Expands to `(quote datum)`, as long as `datum` is not a keyword. If `datum` is a keyword, a syntax error is reported.

See also §1.2.3.2 "Expansion Steps" for information on how the expander introduces `#%datum` identifiers.

Examples:

```
> (#%datum . 10)
10
> (#%datum . x)
x
> (#%datum . #:x)
eval:7:0: #%datum: keyword used as an expression in: #:x
```

2.2 Expression Wrapper: `#%expression`

`(#%expression expr)`

Produces the same result as `expr`. The only use of `#%expression` is to force the parsing of a form as an expression.

Examples:

```
> (#%expression (+ 1 2))
3
> (#%expression (define x 10))
eval:4:0: define: not allowed in an expression context in:
(define x 10)
```

2.3 Variable References and `#%top`

`id`

Refers to a module-level or local binding, when *id* is not bound as a transformer (see §1.2.3 "Expansion"). At run-time, the reference evaluates to the value in the location associated with the binding.

When the expander encounters an *id* that is not bound by a module-level or local binding, it converts the expression to `(#%top . id)` giving `#%top` the lexical context of the *id*; typically, that context refers to `#%top`. See also §1.2.3.2 "Expansion Steps".

Examples:

```
> (define x 10)
> x
10
> (let ([x 5]) x)
5
> ((lambda (x) x) 2)
2
```

```
(#%top . id)
```

Refers to a top-level definition that could bind *id*, even if *id* has a local binding in its context. Such references are disallowed anywhere within a `module` form. See also §1.2.3.2 "Expansion Steps" for information on how the expander introduces `#%top` identifiers.

Examples:

```
> (define x 12)
> (let ([x 5]) (#%top . x))
12
```

2.4 Locations: `#%variable-reference`

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

Produces an opaque value representing the location of *id*, which must be bound as a top-level variable or module-level variable.

The result is useful to low-level extensions; see §"**Inside**: PLT Scheme C API". It can also be used with `variable-reference->empty-namespace`, `variable-reference->resolved-module-path`, and `variable-reference->top-level-namespace`, but facilities like `define-namespace-anchor` and `namespace-anchor->namespace` wrap those to provide an clearer interface.

2.5 Procedure Applications and #%app

§4.3 "Function Calls" in §"**Guide**: PLT Scheme" introduces procedure applications.

```
(proc-expr arg ...)
```

Applies a procedure, when *proc-expr* is not an identifier that has a transformer binding (see §1.2.3 "Expansion").

More precisely, the expander converts this form to `(#%app proc-expr arg ...)`, giving `#%app` the lexical context that is associated with the original form (i.e., the pair that combines *proc-expr* and its arguments). Typically, the lexical context of the pair indicates the procedure-application `#%app` that is described next. See also §1.2.3.2 "Expansion Steps".

Examples:

```
> (+ 1 2)
3
> ((lambda (x #:arg y) (list y x)) #:arg 2 1)
(2 1)
```

```
(#%app proc-expr arg ...)
```

Applies a procedure. Each *arg* is one of the following:

arg-expr

The resulting value is a non-keyword argument.

keyword arg-expr

The resulting value is a keyword argument using *keyword*. Each *keyword* in the application must be distinct.

The *proc-expr* and *arg-expr*s are evaluated in order, left to right. If the result of *proc-expr* is a procedure that accepts as many arguments as non-*keyword* *arg-expr*s, if it accepts arguments for all of the *keyword*s in the application, and if all required keyword-based arguments are represented among the *keyword*s in the application, then the procedure is called with the values of the `arg-expr`s. Otherwise, the `exn:fail:contract` exception is raised.

The continuation of the procedure call is the same as the continuation of the application expression, so the results of the procedure are the results of the application expression.

The relative order of *keyword*-based arguments matters only for the order of *arg-expr* evaluations; the arguments are associated with argument variables in the applied procedure based on the *keyword*s, and not their positions. The other *arg-expr* values, in contrast, are associated with variables according to their order in the application form.

See also §1.2.3.2 "Expansion Steps" for information on how the expander introduces #%app identifiers.

Examples:

```
> (#%app + 1 2)
3
> (#%app (lambda (x #:arg y) (list y x)) #:arg 2 1)
(2 1)
> (#%app cons)
cons: expects 2 arguments, given 0
```

```
(#%plain-app proc-expr arg-expr ...)
(#%plain-app)
```

Like #%app, but without support for keyword arguments. As a special case, (#%plain-app) produces '().

2.6 Procedure Expressions: lambda and case-lambda

§4.4 "Functions: lambda" in §"**Guide**: PLT Scheme" introduces procedure expressions.

```
(lambda kw-formals body ...+)
(λ kw-formals body ...+)

kw-formals = (arg ...)
           | (arg ...+ . rest-id)
           | rest-id

       arg = id
           | [id default-expr]
           | keyword id
           | keyword [id default-expr]
```

Produces a procedure. The *kw-formals* determines the number of arguments and which keyword arguments that the procedure accepts.

Considering only the first *arg* case, a simple *kw-formals* has one of the following three forms:

`(id ...)`

The procedure accepts as many non-keyword argument values as the number of `id`s. Each `id` is associated with an argument value by position.

`(id ...+ . rest-id)`

The procedure accepts any number of non-keyword arguments greater or equal to the number of `id`s. When the procedure is applied, the `id`s are associated with argument values by position, and all leftover arguments are placed into a list that is associated to `rest-id`.

`rest-id`

The procedure accepts any number of non-keyword arguments. All arguments are placed into a list that is associated with `rest-id`.

More generally, an `arg` can include a keyword and/or default value. Thus, the first two cases above are more completely specified as follows:

`(arg ...)`

Each `arg` has the following four forms:

`id`

Adds one to both the minimum and maximum number of non-keyword arguments accepted by the procedure. The `id` is associated with an actual argument by position.

`[id default-expr]`

Adds one to the maximum number of non-keyword arguments accepted by the procedure. The `id` is associated with an actual argument by position, and if no such argument is provided, the `default-expr` is evaluated to produce a value associated with `id`. No `arg` with a `default-expr` can appear before an `id` without a `default-expr` and without a `keyword`.

`keyword id`

The procedure requires a keyword-based argument using *keyword*. The *id* is associated with a keyword-based actual argument using *keyword*.

`keyword [id default-expr]`

The procedure accepts a keyword-based using *keyword*. The *id* is associated with a keyword-based actual argument using *keyword*, if supplied in an application; otherwise, the *default-expr* is evaluated to obtain a value to associate with *id*.

The position of a *keyword arg* in *kw-formals* does not matter, but each specified *keyword* must be distinct.

`(arg ...+ . rest-id)`

Like the previous case, but the procedure accepts any number of non-keyword arguments beyond its minimum number of arguments. When more arguments are provided than non-*keyword* arguments among the *arg*s, the extra arguments are placed into a list that is associated to *rest-id*.

The *kw-formals* identifiers are bound in the *body*s. When the procedure is applied, a new location is created for each identifier, and the location is filled with the associated argument value.

If any identifier appears in the *body*s that is not one of the identifiers in *kw-formals*, then it refers to the same location that it would if it appeared in place of the `lambda` expression. (In other words, variable reference is lexically scoped.)

When multiple identifiers appear in a *kw-formals*, they must be distinct according to `bound-identifier=?`.

If the procedure procedure by `lambda` is applied to fewer or more by-position or arguments than it accepts, to by-keyword arguments that it does not accept, or without required by-keyword arguments, then the `exn:fail:contract` exception is raised.

The last *body* expression is in tail position with respect to the procedure body.

Examples:

```
> ((lambda (x) x) 10)
10
> ((lambda (x y) (list y x)) 1 2)
```

```
(2 1)
> ((lambda (x [y 5]) (list y x)) 1 2)
(2 1)
> (let ([f (lambda (x #:arg y) (list y x))])
    (list (f 1 #:arg 2)
          (f #:arg 2 1)))
((2 1) (2 1))
```

When compiling a `lambda` or `case-lambda` expression, Scheme looks for a `'method-arity-error` property attached to the expression (see §11.6 "Syntax Object Properties"). If it is present with a true value, and if no case of the procedure accepts zero arguments, then the procedure is marked so that an `exn:fail:contract:arity` exception involving the procedure will hide the first argument, if one was provided. (Hiding the first argument is useful when the procedure implements a method, where the first argument is implicit in the original source). The property affects only the format of `exn:fail:contract:arity` exceptions, not the result of `procedure-arity`.

```
(case-lambda [formals body ...+] ...)

formals = (id ...)
        | (id ...+ . rest-id)
        | rest-id
```

Produces a procedure. Each `[forms body ...+]` clause is analogous to a single `lambda` procedure; applying the `case-lambda`-generated procedure is the same as applying a procedure that corresponds to one of the clauses—the first procedure that accepts the given number of arguments. If no corresponding procedure accepts the given number of arguments, the `exn:fail:contract` exception is raised.

Note that a `case-lambda` clause supports only *`formals`*, not the more general *`kw-formals`* of `lambda`. That is, `case-lambda` does not directly support keyword and optional arguments.

Examples:

```
> (let ([f (case-lambda
             [() 10]
             [(x) x]
             [(x y) (list y x)]
             [r r])])
    (list (f)
          (f 1)
          (f 1 2)
          (f 1 2 3)))
(10 1 (2 1) (1 2 3))
```

```
(#%plain-lambda formals body ...+)
```

Like `lambda`, but without support for keyword or optional arguments.

2.7 Local Binding: `let`, `let*`, `letrec`, ...

§4.6 "Local Binding" in §"**Guide**: PLT Scheme" introduces local binding.

```
(let ([id val-expr] ...) body ...+)
(let proc-id ([id init-expr] ...) body ...+)
```

The first form evaluates the *val-expr*s left-to-right, creates a new location for each *id*, and places the values into the locations. It then evaluates the *body*s, in which the *id*s are bound. The last *body* expression is in tail position with respect to the `let` form. The *id*s must be distinct according to `bound-identifier=?`.

Examples:

```
> (let ([x 5]) x)
5
> (let ([x 5])
    (let ([x 2]
          [y x])
      (list y x)))
(5 2)
```

The second form evaluates the *init-expr*s; the resulting values become arguments in an application of a procedure `(lambda (id ...) body ...+)`, where *proc-id* is bound within the *body*s to the procedure itself.

Examples:

```
> (let fac ([n 10])
    (if (zero? n)
        1
        (* n (fac (sub1 n)))))
3628800
```

```
(let* ([id val-expr] ...) body ...+)
```

Similar to `let`, but evaluates the *val-expr*s one by one, creating a location for each *id* as soon as the value is available. The *id*s are bound in the remaining *val-expr*s as well as the *body*s, and the *id*s need not be distinct; later bindings shadow earlier bindings.

Examples:

```
> (let* ([x 1]
         [y (+ x 1)])
```

```
    (list y x))
(2 1)
```

```
(letrec ([id val-expr] ...) body ...+)
```

Similar to `let`, but the locations for all *id*s are created first and filled with `#<undefined>`, and all *id*s are bound in all *val-expr*s as well as the *body*s. The *id*s must be distinct according to `bound-identifier=?`.

Examples:

```
> (letrec ([is-even? (lambda (n)
                       (or (zero? n)
                           (is-odd? (sub1 n))))]
           [is-odd? (lambda (n)
                      (or (= n 1)
                          (is-even? (sub1 n))))])
    (is-odd? 11))
#t
```

```
(let-values ([(id ...) val-expr] ...) body ...+)
```

Like `let`, except that each *val-expr* must produce as many values as corresponding *id*s, otherwise the `exn:fail:contract` exception is raised. A separate location is created for each *id*, all of which are bound in the *body*s.

Examples:

```
> (let-values ([(x y) (quotient/remainder 10 3)])
    (list y x))
(1 3)
```

```
(let*-values ([(id ...) val-expr] ...) body ...+)
```

Like `let*`, except that each *val-expr* must produce as many values as corresponding *id*s. A separate location is created for each *id*, all of which are bound in the later *val-expr*s and in the *body*s.

Examples:

```
> (let*-values ([(x y) (quotient/remainder 10 3)]
                [(z) (list y x)])
    z)
(1 3)
```

```
(letrec-values ([(id ...) val-expr] ...) body ...+)
```

Like `letrec`, except that each *val-expr* must produce as many values as corresponding *id*s. A separate location is created for each *id*, all of which are initialized to `#<undefined>` and bound in all *val-expr*s and in the *body*s.

Examples:

```
> (letrec-values ([(is-even? is-odd?)
                   (values
                     (lambda (n)
                       (or (zero? n)
                           (is-odd? (sub1 n))))
                     (lambda (n)
                       (or (= n 1)
                           (is-even? (sub1 n)))))])
    (is-odd? 11))
#t
```

```
(let-syntax ([id trans-expr] ...) body ...+)
```

Creates a transformer binding (see §1.2.3.5 "Transformer Bindings") of each *id* with the value of *trans-expr*, which is an expression at phase level 1 relative to the surrounding context. (See §1.2.1 "Identifiers and Binding" for information on phase levels.)

Each *id* is bound in the *body*s, and not in other *trans-expr*s.

```
(letrec-syntax ([id trans-expr] ...) body ...+)
```

Like `let-syntax`, except that each *id* is also bound within all *trans-expr*s.

```
(let-syntaxes ([(id ...) trans-expr] ...) body ...+)
```

Like `let-syntax`, but each *trans-expr* must produce as many values as corresponding *id*s, each of which is bound to the corresponding value.

```
(letrec-syntaxes ([(id ...) trans-expr] ...) body ...+)
```

Like `let-syntax`, except that each *id* is also bound within all *trans-expr*s.

```
(letrec-syntaxes+values ([(trans-id ...) trans-expr] ...)
                        ([(val-id ...) val-expr] ...)
   body ...+)
```

Combines `letrec-syntaxes` with `letrec-values`: each *trans-id* and *val-id* is bound in all *trans-expr*s and *val-expr*s.

See also `local`, which supports local bindings with `define`, `define-syntax`, and more.

2.8 Local Definitions: `local`

```
(require scheme/local)
```

The bindings documented in this section are provided by the `scheme/local` and `scheme` libraries, but not `scheme/base`.

```
(local [definition ...] body ...+)
```

Like `letrec`, except that the bindings are expressed in the same way as in the top-level or in a module body: using `define`, `define-values`, `define-syntax`, `define-struct`, etc. Definitions are distinguished from non-definitions by partially expanding *definition* forms (see §1.2.3.6 “Partial Expansion”). As in the top-level or in a module body, a `begin`-wrapped sequence is spliced into the sequence of *definition*s.

2.9 Constructing Graphs: `shared`

```
(require scheme/shared)
```

The bindings documented in this section are provided by the `scheme/shared` and `scheme` libraries, but not `scheme/base`.

```
(shared ([id expr] ...) body ...+)
```

Binds *id*s with shared structure according to `exprs` and then evaluates the `body-exprs`, returning the result of the last expression.

The `shared` form is similar to `letrec`, except that special forms of *expr* are recognized (after partial macro expansion) to construct graph-structured data, where the corresponding `letrec` would instead produce `#<undefined>`s.

Each *expr* (after partial expansion) is matched against the following *shared-expr* grammar, where earlier variants in a production take precedence over later variants:

```
shared-expr = shell-expr
            | plain-expr

 shell-expr = (cons in-immutable-expr in-immutable-expr)
```

```
                    | (list in-immutable-expr ...)
                    | (vector-immutable in-immutable-expr ...)
                    | (box-immutable in-immutable-expr)
                    | (mcons patchable-expr)
                    | (vector patchable-expr ...)
                    | (box patchable-expr ...)
                    | (prefix:make-id patchable-expr ...)

in-immutable-expr = shell-id
                    | shell-expr
                    | early-expr

         shell-id = id

   patchable-expr = expr

       early-expr = expr

       plain-expr = expr
```

The *prefix*:make-*id* identifier above references to any binding whose name has make- in the middle, and where *prefix*:*id* has a transformer binding to structure information with a full set of mutator bindings; see §4.6 "Structure Type Transformer Binding". A *shell-id* must be one of the *id*s bound by the shared form to a *shell-expr*.

When the *expr*s of the shared form are parsed via *shared-expr* (taking into account the order of the variants for precedence), and sub-expression that parses via *early-expr* will be evaluated first when the shared form is evaluated. Among such expressions, they are evaluated in the order as they appear within the shared form. However, any reference to an *id* bound by shared produces #<undefined>, even if the binding for the *id* appears before the corresponding *early-expr* within the shared form.

The *shell-ids* and *shell-exprs* (not counting *patchable-expr* and *early-expr* sub-expressions) are effectively evaluated next. A *shell-id* reference produces the same value as the corresponding *id* will produce within the *body*s, assuming that *id* is never mutated with set!. This special handling of a *shell-id* reference is one way in which shared supports the creation of cyclic data, including immutable cyclic data.

Next, the *plain-expr*s are evaluated as for letrec, where a reference to an *id* produces #<undefined> if it is evaluated before the right-hand side of the *id* binding.

Finally, the *patchable-expr*s are evaluated. At this point, all *id*s are bound, so *patchable-expr*s also created data cycles (but only with cycles that can be created via mutation).

Examples:

```
> (shared ([a (cons 1 a)])
```

```
     a)
#0=(1 . #0#)
> (shared ([a (cons 1 b)]
           [b (cons 2 a)])
     a)
#0=(1 2 . #0#)
> (shared ([a (cons 1 b)]
           [b 7])
     a)
(1 . 7)
> (shared ([a a]) ; no indirection...
     a)
#<undefined>
> (shared ([a (cons 1 b)] ; b is early...
           [b a])
     a)
(1 . #<undefined>)
> (shared ([a (mcons 1 b)] ; b is patchable...
           [b a])
     a)
#0={1 . #0#}
> (shared ([a (vector b b b)]
           [b (box 1)])
    (set-box! b 5)
     a)
#(#&5 #&5 #&5)
> (shared ([a (box b)]
           [b (vector (unbox a)   ; unbox after a is patched
                      (unbox c))] ; unbox before c is patched
           [c (box b)])
     b)
#0=#(#0# #<undefined>)
```

2.10 Conditionals: `if`, `cond`, `and`, and `or`

§4.7 "Conditionals" in §"**Guide**: PLT Scheme" introduces conditionals.

```
(if test-expr then-expr else-expr)
```

Evaluates *test-expr*. If it produces any value other than `#f`, then *then-expr* is evaluated, and its results are the result for the `if` form. Otherwise, *else-expr* is evaluated, and its results are the result for the `if` form. The *then-expr* and *else-expr* are in tail position with respect to the `if` form.

Examples:

```
> (if (positive? -5) (error "doesn't get here") 2)
```

```
2
> (if (positive? 5) 1 (error "doesn't get here"))
1
```

```
(cond cond-clause ...)

cond-clause = [test-expr then-expr ...+]
            | [else then-expr ...+]
            | [test-expr => proc-expr]
            | [test-expr]
```

§4.7.3 "Chaining Tests: cond" in §"**Guide**: PLT Scheme" introduces cond.

A *cond-clause* that starts with else must be the last *cond-clause*.

If no *cond-clause*s are present, the result is #<void>.

If only a [else *then-expr* ...+] is present, then the *then-expr*s are evaluated. The results from all but the last *then-expr* are ignored. The results of the last *then-expr*, which is in tail position with respect to the cond form, are the results for the whole cond form.

Otherwise, the first *test-expr* is evaluated. If it produces #f, then the result is the same as a cond form with the remaining *cond-clause*s, in tail position with respect to the original cond form. Otherwise, evaluation depends on the form of the *cond-clause*:

[*test-expr* *then-expr* ...+]

The *then-expr*s are evaluated in order, and the results from all but the last *then-expr* are ignored. The results of the last *then-expr*, which is in tail position with respect to the cond form, provides the result for the whole cond form.

[*test-expr* => *proc-expr*]

The *proc-expr* is evaluated, and it must produce a procedure that accepts on argument, otherwise the exn:fail:contract exception is raised. The procedure is applied to the result of *test-expr* in tail position with respect to the cond expression.

[*test-expr*]

The result of the *test-expr* is returned as the result of the cond form. The *test-expr* is not in tail position.

Examples:

```
> (cond)
> (cond
    [else 5])
5
> (cond
   [(positive? -5) (error "doesn't get here")]
   [(zero? -5) (error "doesn't get here, either")]
   [(positive? 5) 'here])
here
> (cond
   [(member 2 '(1 2 3)) => (lambda (l) (map - l))])
(-2 -3)
> (cond
   [(member 2 '(1 2 3))])
(2 3)
```

`else`

Recognized specially within forms like `cond`. An `else` form as an expression is a syntax error.

`=>`

Recognized specially within forms like `cond`. A `=>` form as an expression is a syntax error.

`(and expr ...)`

§4.7.2 "Combining Tests: `and` and `or`" in §"**Guide**: PLT Scheme" introduces `and`.

If no *`expr`*s are provided, then result is `#f`.

If a single *`expr`* is provided, then it is in tail position, so the results of the `and` expression are the results of the *`expr`*.

Otherwise, the first *`expr`* is evaluated. If it produces `#f`, the result of the `and` expression is `#f`. Otherwise, the result is the same as an `and` expression with the remaining *`expr`*s in tail position with respect to the original `and` form.

Examples:

```
> (and)
#t
> (and 1)
1
> (and (values 1 2))
1
```

```
2
> (and #f (error "doesn't get here"))
#f
> (and #t 5)
5
```

```
(or expr ...)
```

§4.7.2 "Combining Tests: and and or" in §"**Guide**: PLT Scheme" introduces or.

If no *expr*s are provided, then result is `#t`.

If a single *expr* is provided, then it is in tail position, so the results of the `and` expression are the results of the *expr*.

Otherwise, the first *expr* is evaluated. If it produces a value other than `#f`, that result is the result of the `or` expression. Otherwise, the result is the same as an `or` expression with the remaining *expr*s in tail position with respect to the original `or` form.

Examples:

```
> (or)
#f
> (or 1)
1
> (or (values 1 2))
1
2
> (or 5 (error "doesn't get here"))
5
> (or #f 5)
5
```

2.11 Dispatch: `case`

```
(case val-expr case-clause ...)

case-clause = [(datum ...) then-expr ...+]
            | [else then-expr ...+]
```

Evaluates *val-expr* and uses the result to select a *case-clause*. The selected clause is the first one with a *datum* whose `quote`d form is `eqv?` to the result of *val-expr*. If no such *datum* is present, the `else` *case-clause* is selected; if no `else` *case-clause* is present, either, then the result of the `case` form is `#<void>`.

For the selected *case-clause*, the results of the last *then-expr*, which is in tail position with respect to the case form, are the results for the whole case form.

A *case-clause* that starts with else must be the last *case-clause*.

Examples:

```
> (case (+ 7 5)
    [(1 2 3) 'small]
    [(10 11 12) 'big])
big
> (case (- 7 5)
    [(1 2 3) 'small]
    [(10 11 12) 'big])
small

(define (classify c)
  (case (char-general-category c)
    [(ll lu lt ln lo) "letter"]
    [(nd nl no) "number"]
    [else "other"]))

> (classify #\A)
"letter"
> (classify #\1)
"number"
> (classify #\!)
"other"
```

2.12 Definitions: define, define-syntax, ...

§4.5 "Definitions: define" in §"**Guide**: PLT Scheme" introduces definitions.

```
(define id expr)
(define (head args) body ...+)

head = id
     | (head args)

args = arg ...
     | arg ... . rest-id

 arg = arg-id
     | [arg-id default-expr]
     | keyword arg-id
     | keyword [arg-id default-expr]
```

The first form binds *id* to the result of *expr*, and the second form binds *id* to a procedure.

In the second case, the generation procedure is `(CVT (head args) body ...+)`, using the `CVT` meta-function defined as follows:

```
(CVT (id . kw-formals) . datum)   = (lambda kw-formals . datum)
(CVT (head . kw-formals) . datum) = (lambda kw-formals expr)
                                     if (CVT head . datum) = expr
```

At the top level, the top-level binding *id* is created after evaluating *expr*, if it does not exist already, and the top-level mapping of *id* (in the namespace linked with the compiled definition) is set to the binding at the same time.

Examples:

```
(define x 10)

> x
10

(define (f x)
  (+ x 1))

> (f 10)
11

(define ((f x) [y 20])
  (+ x y))

> ((f 10) 30)
40
> ((f 10))
30
```

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

Evaluates the *expr*, and binds the results to the *id*s, in order, if the number of results matches the number of *id*s; if *expr* produces a different number of results, the `exn:fail:contract` exception is raised.

At the top level, the top-level binding for each *id* is created after evaluating *expr*, if it does not exist already, and the top-level mapping of each *id* (in the namespace linked with the compiled definition) is set to the binding at the same time.

Examples:

```
(define-values () (values))

(define-values (x y z) (values 1 2 3))
```

```
> z
3
```

```
(define-syntax id expr)
(define-syntax (head args) body ...+)
```

The first form creates a transformer binding (see §1.2.3.5 "Transformer Bindings") of *id* with the value of *expr*, which is an expression at phase level 1 relative to the surrounding context. (See §1.2.1 "Identifiers and Binding" for information on phase levels.)

The second form is a shorthand the same as for `define`; it expands to a definition of the first form where the *expr* is a `lambda` form.

```
(define-syntaxes (id ...) expr)
```

Like `define-syntax`, but creates a transformer binding for each *id*. The *expr* should produce as many values as *id*s, and each value is bound to the corresponding *id*.

```
(define-for-syntax id expr)
(define-for-syntax (head args) body ...+)
```

Like `define`, except that the binding is at phase level 1 instead of phase level 0 relative to its context. The expression for the binding is also at phase level 1. (See §1.2.1 "Identifiers and Binding" for information on phase levels.)

```
(define-values-for-syntax (id ...) expr)
```

Like `define-for-syntax`, but *expr* must produce as many value as supplied *id*s, and all of the *id*s are bound (at phase level 1).

2.12.1 `require` Macros

```
(require scheme/require-syntax)
```

The bindings documented in this section are provided by the `scheme/require-syntax` library, not `scheme/base` or `scheme`.

```
(define-require-syntax id proc-expr)
```

Like `define-syntax`, but for a `require` sub-form. The *proc-expr* must produce a procedure that accepts and returns a syntax object representing a `require` sub-form.

This form expands to `define-syntax` with a use of `make-require-transformer`; see §11.4.1 "`require` Transformers" for more information.

2.12.2 `provide` Macros

```
(require scheme/provide-syntax)
```

The bindings documented in this section are provided by the `scheme/provide-syntax` library, not `scheme/base` or `scheme`.

```
(define-provide-syntax id proc-expr)
```

Like `define-syntax`, but for a `provide` sub-form. The *`proc-expr`* must produce a procedure that accepts and returns a syntax object representing a `provide` sub-form.

This form expands to `define-syntax` with a use of `make-provide-transformer`; see §11.4.2 "`provide` Transformers" for more information.

2.13 Sequencing: `begin`, `begin0`, and `begin-for-syntax`

§4.8 "Sequencing" in §"**Guide**: PLT Scheme" introduces `begin` and `begin0`.

```
(begin form ...)
(begin expr ...+)
```

The first form applies when `begin` appears at the top level, at module level, or in an internal-definition position (before any expression in the internal-definition sequence). In that case, the `begin` form is equivalent to splicing the *`form`*s into the enclosing context.

The second form applies for `begin` in an expression position. In that case, the *`expr`*s are evaluated in order, and the results are ignored for all but the last *`expr`*. The last *`expr`* is in tail position with respect to the `begin` form.

Examples:

```
> (begin
    (define x 10)
    x)
10
> (+ 1 (begin
         (printf "hi\n")
         2))
hi
3
> (let-values ([(x y) (begin
```

```
                              (values 1 2 3)
                              (values 1 2))])
    (list x y))
  (1 2)
```

```
(begin0 expr body ...+)
```

Evaluates the *expr*, then evaluates the *body*s, ignoring the *body* results. The results of the *expr* are the results of the `begin0` form, but the *expr* is in tail position only if no *body*s are present.

Examples:

```
> (begin0
    (values 1 2)
    (printf "hi\n"))
hi
1
2
```

```
(begin-for-syntax form ...)
```

Allowed only in a top-level context or module context. Each *form* is partially expanded (see §1.2.3.6 “Partial Expansion”) to determine one of the following classifications:

- `define` or `define-values` form: converted to a `define-for-syntax` form.
- `require` form: content is wrapped with `for-syntax`.
- expression form *expr*: converted to `(define-values () (begin expr (values)))`, which effectively evaluates the expression at expansion time and, in the case of a module context, preserves the expression for future visits of the module.

2.14 Guarded Evaluation: `when` and `unless`

§4.8.3 “Effects If...: `when` and `unless`” in §“**Guide**: PLT Scheme” introduces `when` and `unless`.

```
(when test-expr expr ...)
```

Evaluates the `text-expr`. If the result is any value other than `#f`, the *expr*s are evaluated, and the results are ignored. No *expr* is in tail position with respect to the `when` form.

Examples:

```
> (when (positive? -5)
    (display "hi"))
```

```
> (when (positive? 5)
    (display "hi")
    (display " there"))
hi there
```

(unless *test-expr* *expr* ...)

Equivalent to (when (not *test-expr*) *expr* ...).

Examples:

```
> (unless (positive? 5)
    (display "hi"))
> (unless (positive? -5)
    (display "hi")
    (display " there"))
hi there
```

2.15 Assignment: set! and set!-values

§4.9 "Assignment: set!" in §"**Guide**: PLT Scheme" introduces set!.

(set! *id* *expr*)

If *id* has a transformer binding to an assignment transformer, as produced by make-set!-transformer, then this form is expanded by calling the assignment transformer with the full expressions. If *id* has a transformer binding to a rename transformer as produced by make-rename-transformer, then this form is expanded by replacing *id* with the one provided to make-rename-transformer.

Otherwise, evaluates *expr* and installs the result into the location for *id*, which must be bound as a local variable or defined as a top-level variable or module-level variable. If *id* refers to an imported binding, a syntax error is reported. If *id* refers to a top-level variable that has not been defined, the exn:fail:contract exception is raised.

See also compile-allow-set!-undefined.

Examples:

```
(define x 12)

> (set! x (add1 x))
> x
13
> (let ([x 5])
    (set! x (add1 x))
    x)
```

```
6
> (set! i-am-not-defined 10)
```
set!: cannot set undefined identifier: i-am-not-defined

```
(set!-values (id ...) expr)
```

Assuming that all `id`s refer to variables, this form evaluates `expr`, which must produce as many values as supplied `id`s. The location of each `id` is filled wih to the corresponding value from `expr` in the same way as for `set!`.

Examples:

```
> (let ([a 1]
        [b 2])
    (set!-values (a b) (values b a))
    (list a b))
(2 1)
```

More generally, the `set!-values` form is expanded to

```
(let-values ([(tmp-id ...) expr])
  (set! id tmp-id) ...)
```

which triggers further expansion if any `id` has a transformer binding to an assignment transformer.

2.16 Iterations and Comprehensions: `for`, `for/list`, ...

§11 "Iterations and Comprehensions" in §"**Guide**: PLT Scheme" introduces iterations and comprehensions.

The `for` iteration forms are based on SRFI-42 [SRFI-42].

2.16.1 Iteration and Comprehension Forms

```
(for (for-clause ...) body ...+)

for-clause = [id seq-expr]
           | [(id ...) seq-expr]
           | #:when guard-expr
```

Iteratively evaluates `body`. The `for-clause`s introduce bindings whose scope includes `body` and that determine the number of times that `body` is evaluated.

In the simple case, each `for-clause` has one of its first two forms, where `[id seq-expr]` is a shorthand for `[(id ...) seq-expr]`. In this simple case, the `seq-expr`s are evaluated left-to-right, and each must produce a sequence value (see §3.14 "Sequences").

The `for` form iterates by drawing an element from each sequence; if any sequence is empty, then the iteration stops, and `#<void>` is the result of the `for` expression. Otherwise a location is created for each *`id`* to hold the values of each element; the sequence produced by a *`seq-expr`* must return as many values for each iteration as corresponding *`id`*s.

The *`id`*s are then bound in the *`body`*, which is evaluated, and whose results are ignored. Iteration continues with the next element in each sequence and with fresh locations for each *`id`*.

A `for` form with zero *`for-clause`*s is equivalent to a single *`for-clause`* that binds an unreferenced *`id`* to a sequence containing a single element. All of the *`id`*s must be distinct according to `bound-identifier=?`.

If any *`for-clause`* has the form `#:when` *`guard-expr`*, then only the preceding clauses (containing no `#:when`) determine iteration as above, and the *`body`* is effectively wrapped as

```
(when guard-expr
  (for (for-clause ...) body ...+))
```

using the remaining `for-clauses`.

Examples:

```
> (for ([i '(1 2 3)]
        [j "abc"]
        #:when (odd? i)
        [k #(#t #f)])
    (display (list i j k)))
(1 a #t)(1 a #f)(3 c #t)(3 c #f)
> (for ([(i j) #hash(("a" . 1) ("b" . 20))])
    (display (list i j)))
(a 1)(b 20)
> (for ()
    (display "here"))
here
> (for ([i '()])
    (error "doesn't get here"))
```

```
(for/list (for-clause ...) body ...+)
```

Iterates like `for`, but that the last expression in the *`body`*s must produce a single value, and the result of the `for/list` expression is a list of the results in order.

Examples:

```
> (for/list ([i '(1 2 3)]
             [j "abc"]
             #:when (odd? i)
             [k #(#t #f)])
    (list i j k))
((1 #\a #t) (1 #\a #f) (3 #\c #t) (3 #\c #f))
> (for/list () 'any)
(any)
> (for/list ([i '()])
    (error "doesn't get here"))
()
```

```
(for/hash (for-clause ...) body ...+)
(for/hasheq (for-clause ...) body ...+)
```

Like `for/list`, but the result is an immutable hash table; `for/hash` creates a table using `equal?` to distinguish keys, and `for/hasheq` produces a table using `eq?`. The last expression in the *body*s must return two values: a key and a value to extend the hash table accumulated by the iteration.

Examples:

```
> (for/hash ([i '(1 2 3)])
    (values i (number->string i)))
#hash((1 . "1") (2 . "2") (3 . "3"))
```

```
(for/and (for-clause ...) body ...+)
```

Iterates like `for`, but when last expression of *body* produces `#f`, then iteration terminates, and the result of the `for/and` expression is `#f`. If the *body* is never evaluated, then the result of the `for/and` expression is `#t`. Otherwise, the result is the (single) result from the last evaluation of *body*.

Examples:

```
> (for/and ([i '(1 2 3 "x")])
    (i . < . 3))
#f
> (for/and ([i '(1 2 3 4)])
    i)
4
> (for/and ([i '()])
    (error "doesn't get here"))
#t
```

```
(for/or (for-clause ...) body ...+)
```

Iterates like `for`, but when last expression of *body* produces a value other than `#f`, then iteration terminates, and the result of the `for/or` expression is the same (single) value. If the *body* is never evaluated, then the result of the `for/or` expression is `#f`. Otherwise, the result is `#f`.

Examples:

```
> (for/or ([i '(1 2 3 "x")])
    (i . < . 3))
#t
> (for/or ([i '(1 2 3 4)])
    i)
1
> (for/or ([i '()])
    (error "doesn't get here"))
#f
```

```
(for/lists (id ...) (for-clause ...) body ...+)
```

Similar to `for/list`, but the last *body* expression should produce as many values as given *id*s, and the result is as many lists as supplied *id*s. The *id*s are bound to the lists accumulated so far in the *for-clause*s and *body*s.

```
(for/first (for-clause ...) body ...+)
```

Iterates like `for`, but after *body* is evaluated the first time, then the iteration terminates, and the `for/first` result is the (single) result of *body*. If the *body* is never evaluated, then the result of the `for/first` expression is `#f`.

Examples:

```
> (for/first ([i '(1 2 3 "x")]
              #:when (even? i))
     (number->string i))
"2"
> (for/first ([i '()])
    (error "doesn't get here"))
#f
```

```
(for/last (for-clause ...) body ...+)
```

Iterates like `for`, but the `for/last` result is the (single) result of of the last evaluation of *body*. If the *body* is never evaluated, then the result of the `for/last` expression is `#f`.

Examples:

```
> (for/last ([i '(1 2 3 4 5)]
```

```
              #:when (even? i))
      (number->string i))
"4"
> (for/last ([i '()])
    (error "doesn't get here"))
#f
```

```
(for/fold ([accum-id init-expr] ...) (for-clause ...) . body)
```

Iterates like `for`. Before iteration starts, the *`init-expr`*s are evaluated to produce initial accumulator values. At the start of each out iteration, a location is generated for each *`accum-id`*, and the correspinding current accumulator value is placed into the location. The last expression in *`body`* must produce as many values as *`accum-id`*s, and those values become the current accumulator values. When iteration terminates, the results of the `fold/for` expression are the accumulator values.

Examples:

```
> (for/fold ([sum 0]
             [rev-roots null])
            ([i '(1 2 3 4)])
    (values (+ sum i) (cons (sqrt i) rev-roots)))
10
(2 1.7320508075688772 1.4142135623730951 1)
```

```
(for* (for-clause ...) body ...+)
```

Like `for`, but with an implicit `#:when #t` between each pair of `for-clauses`, so that all sequence iterations are nested.

Examples:

```
> (for* ([i '(1 2)]
         [j "ab"])
    (display (list i j)))
(1 a)(1 b)(2 a)(2 b)
```

```
(for*/list (for-clause ...) body ...+)
(for*/lists (id ...) (for-clause ...) body ...+)
(for*/hash (for-clause ...) body ...+)
(for*/hasheq (for-clause ...) body ...+)
(for*/and (for-clause ...) body ...+)
(for*/or (for-clause ...) body ...+)
(for*/first (for-clause ...) body ...+)
(for*/last (for-clause ...) body ...+)
(for*/fold ([accum-id init-expr] ...) (for-clause ...) body ...+)
```

Like `for/list`, etc., but with the implicit nesting of `for*`.

Examples:

```
> (for*/list ([i '(1 2)]
              [j "ab"])
    (list i j))
((1 #\a) (1 #\b) (2 #\a) (2 #\b))
```

2.16.2 Deriving New Iteration Forms

```
(for/fold/derived orig-datum
  ([accum-id init-expr] ...) (for-clause ...) body ...+)
```

Like `for/fold`, but the extra *`orig-datum`* is used as the source for all syntax errors.

```
(for*/fold/derived orig-datum
  ([accum-id init-expr] ...) (for-clause ...) body ...+)
```

Like `for*/fold`, but the extra *`orig-datum`* is used as the source for all syntax errors.

```
(define-sequence-syntax id
  expr-transform-expr
  clause-transform-expr)
```

Defines *`id`* as syntax. An `(`*`id . rest`*`)` form is treated specially when used to generate a sequence in a *`clause`* of `for` (or one of its variants). In that case, the procedure result of *`clause-transform-expr`* is called to transform the clause.

When *`id`* is used in any other expression position, the result of *`expr-transform-expr`* is used. If it is a procedure of zero arguments, then the result must be an identifier *`other-id`*, and any use of *`id`* is converted to a use of *`other-id`*. Otherwise,*`expr-transform-expr`* must produce a procedure (of one argument) that is used as a macro transformer.

When the *`clause-transform-expr`* transformer is used, it is given a *`clause`* as an argument, where the clause's form is normalized so that the left-hand side is a parenthesized sequence of identifiers. The right-hand side is of the form `(`*`id . rest`*`)`. The result can be either `#f`, to indicate that the forms should not be treated specially (perhaps because the number of bound identifiers is inconsistent with the `(`*`id . rest`*`)` form), or a new *`clause`* to to replace the given one. The new clause might use `:do-in`.

```
(:do-in ([(outer-id ...) outer-expr] ...)
        outer-check
```

```
        ([loop-id loop-expr] ...)
        pos-guard
        ([(inner-id ...) inner-expr] ...)
        pre-guard
        post-guard
        (loop-arg ...))
```

A form that can only be used as a *seq-expr* in a *clause* of `for` (or one of its variants).

Within a `for`, the pieces of the `:do-in` form are spliced into the iteration essentially as follows:

```
(let-values ([(outer-id ...) outer-expr] ...)
  outer-check
  (let loop ([loop-id loop-expr] ...)
    (if pos-guard
        (let-values ([(inner-id ...) inner-expr] ...)
          (if pre-guard
              (let body-bindings
                   (if post-guard
                       (loop loop-arg ...)
                       done-expr))
              done-expr))
         done-expr)))
```

where *body-bindings* and *done-expr* are from the context of the `:do-in` use. The identifiers bound by the `for` clause are typically part of the `([(inner-id ...) inner-expr] ...)` section.

The actual `loop` binding and call has additional loop arguments to support iterations in parallel with the `:do-in` form, and the other pieces are similarly accompanied by pieces form parallel iterations.

2.16.3 Do Loops

```
(do ([id init-expr step-expr-maybe] ...)
    (cont?-expr finish-expr ...)
  expr ...+)

step-expr-maybe =
                | step-expr
```

Iteratively evaluates the *expr*s for as long as `cont-expr?` returns `#t`.

To initialize the loop, the *init-expr*s are evaluated in order and bound to the corresponding *id*s. The *id*s are bound in all expressions within the form other than the *init-expr*s.

After he *id*s are bound, then *cont?-expr* is evaluated. If it produces a true value, then each *expr* is evaluated for its side-effect. The *id*s are then updated with the values of the *step-expr*s, where the default *step-expr* for *id* is just *id*. Iteration continues by evaluating *cont?-expr*.

When *cont?-expr* produces `#f`, then the *finish-expr*s are evaluated in order, and the last one is evaluated in tail position to produce the overall value for the `do` form. If no *finish-expr* is provided, the value of the `do` form is `#<void>`.

2.17 Continuation Marks: `with-continuation-mark`

```
(with-continuation-mark key-expr val-expr result-expr)
```

The *key-expr*, `mark-expr`, and *result-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 *result-expr* is evaluated; the continuation for evaluating *result-expr* is the continuation of the `with-continuation-mark` expression (so the result of the `resultbody-expr` is the result of the `with-continuation-mark` expression, and *result-expr* is in tail position for the `with-continuation-mark` expression).

§9.5 "Continuation Marks" provides more information on continuation marks.

2.18 Quasiquoting: `quasiquote`, `unquote`, and `unquote-splicing`

```
(quasiquote datum)
```

The same as `'datum` if *datum* does not include `,`*expr* or `,@`*expr*. An `,`*expr* expression escapes from the quote, however, and the result of the *expr* takes the place of the `,`*expr* form in the `quasiquote` result. An `,@`*expr* similarly escapes, but the *expr* must produce a list, and its elements are spliced as multiple values place of the `,@`*expr*, which must appear as the `car` or a quoted pair, as an element of a quoted vector, or as an element of a quoted prefab structure; in the case of a pair, if the `cdr` of the relevant quoted pair is empty, then *expr* need not produce a list, and its result is used directly in place of the quoted pair (in the same way that `append` accepts a non-list final argument). If `unquote` or `unquote-splicing` appears within `quasiquote` in any other way than as `,`*expr* or `,@`*expr*, a syntax error is reported.

Examples:

```
> (quasiquote (0 1 2))
(0 1 2)
> (quasiquote (0 (unquote (+ 1 2)) 4))
(0 3 4)
> (quasiquote (0 (unquote-splicing (list 1 2)) 4))
(0 1 2 4)
> (quasiquote (0 (unquote-splicing 1) 4))
unquote-splicing: expected argument of type <proper list>;
given 1
> (quasiquote (0 (unquote-splicing 1)))
(0 . 1)
```

A `quasiquote`, `unquote`, or `unquote-splicing` form is typically abbreviated with `` ` ``, `,`, or `,@`, respectively. See also §12.6.7 “Reading Quotes”.

Examples:

```
> `(0 1 2)
(0 1 2)
> `(1 ,(+ 1 2) 4)
(1 3 4)
> `#s(stuff 1 ,(+ 1 2) 4)
#s(stuff 1 3 4)
> `(1 ,@(list 1 2) 4)
(1 1 2 4)
> `#(1 ,@(list 1 2) 4)
#(1 1 2 4)
```

A `quasiquote` form within the original *datum* increments the level of quasiquotation: within the `quasiquote` form, each `unquote` or `unquote-splicing` is preserved, but a further nested `unquote` or `unquote-splicing` escapes. Multiple nestings of `quasiquote` require multiple nestings of `unquote` or `unquote-splicing` to escape.

Examples:

```
> `(1 `,(+ 1 ,(+ 2 3)) 4)
(1 `,(+ 1 5)  4)
> `(1 ```,,@,,@(list (+ 1 2)) 4)
(1 ```,,@,3    4)
```

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

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

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

`unquote`

See `quasiquote`, where `unquote` is recognized as an escape. An `unquote` form as an expression is a syntax error.

```
unquote-splicing
```

See `quasiquote`, where `unquote-splicing` is recognized as an escape. An `unquote-splicing` form as an expression is a syntax error.

2.19 Syntax Quoting: `quote-syntax`

```
(quote-syntax datum)
```

Produces a syntax object that preserves the lexical information and source-location information attached to *datum* at expansion time.

Examples:

```
> (syntax? (quote-syntax x))
#t
```

2.20 Modules: `module`, ...

§6.2.1 "The `module` Form" in §"**Guide**: PLT Scheme" introduces `module`.

```
(module id module-path form ...)
```

Declares a module. If the `current-module-declare-name` parameter is set, the parameter value is used for the module name, otherwise `(quote id)` is the name of the declared module.

The *module-path* must be as for `require`, and it supplies the initial bindings for the body *form*s. That is, it is treated like a `(require module-path)` prefix before the *form*s, except that the bindings introduced by *module-path* can be shadowed by definitions and `require`s in the module body *form*s.

If a single *form* is provided, then it is partially expanded in a module-begin context. If the expansion leads to `#%plain-module-begin`, then the body of the `#%plain-module-begin` is the body of the module. If partial expansion leads to any other primitive form, then the form is wrapped with `#%module-begin` using the lexical context of the module body; this identifier must be bound by the initial *module-path* import, and its expansion must produce a `#%plain-module-begin` to supply the module body. Finally, if multiple *form*s are provided, they are wrapped with `#%module-begin`, as in the case where a single *form* does not expand to `#%plain-module-begin`.

After such wrapping, if any, and before any expansion, an `'enclosing-module-name` property is attached to the `#%module-begin` syntax object (see §11.6 "Syntax Object Properties"); the property's value is a symbol corresponding to *`id`*.

Each *`form`* is partially expanded (see §1.2.3.6 "Partial Expansion") in a module context. Further action depends on the shape of the form:

- If it is a `begin` form, so the sub-forms are flattened out into the module's body and immediately processed in place of the `begin`.
- If it is a `define-syntaxes` or `define-values-for-syntax` form, then the right-hand side is evaluated (in phase 1), and the binding is immediately installed for further partial expansion within the module.
- If the form is a `require` form, bindings are introduced immediately, and the imported modules are instantiated or visited as appropriate.
- If the form is a `provide` form, then it is recorded for processing after the rest of the body.
- If the form is a `define-values` form, then the binding is installed immediately, but the right-hand expression is not expanded further.
- Similarly, if the form is an expression, it is not expanded further.

After all *`form`*s have been partially expanded this way, then the remaining expression forms (including those on the right-hand side of a definition) are expanded in an expression context.

The scope of all imported identifiers covers the entire module body, as does the scope of any identifier defined within the module body. The ordering of syntax definitions does not affect the scope of the syntax names; a transformer for `A` can produce expressions containing `B`, while the transformer for `B` produces expressions containing `A`, regardless of the order of declarations for `A` and `B`. However, a syntactic form that produces syntax definitions must be defined before it is used.

No identifier can be imported or defined more than once at any phase level. Every exported identifier must be imported or defined. No expression can refer to a top-level variable.

The evaluation of a `module` form does not evaluate the expressions in the body of the module. Evaluation merely declares a module, whose full name depends both on *`id`* and `(current-module-name-prefix)`.

The module body is executed only when the module is explicitly instantiated via `require` or `dynamic-require`. On invocation, expressions and definitions are evaluated in order as they appear within the module; accessing a module-level variable before it is defined signals a run-time error, just like accessing an undefined global variable.

See also §1.1.10 "Modules and Module-Level Variables" and §1.2.3.8 "Module Phases".

```
(#%module-begin form ...)
```

Legal only in a module begin context, and handled by the `module` form.

The pre-defined `#%module-begin` form wraps every top-level expression to print non-`#<void>` results using `current-print`.

```
(#%plain-module-begin form ...)
```

Legal only in a module begin context, and handled by the `module` form.

2.21 Importing and Exporting: `require` and `provide`

§6.4 "Imports: `require`" in §"**Guide**: PLT Scheme" introduces `require`.

```
(require require-spec ...)
```

```
 require-spec = module-path
              | (only-in require-spec id-maybe-renamed ...)
              | (except-in require-spec id ...)
              | (prefix-in prefix-id require-spec)
              | (rename-in require-spec [orig-id bind-id] ...)
              | (combine-in require-spec ...)
              | (only-meta-in phase-level require-spec ...)
              | (for-syntax require-spec ...)
              | (for-template require-spec ...)
              | (for-label require-spec ...)
              | (for-meta phase-level require-spec ...)
              | derived-require-spec

  module-path = (quote id)
              | rel-string
              | (lib rel-string ...+)
              | id
              | (file string)
              | (planet id)
              | (planet string)
              | (planet rel-string
                        (user-string pkg-string vers ...)
                        rel-string ...)

id-maybe-renamed = id
              | [orig-id bind-id]

  phase-level = exact-integer
              | #f

         vers = nat
              | (nat nat)
              | (= nat)
              | (+ nat)
              | (- nat)
```

In a top-level context, `require` instantiates modules (see §1.1.10 "Modules and Module-Level Variables"). In a module context, `require` visits modules (see §1.2.3.8 "Module Phases"). In both contexts, `require` introduces bindings into a namespace or a module (see §1.2.3.4 "Introducing Bindings"). A `require` form in a expression context or internal-definition context is a syntax error.

A *require-spec* designates a particular set of identifiers to be bound in the importing context. Each identifier is mapped to a particular export of a particular module; the identifier to bind may be different from the symbolic name of the originally exported identifier. Each identifier also binds at a particular phase level.

The syntax of `require-spec` can be extended via `define-require-syntax`, but the predefined forms are as follows.

`module-path`

Imports all exported bindings from the named module, using the export identifiers as the local identifiers. (See below for information on `module-path`.) The lexical context of the `module-path` form determines the context of the introduced identifiers.

`(only-in require-spec id-maybe-renamed ...)`

Like `require-spec`, but constrained to those exports for which the identifiers to bind match `id-maybe-renamed`: as `id` or as `orig-id` in `[orig-id bind-id]`. If the `id` of `orig-id` of any `id-maybe-renamed` is not in the set that `require-spec` describes, a syntax error is reported.

`(except-in require-spec id ...)`

Like `require-spec`, but omitting those imports for which `id`s are the identifiers to bind; if any `id` is not in the set that `require-spec` describes, a syntax error is reported.

`(prefix-in prefix-id require-spec)`

Like `require-spec`, but adjusting each identifier to be bound by prefixing it with `prefix-id`. The lexical context of the `prefix-id` is ignored, and instead preserved from the identifiers before prefixing.

`(rename-in require-spec [orig-id bind-id] ...)`

Like `require-spec`, but replacing the identifier to bind `orig-id` with `bind-id`; if any `orig-id` is not in the set that `require-spec` describes, a syntax error is reported.

`(combine-in require-spec ...)`

The union of the `require-spec`s.

`(only-meta-in phase-level require-spec ...)`

Like the combination of *require-spec*s, but removing any binding that is not for *phase-level*, where `#f` for *phase-level* corresponds to the label phase level.

`(for-meta phase-level require-spec ...)`

Like the combination of *require-spec*s, but constrained each binding specified by each *require-spec* is shifted by *phase-level*. The label phase level corresponds to `#f`, and a shifting combination that involves `#f` produces `#f`.

`(for-syntax require-spec ...)`

Same as `(for-meta 1 require-spec ...)`.

`(for-template require-spec ...)`

Same as `(for-meta -1 require-spec ...)`.

`(for-label require-spec ...)`

Same as `(for-meta #f require-spec ...)`.

derived-require-spec

See `define-require-syntax` for information on expanding the set of *require-spec* forms.

§6.3 "Module Paths" in §"**Guide**: PLT Scheme" introduces module paths.

A *module-path* identifies a module, either through a concrete name in the form of an identifier, or through an indirect name that can trigger automatic loading of the module declaration. Except for the *id* case below, the actual resolution is up to the current module name resolver (see `current-module-name-resolver`), and the description below corresponds to the default module name resolver.

`(quote id)`

Refers to a module previously declared interactively with the name *id*.

`rel-string`

A path relative to the containing source (as determined by `current-load-relative-directory` or `current-directory`). Regardless of the current platform, *`rel-string`* is always parsed as a Unix-format relative path: `/` is the path delimiter (multiple adjacent `/`s are treated as a single delimiter), `..` accesses the parent directory, and `.` accesses the current directory. The path cannot be empty or contain a leading or trailing slash, path elements before than the last one cannot include a file suffix, and the only allowed characters are ASCII letters, ASCII digits, `-`, `+`, `_`, `.`, and `/`.

`(lib rel-string ...+)`

A path to a module installed into a collection (see §16.2 "Libraries and Collections"). The *`rel-string`*s in `lib` are constrained similar to the plain *`rel-string`* case, with the additional constraint that a *`rel-string`* cannot contain `.` or `..` directory indicators.

The specific interpretation of the path depends on the number and shape of the *`rel-string`*s:

- If a single *`rel-string`* is provided, and if it consists of a single element (i.e., no `/`) with no file suffix (i.e, no `.`), then *`rel-string`* names a collection, and `"main.ss"` is the library file name.
- If a single *`rel-string`* is provided, and if it consists of multiple `/`-separated elements, then each element up to the last names a collection, subcollection, etc., and the last element names a file. If the last element has no file suffix, `".ss"` is added.
- If a single *`rel-string`* is provided, and if it consists of a single element *with* a file suffix (i.e, no `.`), then *`rel-string`* names a file within the `"mzlib"` collection. (This convention is for compatibility with older version of PLT Scheme.)
- Otherwise, when multiple *`rel-string`*s are provided, the first *`rel-string`* is effectively moved after the others, and all *`rel-string`*s are appended with `/` separators. The resulting path names a collection, then subcollection, etc., ending with a file name. No suffix is added automatically. (This convention is for compatibility with older version of PLT Scheme.)

`id`

A shorthand for a `lib` form with a single *`rel-string`* whose characters are the same as in the symbolic form of *`id`*. In addition to the constraints of a `lib` *`rel-string`*, *`id`* must not contain `.`.

```
(file string)
```

Similar to the plain `rel-string` case, but `string` is a path—possibly absolute—using the current platform's path conventions and `expand-user-path`.

```
(planet id)
(planet string)
(planet rel-string (user-string pkg-string vers ...)
        rel-string ...)
```

Specifies a library available via the PLaneT server.

The first form is a shorthand for the last one, where the `id`'s character sequence must match the following ⟨*spec*⟩ grammar:

⟨*spec*⟩	::=	⟨*owner*⟩ `/` ⟨*pkg*⟩ ⟨*lib*⟩
⟨*owner*⟩	::=	⟨*elem*⟩
⟨*pkg*⟩	::=	⟨*elem*⟩ | ⟨*elem*⟩ `:` ⟨*version*⟩
⟨*version*⟩	::=	⟨*int*⟩ | ⟨*int*⟩ `:` ⟨*minor*⟩
⟨*minor*⟩	::=	⟨*int*⟩ | `<=` ⟨*int*⟩ | `>=` ⟨*int*⟩ | `=` ⟨*int*⟩
	|	⟨*int*⟩ `-` ⟨*int*⟩
⟨*lib*⟩	::=	⟨*empty*⟩ | `/` ⟨*path*⟩
⟨*path*⟩	::=	⟨*elem*⟩ | ⟨*elem*⟩ `/` ⟨*path*⟩

and where an ⟨*elem*⟩ is a non-empty sequence of characters that are ASCII letters, ASCII digits, `-`, `+`, or `_`, and an ⟨*int*⟩ is a non-empty sequence of ASCII digits. As this shorthand is expended, a `".plt"` extension is added to ⟨*pkg*⟩, and a `".ss"` extension is added to `"path"`; if no ⟨*path*⟩ is included, `"main.ss"` is used in the expansion.

A `(planet string)` form is like a `(planet id)` form with the identifier converted to a string, except that the `string` can optionally end with a file extension for a ⟨*path*⟩.

In the more general last form of a `planet` module path, the `rel-string`s are similar to the `lib` form, except that the `(user-string pkg-string vers ...)` names a PLaneT-based package instead of a collection.

No identifier can be bound multiple times in a given phase level by an import, unless all of the bindings refer to the same original definition in the same module. In a module context, an identifier can be either imported or defined for a given phase level, but not both.

§6.5 "Exports: `provide`" in §"**Guide**: PLT Scheme" introduces `provide`.

```
(provide provide-spec ...)
```

```
provide-spec = id
             | (all-defined-out)
             | (all-from-out module-path ...)
             | (rename-out [orig-id export-id] ...)
             | (except-out provide-spec provide-spec ...)
             | (prefix-out prefix-id provide-spec)
             | (struct-out id)
             | (combine-out provide-spec ...)
             | (protect-out provide-spec ...)
             | (for-meta phase-level provide-spec ...)
             | (for-syntax provide-spec ...)
             | (for-template provide-spec ...)
             | (for-label provide-spec ...)
             | derived-provide-spec

 phase-level = exact-integer
             | #f
```

Declares exports from a module. A `provide` form must appear in a module context or a module-begin context.

A *`provide-spec`* indicates one or more bindings to provide. For each exported binding, the external name is a symbol that can be different from the symbolic form of the identifier that is bound within the module. Also, each export is drawn from a particular phase level and exported at the same phase level.

The syntax of *`provide-spec`* can be extended via `define-provide-syntax`, but the pre-defined forms are as follows.

`id`

Exports *`id`*, which must be bound within the module (i.e., either defined or imported) at the relevant phase level. The symbolic form of *`id`* is used as the external name, and the symbolic form of the defined or imported identifier must match (otherwise, the external name could be ambiguous).

`(all-defined-out)`

Exports all identifiers that are defined at phase level 0 or phase level 1 within the exporting module, and that have the same lexical context as the `(all-defined-out)` form. The external name for each identifier is the symbolic form of the identifier. Only identifiers accessible from the lexical context of the `(all-defined-out)` form are included; that is, macro-introduced imports are not re-exported, unless the `(all-defined-out)` form was introduced at the same time.

`(all-from-out module-path ...)`

Exports all identifiers that are imported into the exporting module using a `require-spec` built on each *`module-path`* (see §2.21 "Importing and Exporting: `require` and `provide`") with no phase-level shift. The symbolic name for export is derived from the name that is bound within the module, as opposed to the symbolic name of the export from each *`module-path`*. Only identifiers accessible from the lexical context of the *`module-path`* are included; that is, macro-introduced imports are not re-exported, unless the *`module-path`* was introduced at the same time.

`(rename-out [orig-id export-id] ...)`

Exports each *`orig-id`*, which must be bound within the module at phase level 0. The symbolic name for each export is *`export-id`* instead `orig-d`.

`(except-out provide-spec provide-spec ...)`

Like the first *`provide-spec`*, but omitting the bindings listed in each subsequent *`provide-spec`*. If one of the latter bindings is not included in the initial *`provide-spec`*, a syntax error is reported. The symbolic export name information in the latter *`provide-spec`*s is ignored; only the bindings are used.

`(prefix-out prefix-id provide-spec)`

Like *`provide-spec`*, but with each symbolic export name from *`provide-spec`* prefixed with *`prefix-id`*.

`(struct-out id)`

Exports the bindings associated with a structure type *`id`*. Typically, *`id`* is bound with `(define-struct id ....)` or `(define-struct (id super-id) ....)`; more generally, *`id`* must have a transformer binding of structure-type information at phase level 0; see §4.6 "Structure Type Transformer Binding". Furthermore, for each identifier mentioned in the structure-type information, the enclosing module must define or import one identifier that is `free-`

`identifier=?`. If the structure-type information includes a super-type identifier, and if the identifier has a transformer binding of structure-type information, the accessor and mutator bindings of the super-type are *not* included by `struct-out` for export.

`(combine-out provide-spec ...)`

The union of the *`provide-spec`*s.

`(protect-out provide-spec ...)`

Like the union of the *`provide-spec`*s, except that the exports are protected; see §13.9 "Code Inspectors". The *`provide-spec`* must specify only bindings that are defined within the exporting module.

`(for-meta phase-level provide-spec ...)`

Like the union of the *`provide-spec`*s, but adjusted to apply to phase level specified by *`phase-level`* (where `#f` corresponds to the label phase level). In particular, an *`id`* or `rename-out` form as a *`provide-spec`* refers to a binding at *`phase-level`*, an `all-define-out` exports only *`phase-level`* definitions, and an `all-from-out` exports bindings imported with a shift by *`phase-level`*.

`(for-syntax provide-spec ...)`

Same as `(for-meta 1 provide-spec ...)`.

`(for-template provide-spec ...)`

Same as `(for-meta -1 provide-spec ...)`.

`(for-label provide-spec ...)`

Same as `(for-meta #f provide-spec ...)`.

derived-provide-spec

See `define-provide-syntax` for information on expanding the set of *provide-spec* forms.

Each export specified within a module must have a distinct symbolic export name, though the same binding can be specified with the multiple symbolic names.

```
(for-meta require-spec ...)
```

See `require` and `provide`.

```
(for-syntax require-spec ...)
```

See `require` and `provide`.

```
(for-template require-spec ...)
```

See `require` and `provide`.

```
(for-label require-spec ...)
```

See `require` and `provide`.

```
(#%require raw-require-spec ...)
```

```
raw-require-spec = phaseless-spec
                 | (for-meta phase-level phaseless-spec ...)
                 | (for-syntax phaseless-spec ...)
                 | (for-template phaseless-spec ...)
                 | (for-label phaseless-spec ...)
                 | (just-meta phase-level raw-require-spec ...)

     phase-level = exact-integer
                 | #f

  phaseless-spec = raw-module-path
                 | (only rw-module-path id ...)
                 | (prefix prefix-id raw-module-path)
                 | (all-except raw-module-path id ...)
                 | (prefix-all-except prefix-id
                                      raw-module-path id ...)
                 | (rename raw-module-path local-id exported-id)

 raw-module-path = (quote id)
                 | rel-string
                 | (lib rel-string ...)
                 | id
                 | (file string)
                 | (planet rel-string
                           (user-string pkg-string vers ...))
```

The primitive import form, to which `require` expands. A *raw-require-spec* is similar to a *require-spec* in a `require` form, except that the syntax is more constrained, not composable, and not extensible. Also, sub-form names like `for-syntax` and `lib` are recognized symbolically, instead of via bindings. Although not formalized in the grammar above, a `just-meta` form cannot appear within a `just-meta` form.

Each *raw-require-spec* corresponds to the obvious *require-spec*, but the `rename` sub-form has the identifiers in reverse order compared to `rename-in`.

For most *raw-require-spec*s, the lexical context of the *raw-require-spec* determines the context of introduced identifiers. The exception is the `rename` sub-form, where the lexical context of the *local-id* is preserved.

```
(#%provide raw-provide-spec ...)
```

```
raw-provide-spec = phaseless-spec
                 | (for-meta phase-level phaseless-spec)
                 | (for-syntax phaseless-spec)
                 | (for-label phaseless-spec)
                 | (protect raw-provide-spec)

     phase-level = exact-integer
                 | #f

  phaseless-spec = id
                 | (rename local-id export-id)
                 | (struct struct-id (field-id ...))
                 | (all-from raw-module-path)
                 | (all-from-except raw-module-path id ...)
                 | (all-defined)
                 | (all-defined-except id ...)
                 | (prefix-all-defined prefix-id)
                 | (prefix-all-defined-except prefix-id id ...)
                 | (protect phaseless-spec ...)
                 | (expand (id . datum))
```

The primitive export form, to which `provide` expands. A *`raw-module-path`* is as for `#%require`. A `protect` sub-form cannot appear within a `protect` sub-form.

Like `#%require`, the sub-form keywords for `#%provide` are recognized symbolically, and nearly every *`raw-provide-spec`* has an obvious equivalent *`provide-spec`* via `provide`, with the exception of the `struct` and `expand` sub-forms.

A `(struct struct-id (field-id ...))` sub-form expands to *`struct-id`*, `make-`*`struct-id`*, `struct:`*`struct-id`*, *`struct-id`*`?`, *`struct-id`*`-`*`field-id`* for each *`field-id`*, and `set-`*`struct-id`*`-`*`field-id`*`!` for each *`field-id`*. The lexical context of the *`struct-id`* is used for all generated identifiers.

Unlike `#%require`, the `#%provide` form is macro-extensible via an explicit `expand` sub-form; the `(id . datum)` part is locally expanded as an expression (even though it is not actually an expression), stopping when a `begin` form is produced; if the expansion result is `(begin raw-provide-spec ...)`, it is spliced in place of the `expand` form, otherwise a syntax error is reported. The `expand` sub-form is not normally used directly; it provides a hook for implementing `provide` and provide transformers.

The `all-from` and `all-from-except` forms re-export only identifiers that are accessible in lexical context of the `all-from` or `all-from-except` form itself. That is, macro-introduced imports are not re-exported, unless the `all-from` or `all-from-except` form was introduced at the same time. Similarly, `all-defined` and its variants export only definitions accessible from the lexical context of the *`phaseless-spec`* form.

2.21.1 Additional require Forms

```
(require scheme/require)
```

The bindings documented in this section are provided by the scheme/require library, not scheme/base or scheme.

The following forms support more complex selection and manipulation of sets of imported identifiers. Note that a require form is expanded before it is used, which means that requiring the library itself should be a separate form. For example, use

```
(require scheme/require)
(require (matching-identifiers-in #rx"foo" "foo.ss"))
```

instead of

```
(require scheme/require
         (matching-identifiers-in #rx"foo" "foo.ss"))
```

```
(matching-identifiers-in regexp require-spec)
```

Like *require-spec*, but including only imports whose names match *regexp*. The *regexp* must be a literal regular expression (see §3.7 "Regular Expressions").

```
(subtract-in require-spec subtracted-spec ...)
```

Like *require-spec*, but omitting those imports that would be imported by one of the *subtracted-spec*s.

2.21.2 Additional provide Forms

```
(require scheme/provide)
```

The bindings documented in this section are provided by the scheme/provide library, not scheme/base or scheme.

```
(matching-identifiers-out regexp provide-spec)
```

Like *provide-spec*, but omitting the export of each binding with an external name that matches *regexp*. The *regexp* must be a literal regular expression (see §3.7 "Regular Expressions").

2.22 Interaction Wrapper: #%top-interaction

```
(#%top-interaction . form)
```

Expands to simply *form*. The #%top-interaction form is similar to #%app and #%module-begin, in that it provides a hook to control interactive evaluation through load (more precisely, the default load handler) or read-eval-print-loop.

3 Datatypes

Each pre-defined datatype comes with a set of procedures for manipulating instances of the datatype.

3.1 Booleans and Equality

True and false are represented by the values `#t` and `#f`, respectively, though operations that depend a boolean value typically treat anything other than `#f` as true.

See also: `and`, `or`, `andmap`, `ormap`.

```
(boolean? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is `#t` or `#f`, `#f` otherwise.

```
(not v) → boolean?
  v : any/c
```

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

```
(equal? v1 v2) → boolean?
  v1 : any/c
  v2 : any/c
```

Two values are `equal?` if and only if they are `eqv?`, unless otherwise specified for a particular datatype.

Datatypes with further specification of `equal?` include strings, byte strings, numbers, pairs, mutable pairs, vectors, hash tables, and inspectable structures. In the last five cases, equality is recursively defined; if both *v1* and *v2* contain reference cycles, they are equal when the infinite unfoldings of the values would be equal. See also `prop:equal+hash`.

```
(eqv? v1 v2) → boolean?
  v1 : any/c
  v2 : any/c
```

Two values are `eqv?` if and only if they are `eq?`, unless otherwise specified for a particular datatype.

The number and character datatypes are the only ones for which `eqv?` differs from `eq?`.

```
(eq? v1 v2) → boolean?
  v1 : any/c
  v2 : any/c
```

Return `#t` if *v1* and *v2* refer to the same object, `#f` otherwise. See also §1.1.6 "Object Identity and Comparisons".

```
(immutable? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is an immutable string, byte string, vector, hash table, or box, `#f` otherwise.

```
prop:equal+hash : struct-type-property?
```

A structure type property (see §4.3 "Structure Type Properties") that supplies an equality predicate and hashing functions for a structure type. The property value must be a list of three procedures:

- `equal-proc : (-> any/c any/c (-> any/c any/c boolean?) any/c)` — tests whether the first two arguments are equal, where both values are instances of the structure type to which the property is associated (or a subtype of the structure type).

  The third argument is an `equal?` predicate to use for recursive equality checks; use the given predicate instead of `equal?` to ensure that data cycles are handled properly.

  The *equal-proc* is called for a pair of structures only when they are not `eq?`, and only when they both have a `prop:equal+hash` value inherited from the same structure type. With this strategy, the order in which `equal?` receives two structures does not matter. It also means that, by default, a structure sub-type inherits the equality predicate of its parent, if any.

- `hash-proc : (-> any/c (-> any/c exact-integer?) exact-integer?)` — computes a hash code for the given structure, like `equal-hash-code`. The first argument is an instance of the structure type (or one of its subtypes) to which the property is associated.

  The second argument is a `equal-hash-code`-like procedure to use for recursive hash-code computation; use the given procedure instead of `equal-hash-code` to ensure that data cycles are handled properly.

- `hash2-proc : (-> any/c (-> any/c exact-integer?) exact-integer?)` — computes a secondary hash code for the given structure. This procedure is like *hash-proc*, but analogous to `equal-secondary-hash-code`.

Take care to ensure that *hash-proc* and *hash2-proc* are consistent with *equal-proc*. Specifically, *hash-proc* and *hash2-proc* should produce the same value for any two structures for which *equal-proc* produces a true value.

When a structure type has no `prop:equal+hash` property, then transparent structures (i.e., structures with an inspector that is controlled by the current inspector) are `equal?` when they are instances of the same structure type (not counting sub-types), and when they have `equal?` field values. For transparent structures, `equal-hash-code` and `equal-secondary-hash-code` derive hash code using the field values. For opaque structure types, `equal?` is the same as `eq?`, and `equal-hash-code` and `equal-secondary-hash-code` results are based only on `eq-hash-code`.

3.1.1 Boolean Synonyms

```
(require scheme/bool)
```

The bindings documented in this section are provided by the `scheme/bool` and `scheme` libraries, but not `scheme/base`.

```
true : boolean?
```

A synonym for `#t`.

```
false : boolean?
```

A synonym for `#f`.

```
(symbol=? a b) → boolean?
  a : symbol?
  b : symbol?
```

Returns `(equal? a b)`.

```
(boolean=? a b) → boolean?
  a : boolean?
  b : boolean?
```

Returns `(equal? a b)`.

3.2 Numbers

§3.2 "Numbers" in §"**Guide**: PLT Scheme" introduces numbers.

All numbers are *complex numbers*. Some of them are *real numbers*, and all of the real numbers that can be represented are also *rational numbers*, except for `+inf.0` (positive infinity), `-inf.0` (negative infinity), and `+nan.0` (not-a-number). Among the rational numbers, some are *integers*, because `round` applied to the number produces the same number.

Orthogonal to those categories, each number is also either an *exact number* or an *inexact number*. Unless otherwise specified, computations that involve an inexact number produce inexact results. Certain operations on inexact numbers, however, produce an exact number, such as multiplying an inexact number with an exact `0`. Some operations, which can produce an irrational number for rational arguments (e.g., `sqrt`), may produce inexact results even for exact arguments.

In the case of complex numbers, 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 exact zero imaginary part is a real number.

Inexact real numbers are implemented as either single- or double-precision IEEE floating-point numbers—the latter by default, and the former only when support for 32-bit inexact numbers is specifically enabled when the run-time system is built, and when computation starts with numerical constants specified as single-precision numbers.

The precision and size of exact numbers is limited only by available memory (and the precision of operations that can produce irrational numbers). In particular, adding, multiplying, subtracting, and dividing exact numbers always produces an exact result.

Inexact numbers can be coerced to exact form, except for the inexact numbers `+inf.0`, `-inf.0`, and `+nan.0`, which have no exact form. Dividing a number by exact zero raises an exception; dividing a non-zero number other than `+nan.0` by an inexact zero returns `+inf.0` or `-inf.0`, depending on the sign of the dividend. The `+nan.0` value is not `=` to itself, but `+nan.0` is `eqv?` to itself. Conversely, `(= 0.0 -0.0)` is `#t`, but `(eqv? 0.0 -0.0)` is `#f`. The datum `-nan.0` refers to the same constant as `+nan.0`.

Calculations with infinites produce results consistent with IEEE double-precision floating point where IEEE specifies the result; in cases where IEEE provides no specification (e.g., `(angle +inf.0+inf.0i)`), the result corresponds to the limit approaching infinity, or `+nan.0` if no such limit exists.

A *fixnum* is an exact integer whose two's complement representation fit into 31 bits on a 32-bit platform or 63 bits on a 64-bit platform. Two fixnums that are `=` are also the same according to `eq?`. Otherwise, the result of `eq?` applied to two numbers is undefined.

Two numbers are `eqv?` when they are both inexact or both exact, and when they are `=` (except for `+nan.0`, as noted above). Two numbers are `equal?` when they are `eqv?`.

3.2.1 Number Types

```
(number? v) → boolean?
  v : any/c
```

Returns `#t` if `v` is a number, `#f` otherwise.

Examples:

```
> (number? 1)
#t
> (number? 2+3i)
#t
> (number? "hello")
#f
```

```
(complex? v) → boolean?
  v : any/c
```

Returns `(number? v)`, because all numbers are complex numbers.

```
(real? v) → boolean?
  v : any/c
```

Returns `#t` if `v` is a real number, `#f` otherwise.

Examples:

```
> (real? 1)
#t
> (real? +inf.0)
#t
> (real? 2+3i)
#f
> (real? 2.0+0.0i)
#f
> (real? "hello")
#f
```

```
(rational? v) → boolean?
  v : any/c
```

Returns `#t` if `v` is a rational number, `#f` otherwise.

Examples:

```
> (rational? 1)
#t
> (rational? +inf.0)
#f
> (real? "hello")
#f
```

```
(integer? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a number that is an integer, `#f` otherwise.

Examples:

```
> (integer? 1)
#t
> (integer? 2.3)
#f
> (integer? 4.0)
#t
> (integer? +inf.0)
#f
> (integer? 2+3i)
#f
> (integer? "hello")
#f
```

```
(exact-integer? v) → boolean?
  v : any/c
```

Returns `(and (integer? v) (exact? v))`.

Examples:

```
> (exact-integer? 1)
#t
> (exact-integer? 4.0)
#f
```

```
(exact-nonnegative-integer? v) → boolean?
  v : any/c
```

Returns `(and (exact-integer? v) (not (negative? v)))`.

Examples:

```
> (exact-nonnegative-integer? 0)
```

```
#t
> (exact-nonnegative-integer? -1)
#f
```

```
(exact-positive-integer? v) → boolean?
  v : any/c
```

Returns `(and (exact-integer? v) (positive? v))`.

Examples:

```
> (exact-positive-integer? 1)
#t
> (exact-positive-integer? 0)
#f
```

```
(inexact-real? v) → boolean?
  v : any/c
```

Returns `(and (real? v) (inexact? v))`.

```
(fixnum? v) → boolean?
  v : any/c
```

Return `#t` if *v* is a fixnum, `#f` otherwise.

```
(zero? z) → boolean?
  z : number?
```

Returns `(= 0 z)`.

Examples:

```
> (zero? 0)
#t
> (zero? -0.0)
#t
```

```
(positive? x) → boolean?
  x : real?
```

Returns `(> x 0)`.

Examples:

```
> (positive? 10)
```

```
#t
> (positive? -10)
#f
> (positive? 0.0)
#f
```

```
(negative? x) → boolean?
  x : real?
```

Returns `(< x 0)`.

Examples:
```
> (negative? 10)
#f
> (negative? -10)
#t
> (negative? -0.0)
#f
```

```
(even? n) → boolean?
  n : integer?
```

Returns `(zero? (modulo n 2))`.

Examples:
```
> (even? 10.0)
#t
> (even? 11)
#f
> (even? +inf.0)
even?: expects argument of type <integer>; given +inf.0
```

```
(odd? n) → boolean?
  n : integer?
```

Returns `(not (even? n))`.

Examples:
```
> (odd? 10.0)
#f
> (odd? 11)
#t
> (odd? +inf.0)
odd?: expects argument of type <integer>; given +inf.0
```

```
(exact? z) → boolean?
  z : number?
```

Returns `#t` if *z* is an exact number, `#f` otherwise.

Examples:

```
> (exact? 1)
#t
> (exact? 1.0)
#f
```

```
(inexact? z) → boolean?
  z : number?
```

Returns `#t` if *z* is an inexact number, `#f` otherwise.

Examples:

```
> (inexact? 1)
#f
> (inexact? 1.0)
#t
```

```
(inexact->exact z) → exact?
  z : number?
```

Coerces *z* to an exact number. If *z* is already exact, it is returned. If *z* is `+inf.0`, `-inf.0`, or `+nan.0`, then the `exn:fail:contract` exception is raised.

Examples:

```
> (inexact->exact 1)
1
> (inexact->exact 1.0)
1
```

```
(exact->inexact z) → inexact?
  z : number?
```

Coerces *z* to an inexact number. If *z* is already inexact, it is returned.

Examples:

```
> (exact->inexact 1)
1.0
> (exact->inexact 1.0)
```

```
1.0
```

3.2.2 Arithmetic

```
(+ z ...) → number?
  z : number?
```

Returns the sum of the *z*s, adding pairwise from left to right. If no arguments are provided, the result is `0`.

Examples:

```
> (+ 1 2)
3
> (+ 1.0 2+3i 5)
8.0+3.0i
> (+)
0
```

```
(- z) → number?
  z : number?
(- z w ...+) → number?
  z : number?
  w : number?
```

When no *w*s are supplied, returns `(- 0 z)`. Otherwise, returns the subtraction of the *w*s from *z* working pairwise from left to right.

Examples:

```
> (- 5 3.0)
2.0
> (- 1)
-1
> (- 2+7i 1 3)
-2+7i
```

```
(* z ...) → number?
  z : number?
```

Returns the product of the *z*s, multiplying pairwise from left to right. If no arguments are provided, the result is `1`.

Examples:

```
> (* 2 3)
6
> (* 8.0 9)
72.0
> (* 1+2i 3+4i)
-5+10i
```

`(/ z) → number?`
  `z : number?`
`(/ z w ...+) → number?`
  `z : number?`
  `w : number?`

When no `w`s are supplied, returns `(/ 1 z)`. Otherwise, returns the division `z` by the var[w]s working pairwise from left to right.

Examples:

```
> (/ 3 4)
3/4
> (/ 81 3 3)
9
> (/ 10.0)
0.1
> (/ 1+2i 3+4i)
11/25+2/25i
```

`(quotient n m) → integer?`
  `n : integer?`
  `m : integer?`

Returns `(truncate (/ n m))`.

Examples:

```
> (quotient 10 3)
3
> (quotient -10.0 3)
-3.0
> (quotient +inf.0 3)
quotient: expects type <integer> as 1st argument, given:
+inf.0; other arguments were: 3
```

`(remainder n m) → integer?`
  `n : integer?`
  `m : integer?`

Returns `q` with the same sign as *n* such that

- `(abs q)` is between `0` (inclusive) and `(abs m)` (exclusive), and
- `(+ q (* m (quotient n m)))` equals *n*.

Examples:

```
> (remainder 10 3)
1
> (remainder -10.0 3)
-1.0
> (remainder 10.0 -3)
1.0
> (remainder -10 -3)
-1
> (remainder +inf.0 3)
```

remainder: expects type <integer> as 1st argument, given: +inf.0; other arguments were: 3

```
(quotient/remainder n m) → number? number?
  n : integer?
  m : integer?
```

Returns `(values (quotient n m) (remainder n m))`, but the combination is computed more efficiently than separate calls to `quotient` and `remainder`.

Examples:

```
> (quotient/remainder 10 3)
3
1
```

```
(modulo n m) → number?
  n : integer?
  m : integer?
```

Returns `q` with the same sign as *m* where

- `(abs q)` is between `0` (inclusive) and `(abs m)` (exclusive), and
- the difference between `q` and `(- n (* m (quotient n m)))` is a multiple of *m*.

Examples:

```
> (modulo 10 3)
1
> (modulo -10.0 3)
2.0
> (modulo 10.0 -3)
-2.0
> (modulo -10 -3)
-1
> (modulo +inf.0 3)
```
modulo: expects type <integer> as 1st argument, given: +inf.0; other arguments were: 3

`(add1 z) → number?`
  `z : number?`

Returns `(+ z 1)`.

`(sub1 z) → number?`
  `z : number?`

Returns `(- z 1)`.

`(abs x) → number?`
  `x : real?`

Returns the absolute value of `x`.

Examples:
```
> (abs 1.0)
1.0
> (abs -1)
1
```

`(max x ...+) → boolean?`
  `x : real?`

Returns the largest of the `x`s, or `+nan.0` if any `x` is `+nan.0`. If any `x` is inexact, the result is coerced to inexact.

Examples:
```
> (max 1 3 2)
3
> (max 1 3 2.0)
```

```
3.0
```

```
(min x ...+) → boolean?
  x : real?
```

Returns the smallest of the `x`s, or `+nan.0` if any `x` is `+nan.0`. If any `x` is inexact, the result is coerced to inexact.

Examples:

```
> (min 1 3 2)
1
> (min 1 3 2.0)
1.0
```

```
(gcd n ...) → integer?
  n : integer?
```

Returns the greatest common divisor of the `n`s. If no arguments are provided, the result is `0`.

Examples:

```
> (gcd 10)
10
> (gcd 12 81.0)
3.0
```

```
(lcm n ...) → integer?
  n : integer?
```

Returns the least common multiple of the `n`s. If no arguments are provided, the result is `1`.

Examples:

```
> (lcm 10)
10
> (lcm 3 4.0)
12.0
```

```
(round x) → integer?
  x : real?
```

Returns the integer closest to `x`, resolving ties in favor of an even number.

Examples:

```
> (round 17/4)
```

```
 4
 > (round -17/4)
 -4
 > (round 2.5)
 2.0
 > (round -2.5)
 -2.0
```

(floor x) → integer?
  x : real?

Returns the largest integer is that is no more than x.

Examples:

```
 > (floor 17/4)
 4
 > (floor -17/4)
 -5
 > (floor 2.5)
 2.0
 > (floor -2.5)
 -3.0
```

(ceiling x) → integer?
  x : real?

Returns the smallest integer is that is at least as large as x.

Examples:

```
 > (ceiling 17/4)
 5
 > (ceiling -17/4)
 -4
 > (ceiling 2.5)
 3.0
 > (ceiling -2.5)
 -2.0
```

(truncate x) → integer?
  x : real?

Returns the integer farthest from 0 that is no closer to 0 than x.

Examples:

```
> (truncate 17/4)
4
> (truncate -17/4)
-4
> (truncate 2.5)
2.0
> (truncate -2.5)
-2.0
```

```
(numerator q) → integer?
  q : rational?
```

Coreces *q* to an exact number, finds the numerator of the number expressed in its simplest fractional form, and returns this number coerced to the exactness of *q*.

Examples:

```
> (numerator 5)
5
> (numerator 17/4)
17
> (numerator 2.3)
2589569785738035.0
```

```
(denominator q) → integer?
  q : rational?
```

Coreces *q* to an exact number, finds the numerator of the number expressed in its simplest fractional form, and returns this number coerced to the exactness of *q*.

Examples:

```
> (denominator 5)
1
> (denominator 17/4)
4
> (denominator 2.3)
1125899906842624.0
```

```
(rationalize x tolerance) → real?
  x : real?
  tolerance : real?
```

Among the real numbers within `(abs tolerance)` of *x*, returns the one corresponding to an exact number whose `denominator` is smallest. If multiple integers are within *tolerance* of *x*, the one closest to `0` is used.

Examples:

```
> (rationalize 1/4 1/10)
1/3
> (rationalize -1/4 1/10)
-1/3
> (rationalize 1/4 1/4)
0
> (rationalize 11/40 1/4)
1/2
```

3.2.3 Number Comparison

```
(= z w ...+) → boolean?
  z : number?
  w : number?
```

Returns `#t` if all of the arguments are numerically equal, `#f` otherwise. An inexact number is numerically equal to an exact number when the exact coercion of the inexact number is the exact number. Also, `0.0` and `-0.0` are numerically equal, but `+nan.0` is not numerically equal to itself.

Examples:

```
> (= 1 1.0)
#t
> (= 1 2)
#f
> (= 2+3i 2+3i 2+3i)
#t
```

```
(< x y ...+) → boolean?
  x : real?
  y : real?
```

Returns `#t` if the arguments in the given order are in strictly increasing, `#f` otherwise.

Examples:

```
> (< 1 1)
#f
> (< 1 2 3)
#t
> (< 1 +inf.0)
#t
> (< 1 +nan.0)
```

```
#f
```

(<= *x* *y* ...+) → boolean?
  x : real?
  y : real?

Returns `#t` if the arguments in the given order are in non-decreasing, `#f` otherwise.

Examples:
```
> (<= 1 1)
#t
> (<= 1 2 1)
#f
```

(> *x* *y* ...+) → boolean?
  x : real?
  y : real?

Returns `#t` if the arguments in the given order are in strictly decreasing, `#f` otherwise.

Examples:
```
> (> 1 1)
#f
> (> 3 2 1)
#t
> (> +inf.0 1)
#t
> (< +nan.0 1)
#f
```

(>= *x* *y* ...+) → boolean?
  x : real?
  y : real?

Returns `#t` if the arguments in the given order are in non-increasing, `#f` otherwise.

Examples:
```
> (>= 1 1)
#t
> (>= 1 2 1)
#f
```

3.2.4 Powers and Roots

`(sqrt z) → number?`
  `z : number?`

Returns the principal square root of `z`.The result is exact if `z` is exact and `z`'s square root is rational. See also `integer-sqrt`.

Examples:

```
> (sqrt 4/9)
2/3
> (sqrt 2)
1.4142135623730951
> (sqrt -1)
0+1i
```

`(integer-sqrt n) → complex?`
  `n : integer?`

Returns `(floor (sqrt n))` for positive `n`. For negative `n`, the result is `(* (integer-sqrt (- n)) 0+1i)`.

Examples:

```
> (integer-sqrt 4.0)
2.0
> (integer-sqrt 5)
2
```

`(integer-sqrt/remainder n) → integer? integer?`
  `n : integer?`

Returns `(integer-sqrt n)` and `(- n (expt (integer-sqrt n) 2))`.

Examples:

```
> (integer-sqrt/remainder 4.0)
2.0
0.0
> (integer-sqrt/remainder 5)
2
1
```

`(expt z w) → number?`
  `z : number?`

```
w : number?
```

Returns `z` raised to the power of `w`. If `w` is exact `0`, the result is `1`. If `z` is exact `0` and `w` is negative, the `exn:fail:contract` exception is raised.

Examples:

```
> (expt 2 3)
8
> (expt 4 0.5)
2.0
> (expt +inf.0 0)
1
```

```
(exp z) → number?
  z : number?
```

Returns Euler's number raised to the power of `z`. The result is normally inexact, but it is `1` when `z` is an exact `0`.

Examples:

```
> (exp 1)
2.718281828459045
> (exp 2+3i)
-7.315110094901103+1.0427436562359045i
> (exp 0)
1
```

```
(log z) → number?
  z : number?
```

Returns the natural logarithm of `z`. The result is normally inexact, but it is `0` when `z` is an exact `1`.

Examples:

```
> (log (exp 1))
1.0
> (log 2+3i)
1.2824746787307684+0.982793723247329i
> (log 1)
0
```

3.2.5 Trignometric Functions

```
(sin z) → number?
  z : number?
```

Returns the sine of `z`, where `z` is in radians.

Examples:

```
> (sin 3.14159)
2.65358979335273e-06
> (sin 1.0+5.0i)
62.44551846769653+40.0921657779984i
```

```
(cos z) → number?
  z : number?
```

Returns the cosine of `z`, where `z` is in radians.

Examples:

```
> (cos 3.14159)
-0.9999999999964793
> (cos 1.0+5.0i)
40.095806306298826-62.43984868079963i
```

```
(tan z) → number?
  z : number?
```

Returns the tangent of `z`, where `z` is in radians.

Examples:

```
> (tan 0.7854)
1.0000036732118496
> (tan 1.0+5.0i)
8.256719834227411e-05+1.0000377833796008i
```

```
(asin z) → number?
  z : number?
```

Returns the arcsin in radians of `z`.

Examples:

```
> (asin 0.25)
0.25268025514207865
```

```
> (asin 1.0+5.0i)
0.1937931365549317+2.330974653049312i
```

```
(acos z) → number?
  z : number?
```

Returns the arccosine in radians of `z`.

Examples:

```
> (acos 0.25)
1.318116071652818
> (acos 1.0+5.0i)
1.3770031902399644-2.3309746530493123i
```

```
(atan z) → number?
  z : number?
(atan y x) → number?
  y : real?
  x : real?
```

In the one-argument case, returns the arctangent of the inexact approximation of `z`, except that the result is an exact `0` for an exact `0` argument.

In the two-argument case, the result is roughly the same as `(/ (exact->inexact y) (exact->inexact x))`, but the signs of `y` and `x` determine the quadrant of the result. Moreover, a suitable angle is returned when `y` divided by `x` produces `+nan.0` in the case that neither `y` nor `x` is `+nan.0`.

Examples:

```
> (atan 0.5)
0.4636476090008061
> (atan 2 1)
1.1071487177940904
> (atan -2 -1)
-2.0344439357957027
> (atan 1.0+5.0i)
1.530881333938778+0.19442614214700213i
> (atan +inf.0 -inf.0)
2.356194490192345
```

3.2.6 Complex Numbers

```
(make-rectangular x y) → number?
  x : real?
  y : real?
```

Returns `(+ x (* y 0+1i))`.

Examples:

```
> (make-rectangular 3 4.0)
3.0+4.0i
```

```
(make-polar magnitude angle) → number?
  magnitude : real?
  angle : real?
```

Returns `(+ (* magnitude (cos angle)) (* magnitude (sin angle) 0+1i))`.

Examples:

```
> (make-polar 2 3.14159)
-1.9999999999929585+5.30717958670546e-06i
```

```
(real-part z) → real?
  z : number?
```

Returns the real part of the complex number *z* in rectangle coordinates.

Examples:

```
> (real-part 3+4i)
3
> (real-part 5.0)
5.0
```

```
(imag-part z) → real?
  z : number?
```

Returns the imaginary part of the complex number *z* in rectangle coordinates.

Examples:

```
> (imag-part 3+4i)
4
> (imag-part 5.0)
0
> (imag-part 5.0+0.0i)
0.0
```

```
(magnitude z) → (and/c real? (not/c negative?))
  z : number?
```

Returns the magnitude of the complex number `z` in polar coordinates.

Examples:

```
> (magnitude -3)
3
> (magnitude 3.0)
3.0
> (magnitude 3+4i)
5
```

```
(angle z) → real?
  z : number?
```

Returns the angle of the complex number `z` in polar coordinates.

Examples:

```
> (angle -3)
3.141592653589793
> (angle 3.0)
0
> (angle 3+4i)
0.9272952180016122
> (angle +inf.0+inf.0i)
0.7853981633974483
```

3.2.7 Bitwise Operations

```
(bitwise-ior n ...) → exact-integer?
  n : exact-integer?
```

Returns the bitwise "inclusive or" of the `n`s in their (semi-infinite) two's complement representation. If no arguments are provided, the result is `0`.

Examples:

```
> (bitwise-ior 1 2)
3
> (bitwise-ior -32 1)
-31
```

```
(bitwise-and n ...) → exact-integer?
  n : exact-integer?
```

Returns the bitwise “and” of the `n`s in their (semi-infinite) two’s complement representation. If no arguments are provided, the result is `-1`.

Examples:

```
> (bitwise-and 1 2)
0
> (bitwise-and -32 -1)
-32
```

```
(bitwise-xor n ...) → exact-integer?
  n : exact-integer?
```

Returns the bitwise “exclusive or” of the `n`s in their (semi-infinite) two’s complement representation. If no arguments are provided, the result is `0`.

Examples:

```
> (bitwise-xor 1 5)
4
> (bitwise-xor -32 -1)
31
```

```
(bitwise-not n) → exact-integer?
  n : exact-integer?
```

Returns the bitwise “not” of `n` in its (semi-infinite) two’s complement representation.

Examples:

```
> (bitwise-not 5)
-6
> (bitwise-not -1)
0
```

```
(arithmetic-shift n m) → exact-integer?
  n : exact-integer?
  m : exact-integer?
```

Returns the bitwise “shift” of `n` in its (semi-infinite) two’s complement representation. If `m` is non-negative, 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.

Examples:

```
> (arithmetic-shift 1 10)
1024
> (arithmetic-shift 255 -3)
31
```

```
(integer-length n) → exact-integer?
  n : exact-integer?
```

Returns the number of bits in the (semi-infinite) two's complement representation of `n` after removing all leading zeros (for non-negative `n`) or ones (for negative `n`).

Examples:

```
> (integer-length 8)
4
> (integer-length -8)
3
```

3.2.8 Random Numbers

```
(random k [generator]) → nonnegative-exact-integer?
  k : (integer-in 1 4294967087)
  generator : pseudo-random-generator?
            = (current-pseudo-random-generator)
(random [generator]) → (and/c real? inexact? (>/c 0) (</c 1))
  generator : pseudo-random-generator?
            = (current-pseudo-random-generator)
```

When called with and integer argument `k`, returns a random exact integer in the range `0` to `k`-1. When called with zero arguments, returns a random inexact number between `0` and `1`, exclusive.

In each case, the number is provided by the given pseudo-random number generator (which defaults to the current one, as produced by `current-pseudo-random-generator`). The generator maintains an internal state for generating numbers. The random number generator uses a 54-bit version of L'Ecuyer's MRG32k3a algorithm [L'Ecuyer02].

```
(random-seed k) → void?
  k : (integer-in 1 (sub1 (expt 2 31)))
```

Seeds the current pseudo-random number generator with `k`. Seeding a generator sets its internal state deterministically; that is, 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.

```
(make-pseudo-random-generator) → 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) → boolean?
  v : any/c
```

Returns `#t` if *v* is a pseudo-random number generator, `#f` otherwise.

```
(current-pseudo-random-generator) → pseudo-random-generator?
(current-pseudo-random-generator generator) → void?
  generator : pseudo-random-generator?
```

A parameter that determines the pseudo-random number generator used by `random`.

```
(pseudo-random-generator->vector generator) → vector?
  generator : pseudo-random-generator?
```

Produces a vector that represents the complete internal state of *generator*. The vector is suitable as an argument to `vector->pseudo-random-generator` to recreate the generator in its current state (across runs and across platforms).

```
(vector->pseudo-random-generator vec)
 → pseudo-random-generator?
  vec : vector?
```

Produces a pseudo-random number generator whose internal state corresponds to *vec*. The vector *vec* must contain six exact integers; the first three integers must be in the range `0` to `4294967086`, inclusive; the last three integers must be in the range `0` to `4294944442`, inclusive; at least one of the first three integers must be non-zero; and at least one of the last three integers must be non-zero.

```
(vector->pseudo-random-generator! generator
                                  vec)     → void?
  generator : pseudo-random-generator?
  vec : vector?
```

Like `vector->pseudo-random-generator`, but changes *generator* to the given state, instead of creating a new generator.

3.2.9 Number–String Conversions

```
(number->string z [radix]) → string?
  z : number?
  radix : (one-of/c 2 8 10 16) = 10
```

Returns a string that is the printed form of *z* in the base specific by *radix*. If *z* is inexact, *radix* must be `10`, otherwise the `exn:fail:contract` exception is raised.

Examples:

```
> (number->string 3.0)
"3.0"
> (number->string 255 8)
"377"
```

```
(string->number s [radix]) → (or/c number? false/c)
  s : string?
  radix : (integer-in 2 16) = 10
```

Reads and returns a number datum from *s* (see §12.6.3 “Reading Numbers”), returning `#f` if *s* does not parse exactly as a number datum (with no whitespace). The optional *radix* argument specifies the default base for the number, which can be overriden by `#b`, `#o`, `#d`, or `#x` in the string.

Examples:

```
> (string->number "3.0+2.5i")
3.0+2.5i
> (string->number "hello")
#f
> (string->number "111" 7)
57
> (string->number "#b111" 7)
7
```

```
(real->decimal-string n [decimal-digits]) → string?
  n : real?
  decimal-digits : nonnegative-exact-integer? = 2
```

Prints *n* into a string and returns the string. The printed form of *n* shows exactly *decimal-*

digits digits after the decimal point. The printed for uses a minus sign if *n* is negative, and it does not use a plus sign if *n* is positive.

Before printing, *n* is converted to an exact number, multiplied by `(expt 10 decimal-digits)`, rounded, and then divided again by `(expt 10 decimal-digits)`. The result of ths process is an exact number whose decimal representation has no more than *decimal-digits* digits after the decimal (and it is padded with trailing zeros if necessary).

Examples:

```
> (real->decimal-string pi)
"3.14"
> (real->decimal-string pi 5)
"3.14159"
```

```
(integer-bytes->integer  bstr
                         signed?
                        [big-endian?
                         start
                         end])        → exact-integer?
  bstr : bytes?
  signed? : any/c
  big-endian? : any/c = (system-big-endian?)
  start : exact-nonnegative-integer? = 0
  end : exact-nonnegative-integer? = (bytes-length bstr)
```

Converts the machine-format number encoded in *bstr* to an exact integer. The *start* and *end* arguments specify the substring to decode, where `(- end start)` must be 2, 4, or 8. If *signed?* is true, then the bytes are decoded as a two's-complement number, otherwise it is decoded as an unsigned integer. If *big-endian?* is true, then the first character's ASCII value provides the most significant eight bits of the number, otherwise the first character provides the least-significant eight bits, and so on.

```
(integer->integer-bytes  n
                         size-n
                         signed?
                        [big-endian?
                         dest-bstr
                         start])      → bytes?
  n : exact-integer?
  size-n : (one-of/c 2 4 8)
  signed? : any/c
  big-endian? : any/c = (system-big-endian?)
  dest-bstr : (and/c bytes?                = (make-bytes size-n)
                     (not/c immutable?))
  start : exact-nonnegative-integer? = 0
```

Converts the exact integer *n* to a machine-format number encoded in a byte string of length *size-n*, which must be 2, 4, or 8. If *signed?* is true, then the number is encoded as two's complement, otherwise it is encoded as an unsigned bit stream. If *big-endian?* is true, then the most significant eight bits of the number are encoded in the first character of the resulting byte string, otherwise the least-significant bits are encoded in the first byte, and so on.

The *dest-bstr* argument must be a mutable byte string of length *size-n*. The encoding of *n* is written into *dest-bstr* starting at offset *start*, and *dest-bstr* is returned as the result.

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

```
(floating-point-bytes->real  bstr
                            [big-endian?
                             start
                             end])
 → (and/c real? inexact?)
  bstr : bytes?
  big-endian? : any/c = (system-big-endian?)
  start : exact-nonnegative-integer? = 0
  end : exact-nonnegative-integer? = (bytes-length bstr)
```

Converts the IEEE floating-point number encoded in *bstr* from position *start* (inclusive) to *end* (exclusive) to an inexact real number. The difference between *start* an *end* must be either 4 or 8 bytes. If *big-endian?* is true, then the first byte's ASCII value provides the most significant eight bits of the IEEE representation, otherwise the first byte provides the least-significant eight bits, and so on.

```
(real->floating-point-bytes  x
                             size-n
                            [big-endian?
                             dest-bstr
                             start])     → bytes?
  x : real?
  size-n : (one-of/c 4 8)
  big-endian? : any/c = (system-big-endian?)
  dest-bstr : (and/c bytes?                = (make-bytes size-n)
                     (not/c immutable?))
  start : exact-nonnegative-integer? = 0
```

Converts the real number *x* to its IEEE representation in a byte string of length *size-n*, which must be 4 or 8. If *big-endian?* is true, then the most significant eight bits of the number are encoded in the first byte of the resulting byte string, otherwise the least-significant bits are encoded in the first character, and so on.

The *dest-bstr* argument must be a mutable byte string of length *size-n*. The encoding of n is written into *dest-bstr* starting with byte *start*, and *dest-bstr* is returned as the result.

If *dest-bstr* is provided and it has less than *start* plus *size-n* bytes, the `exn:fail:contract` exception is raised.

`(system-big-endian?) → boolean?`

Returns `#t` if the native encoding of numbers is big-endian for the machine running Scheme, `#f` if the native encoding is little-endian.

3.2.10 Extra Constants and Functions

```
(require scheme/math)
```

The bindings documented in this section are provided by the `scheme/math` and `scheme` libraries, but not `scheme/base`.

`pi : real`

An approximation to the ratio of a circle's circumference to its diameter: 3.141592653589793.

```
(sqr z) → number?
  z : number?
```

Returns `(* z z)`.

```
(sgn x) → (one-of/c 1 0 -1 1.0 0.0 -1.0)
  x : real?
```

Returns the sign of *x* as either -1, 0, or 1.

Examples:

```
> (sgn 10)
1
> (sgn -10.0)
-1.0
```

```
> (sgn 0)
0
```

```
(conjugate z) → number?
  z : number?
```

Returns the complex conjugate of `z`.

Examples:

```
> (conjugate 1)
1
> (conjugate 3+4i)
3-4i
```

```
(sinh z) → number?
  z : number?
```

Returns the hyperbolic sine of `z`.

```
(cosh z) → number?
  z : number?
```

Returns the hyperbolic cosine of `z`.

3.3 Strings

§3.4 "Strings (Unicode)" in §"**Guide**: PLT Scheme" introduces strings.

A *string* is a fixed-length array of characters.

A string can be *mutable* or *immutable*. When an immutable string is provided to a procedure like `string-set!`, the `exn:fail:contract` exception is raised. String constants generated by the default reader (see §12.6.6 "Reading Strings") are immutable.

Two strings are `equal?` when they have the same length and contain the same sequence of characters.

A string can be used as a single-valued sequence (see §3.14 "Sequences"). The characters of the string serve as elements of the sequence. See also `in-string`.

See also: `immutable`, `symbol->string`, `bytes->string/utf-8`.

3.3.1 String Constructors, Selectors, and Mutators

```
(string? v) → boolean?
  v : any/c
```

Returns #t if *v* is a string, #f otherwise.

Examples:

```
> (string? "Apple")
#t
> (string? 'apple)
#f
```

```
(make-string k [char]) → string?
  k : exact-nonnegative-integer?
  char : char? = #\nul
```

Returns a new mutable string of length *k* where each position in the string is initialized with the character *char*.

Examples:

```
> (make-string 5 #\z)
"zzzzz"
```

```
(string char ...) → string?
  char : char?
```

Returns a new mutable string whose length is the number of provided *char*s, and whose positions are initialized with the given *char*s.

Examples:

```
> (string #\A #\p #\p #\l #\e)
"Apple"
```

```
(string->immutable-string str) → (and/c string? immutable?)
  str : string?
```

Returns an immutable string with the same content as *str*, returning *str* itself if *str* is immutable.

```
(string-length str) → exact-nonnegative-integer?
  str : string?
```

Returns the length of *str*.

Examples:

```
> (string-length "Apple")
5
```

```
(string-ref str k) → char?
  str : string?
  k : exact-nonnegative-integer?
```

Returns the character at position *k* in *str*. The first position in the string corresponds to `0`, so the position *k* must be less than the length of the string, otherwise the `exn:fail:contract` exception is raised.

Examples:

```
> (string-ref "Apple" 0)
#\A
```

```
(string-set! str k char) → void?
  str : (and/c string? (not/c immutable?))
  k : exact-nonnegative-integer?
  char : char?
```

Changes the character position *k* in *str* to *char*. The first position in the string corresponds to `0`, so the position *k* must be less than the length of the string, otherwise the `exn:fail:contract` exception is raised.

Examples:

```
> (define s (string #\A #\p #\p #\l #\e))
> (string-set! s 4 #\y)
> s
"Apply"
```

```
(substring str start [end]) → string?
  str : string?
  start : exact-nonnegative-integer?
  end : exact-nonnegative-integer? = (string-length str)
```

Returns a new mutable string that is `(- end start)` characters long, and that contains the same characters as *str* from *start* inclusive to *end* exclusive. The *start* and *end* arguments must be less than the length of *str*, and *end* must be greater than or equal to *str*, otherwise the `exn:fail:contract` exception is raised.

Examples:

```
> (substring "Apple" 1 3)
"pp"
> (substring "Apple" 1)
"pple"
```

```
(string-copy str) → string?
  str : string?
```

Returns `(substring str 0)`.

```
(string-copy!  dest
               dest-start
               src
              [src-start
               src-end])   → void?
  dest : (and/c string? (not/c immutable?))
  dest-start : exact-nonnegative-integer?
  src : string?
  src-start : exact-nonnegative-integer? = 0
  src-end : exact-nonnegative-integer? = (string-length src)
```

Changes the characters of *dest* starting at position *dest-start* to match the characters in *src* from *src-start* (inclusive) to *src-end* (exclusive). The strings *dest* and *src* can be the same string, and in that case the destination region can overlap with the source region; the destination characters after the copy match the source characters from before the copy. If any of *dest-start*, *src-start*, or *src-end* are out of range (taking into account the sizes of the strings and the source and destination regions), the `exn:fail:contract` exception is raised.

Examples:

```
> (define s (string #\A #\p #\p #\l #\e))
> (string-copy! s 4 "y")
> (string-copy! s 0 s 3 4)
> s
"lpply"
```

```
(string-fill! dest char) → void?
  dest : (and/c string? (not/c immutable?))
  char : char?
```

Changes *dest* so that every position in the string is filled with *char*.

Examples:

```
> (define s (string #\A #\p #\p #\l #\e))
```

```
> (string-fill! s #\q)
> s
"qqqqq"
```

`(string-append str ...) → string?`
  `str : string?`

Returns a new mutable string that is as long as the sum of the given `str`s' lengths, and that contains the concatenated characters of the given `str`s. If no `str`s are provided, the result is a zero-length string.

Examples:

```
> (string-append "Apple" "Banana")
"AppleBanana"
```

`(string->list str) → (listof char?)`
  `str : string?`

Returns a new list of characters coresponding to the content of `str`. That is, the length of the list is `(string-length (scheme str))`, and the sequence of characters of `str` are in the same sequence in the result list.

Examples:

```
> (string->list "Apple")
(#\A #\p #\p #\l #\e)
```

`(list->string lst) → string?`
  `lst : (listof char?)`

Returns a new mutable string whose content is the list of characters in `lst`. That is, the length of the string is `(length (scheme lst))`, and the sequence of characters in `lst` is in the same sequence in the result string.

Examples:

```
> (list->string (list #\A #\p #\p #\l #\e))
"Apple"
```

`(build-string n proc) → string?`
  `n : exact-nonnegative-integer?`
  `proc : (exact-nonnegative-integer? . -> . char?)`

Creates a string of `n` characters by applying `proc` to the integers from `0` to `(sub1 n)` in order. If `str` is the resulting string, then `(string-ref str i)` is the character produced by (*proc i*).

Examples:

```
> (build-string 5 (lambda (i) (integer->char (+ i 97))))
"abcde"
```

3.3.2 String Comparisons

```
(string=? str1 str2 ...+) → boolean?
  str1 : string?
  str2 : string?
```

Returns `#t` if all of the arguments are `equal?`.

Examples:

```
> (string=? "Apple" "apple")
#f
> (string=? "a" "as" "a")
#f
```

```
(string<? str1 str2 ...+) → boolean?
  str1 : string?
  str2 : string?
```

Returns `#t` if the arguments are lexicographically sorted increasing, where individual characters are ordered by `char<?`, `#f` otherwise.

Examples:

```
> (string<? "Apple" "apple")
#t
> (string<? "apple" "Apple")
#f
> (string<? "a" "b" "c")
#t
```

```
(string<=? str1 str2 ...+) → boolean?
  str1 : string?
  str2 : string?
```

Like `string<?`, but checks whether the arguments are nondecreasing.

Examples:

```
> (string<=? "Apple" "apple")
```

```
#t
> (string<=? "apple" "Apple")
#f
> (string<=? "a" "b" "b")
#t
```

```
(string>? str1 str2 ...+) → boolean?
  str1 : string?
  str2 : string?
```

Like `string<?`, but checks whether the arguments are decreasing.

Examples:

```
> (string>? "Apple" "apple")
#f
> (string>? "apple" "Apple")
#t
> (string>? "c" "b" "a")
#t
```

```
(string>=? str1 str2 ...+) → boolean?
  str1 : string?
  str2 : string?
```

Like `string<?`, but checks whether the arguments are nonincreasing.

Examples:

```
> (string>=? "Apple" "apple")
#f
> (string>=? "apple" "Apple")
#t
> (string>=? "c" "b" "b")
#t
```

```
(string-ci=? str1 str2 ...+) → boolean?
  str1 : string?
  str2 : string?
```

Returns `#t` if all of the arguments are `eqv?` after locale-insensitive case-folding via `string-foldcase`.

Examples:

```
> (string-ci=? "Apple" "apple")
#t
```

```
> (string-ci=? "a" "a" "a")
#t
```

```
(string-ci<? str1 str2 ...+) → boolean?
  str1 : string?
  str2 : string?
```

Like `string<?`, but checks whether the arguments would be in increasing order if each was first case-folded using `string-foldcase` (which is locale-insensitive).

Examples:

```
> (string-ci<? "Apple" "apple")
#f
> (string-ci<? "apple" "banana")
#t
> (string-ci<? "a" "b" "c")
#t
```

```
(string-ci<=? str1 str2 ...+) → boolean?
  str1 : string?
  str2 : string?
```

Like `string-ci<?`, but checks whether the arguments would be nondecreasing after case-folding.

Examples:

```
> (string-ci<=? "Apple" "apple")
#t
> (string-ci<=? "apple" "Apple")
#t
> (string-ci<=? "a" "b" "b")
#t
```

```
(string-ci>? str1 str2 ...+) → boolean?
  str1 : string?
  str2 : string?
```

Like `string-ci<?`, but checks whether the arguments would be decreasing after case-folding.

Examples:

```
> (string-ci>? "Apple" "apple")
#f
> (string-ci>? "banana" "Apple")
```

```
#t
> (string-ci>? "c" "b" "a")
#t
```

```
(string-ci>=? str1 str2 ...+) → boolean?
  str1 : string?
  str2 : string?
```

Like `string-ci<?`, but checks whether the arguments would be nonincreasing after case-folding.

Examples:

```
> (string-ci>=? "Apple" "apple")
#t
> (string-ci>=? "apple" "Apple")
#t
> (string-ci>=? "c" "b" "b")
#t
```

3.3.3 String Conversions

```
(string-upcase str) → string?
  str : string?
```

Returns a string whose characters are the upcase conversion of the characters in *str*. The conversion uses Unicode's locale-independent conversion rules that map code-point sequences to code-point sequences (instead of simply mapping a 1-to-1 function on code points over the string), so the string produced by the conversion can be longer than the input string.

Examples:

```
> (string-upcase "abc!")
"ABC!"
> (string-upcase "Straße")
"STRASSE"
```

```
(string-downcase string) → string?
  string : string?
```

Like `string-upcase`, but the downcase conversion.

Examples:

```
> (string-downcase "aBC!")
```

```
"abc!"
> (string-downcase "Straße")
"straße"
> (string-downcase "ΚΑΟΣ")
"κα ος"
> (string-downcase "Σ")
"σ"
```

```
(string-titlecase string) → string?
  string : string?
```

Like `string-upcase`, but the titlecase conversion only for the first character in each sequence of cased characters in `str` (ignoring case-ignorable characters).

Examples:

```
> (string-titlecase "aBC  twO")
"Abc  Two"
> (string-titlecase "y2k")
"Y2K"
> (string-titlecase "main straße")
"Main Straße"
> (string-titlecase "stra ße")
"Stra Sse"
```

```
(string-foldcase string) → string?
  string : string?
```

Like `string-upcase`, but the case-folding conversion.

Examples:

```
> (string-foldcase "aBC!")
"abc!"
> (string-foldcase "Straße")
"strasse"
> (string-foldcase "ΚΑΟΣ")
"καοσ"
```

```
(string-normalize-nfd string) → string?
  string : string?
```

Returns a string that is the Unicode normalized form D of *string*. If the given string is already in the corresponding Unicode normal form, the string may be returned directly as the result (instead of a newly allocated string).

```
(string-normalize-nfkd string) → string?
  string : string?
```

Like `string-normalize-nfd`, but for normalized form KD.

```
(string-normalize-nfc string) → string?
  string : string?
```

Like `string-normalize-nfd`, but for normalized form C.

```
(string-normalize-nfkc string) → string?
  string : string?
```

Like `string-normalize-nfd`, but for normalized form KC.

3.3.4 Locale-Specific String Operations

```
(string-locale=? str1 str2 ...+) → boolean?
  str1 : string?
  str2 : string?
```

Like `string=?`, but the strings are compared in a locale-specific way, based the value of `current-locale`. See §12.1.1 “Encodings and Locales” for more information on locales.

```
(string-locale<? str1 str2 ...+) → boolean?
  str1 : string?
  str2 : string?
```

Like `string<?`, but the sort order compares strings in a locale-specific way, based the value of `current-locale`. In particular, the sort order may not be simply a lexicographic extension of character ordering.

```
(string-locale>? str1 str2 ...+) → boolean?
  str1 : string?
  str2 : string?
```

Like `string>?`, but locale-specific like `string-locale<?`.

```
(string-locale-ci=? str1 str2 ...+) → boolean?
```

```
str1 : string?
str2 : string?
```

Like `string-locale=?`, but strings are compared using rules that are both locale-specific and case-insensitive (depending on what "case-insensitive" means for the current locale).

```
(string-locale-ci<? str1 str2 ...+) → boolean?
  str1 : string?
  str2 : string?
```

Like `string<?`, but both locale-sensitive and case-insensitive like `string-locale-ci=?`.

```
(string-locale-ci>? str1 str2 ...+) → boolean?
  str1 : string?
  str2 : string?
```

Like `string>?`, but both locale-sensitive and case-insensitive like `string-locale-ci=?`.

```
(string-locale-upcase string) → string?
  string : string?
```

Like `string-upcase`, but using locale-specific case-conversion rules based the value of `current-locale`.

```
(string-locale-downcase string) → string?
  string : string?
```

Like `string-downcase`, but using locale-specific case-conversion rules based the value of `current-locale`.

3.3.5 Additional String Functions

```
(require scheme/string)
```

The bindings documented in this section are provided by the `scheme/string` and `scheme` libraries, but not `scheme/base`.

```
(string-append* str ... strs) → string?
  str : string?
  strs : (listof string?)
```

Like `string-append`, but the last argument is used as a list of arguments for `string-append`, so `(string-append* str ... strs)` is the same as `(apply string-append str ... strs)`. In other words, the relationship between `string-append` and `string-append*` is similar to the one between `list` and `list*`.

Examples:

```
> (string-append* "a" "b" '("c" "d"))
"abcd"
> (string-append* (cdr (append* (map (lambda (x) (list ", " x))
                                      '("Alpha" "Beta" "Gamma")))))
"Alpha, Beta, Gamma"
```

3.4 Byte Strings

§3.5 "Bytes and Byte Strings" in §"**Guide**: PLT Scheme" introduces byte strings.

A *byte string* is a fixed-length arary of bytes. A *byte* is an exact integer between `0` and `255` inclusive.

A byte string can be *mutable* or *immutable*. When an immutable byte string is provided to a procedure like `bytes-set!`, the `exn:fail:contract` exception is raised. Byte-string constants generated by the default reader (see §12.6.6 "Reading Strings") are immutable.

Two byte strings are `equal?` when they have the same length and contain the same sequence of bytes.

A byte string can be used as a single-valued sequence (see §3.14 "Sequences"). The bytes of the string serve as elements of the sequence. See also `in-bytes`.

See also: `immutable`.

3.4.1 Byte String Constructors, Selectors, and Mutators

```
(bytes? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a byte string, `#f` otherwise.

Examples:

```
> (bytes? #"Apple")
#t
> (bytes? "Apple")
#f
```

```
(make-bytes k [b]) → bytes?
  k : exact-nonnegative-integer?
  b : byte? = 0
```

Returns a new mutable byte string of length *k* where each position in the byte string is initialized with the byte *b*.

Examples:

```
> (make-bytes 5 65)
#"AAAAA"
```

```
(bytes b ...) → bytes?
  b : byte?
```

Returns a new mutable byte string whose length is the number of provided *b*s, and whose positions are initialized with the given *b*s.

Examples:

```
> (bytes 65 112 112 108 101)
#"Apple"
```

```
(bytes->immutable-bytes bstr) → (and/c bytes?
                                       immutable?)
  bstr : bytes?
```

Returns an immutable byte string with the same content as *bstr*, returning *bstr* itself if *bstr* is immutable.

```
(byte? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a byte (i.e., an exact integer between `0` and `255` inclusive), `#f` otherwise.

Examples:

```
> (byte? 65)
#t
> (byte? 0)
#t
> (byte? 256)
#f
> (byte? -1)
#f
```

```
(bytes-length bstr) → exact-nonnegative-integer?
  bstr : bytes?
```

Returns the length of *bstr*.

Examples:

```
> (bytes-length #"Apple")
5
```

```
(bytes-ref bstr k) → byte?
  bstr : bytes?
  k : exact-nonnegative-integer?
```

Returns the character at position *k* in *bstr*. The first position in the bytes cooresponds to `0`, so the position *k* must be less than the length of the bytes, otherwise the `exn:fail:contract` exception is raised.

Examples:

```
> (bytes-ref #"Apple" 0)
65
```

```
(bytes-set! bstr k b) → void?
  bstr : (and/c bytes? (not/c immutable?))
  k : exact-nonnegative-integer?
  b : byte?
```

Changes the character position *k* in *bstr* to *b*. The first position in the byte string coorесponds to `0`, so the position *k* must be less than the length of the bytes, otherwise the `exn:fail:contract` exception is raised.

Examples:

```
> (define s (bytes 65 112 112 108 101))
> (bytes-set! s 4 121)
> s
#"Apply"
```

```
(subbytes bstr start [end]) → bytes?
  bstr : bytes?
  start : exact-nonnegative-integer?
  end : exact-nonnegative-integer? = (bytes-length str)
```

Returns a new mutable byte string that is `(- end start)` bytes long, and that contains the same bytes as *bstr* from *start* inclusive to *end* exclusive. The *start* and *end* arguments must be less than the length of *bstr*, and *end* must be greater than or equal to *bstr*, otherwise the `exn:fail:contract` exception is raised.

Examples:

```
> (subbytes #"Apple" 1 3)
#"pp"
> (subbytes #"Apple" 1)
#"pple"
```

```
(bytes-copy bstr) → bytes?
  bstr : bytes?
```

Returns `(subbytes str 0)`.

```
(bytes-copy! dest
             dest-start
             src
            [src-start
             src-end])  → void?
  dest : (and/c bytes? (not/c immutable?))
  dest-start : exact-nonnegative-integer?
  src : bytes?
  src-start : exact-nonnegative-integer? = 0
  src-end : exact-nonnegative-integer? = (bytes-length src)
```

Changes the bytes of *dest* starting at position *dest-start* to match the bytes in *src* from *src-start* (inclusive) to *src-end* (exclusive). The bytes strings *dest* and *src* can be the same byte string, and in that case the destination region can overlap with the source region; the destination bytes after the copy match the source bytes from before the copy. If any of *dest-start*, *src-start*, or *src-end* are out of range (taking into account the sizes of the bytes strings and the source and destination regions), the `exn:fail:contract` exception is raised.

Examples:

```
> (define s (bytes 65 112 112 108 101))
> (bytes-copy! s 4 #"y")
> (bytes-copy! s 0 s 3 4)
> s
#"lpply"
```

```
(bytes-fill! dest char) → void?
  dest : (and/c bytes? (not/c immutable?))
  char : char?
```

Changes *dest* so that every position in the bytes is filled with *char*.

Examples:

```
> (define s (bytes 65 112 112 108 101))
> (bytes-fill! s 113)
> s
#"qqqqq"
```

```
(bytes-append bstr ...) → bytes?
  bstr : bytes?
```

Returns a new mutable byte string that is as long as the sum of the given *bstr*s' lengths, and that contains the concatenated bytes of the given *bstr*s. If no *bstr*s are provided, the result is a zero-length byte string.

Examples:

```
> (bytes-append #"Apple" #"Banana")
#"AppleBanana"
```

```
(bytes->list bstr) → (listof byte?)
  bstr : bytes?
```

Returns a new list of bytes coresponding to the content of *bstr*. That is, the length of the list is `(bytes-length (scheme bstr))`, and the sequence of bytes of *bstr* are in the same sequence in the result list.

Examples:

```
> (bytes->list #"Apple")
(65 112 112 108 101)
```

```
(list->bytes lst) → bytes?
  lst : (listof byte?)
```

Returns a new mutable bytes whose content is the list of bytes in *lst*. That is, the length of the bytes is `(length (scheme lst))`, and the sequence of bytes in *lst* is in the same sequence in the result bytes.

Examples:

```
> (list->bytes (list 65 112 112 108 101))
#"Apple"
```

3.4.2 Byte String Comparisons

```
(bytes=? bstr1 bstr2 ...+) → boolean?
```

```
bstr1 : bytes?
bstr2 : bytes?
```

Returns `#t` if all of the arguments are `eqv?`.

Examples:

```
> (bytes=? #"Apple" #"apple")
#f
> (bytes=? #"a" #"as" #"a")
#f
```

```
(bytes<? bstr1 bstr2 ...+) → boolean?
  bstr1 : bytes?
  bstr2 : bytes?
```

Returns `#t` if the arguments are lexicographically sorted increasing, where individual bytes are ordered by `<`, `#f` otherwise.

Examples:

```
> (bytes<? #"Apple" #"apple")
#t
> (bytes<? #"apple" #"Apple")
#f
> (bytes<? #"a" #"b" #"c")
#t
```

```
(bytes>? bstr1 bstr2 ...+) → boolean?
  bstr1 : bytes?
  bstr2 : bytes?
```

Like `bytes<?`, but checks whether the arguments are decreasing.

Examples:

```
> (bytes>? #"Apple" #"apple")
#f
> (bytes>? #"apple" #"Apple")
#t
> (bytes>? #"c" #"b" #"a")
#t
```

3.4.3 Bytes to/from Characters, Decoding and Encoding

```
(bytes->string/utf-8 bstr [err-char start end]) → string?
  bstr : bytes?
  err-char : (or/c false/c char?) = #f
  start : exact-nonnegative-integer? = 0
  end : exact-nonnegative-integer? = (bytes-length bstr)
```

Produces a string by decoding the *start* to *end* substring of *bstr* as a UTF-8 encoding of Unicode code points. If *err-char* is not `#f`, then it is used for bytes that fall in the range `128` to `255` but are not part of a valid encoding sequence. (This is consistent with reading characters from a port; see §12.1.1 "Encodings and Locales" for more details.) If *err-char* is `#f`, and if the *start* to *end* substring of *bstr* is not a valid UTF-8 encoding overall, then the `exn:fail:contract` exception is raised.

```
(bytes->string/locale  bstr
                      [err-char
                       start
                       end])     → string?
  bstr : bytes?
  err-char : (or/c false/c char?) = #f
  start : exact-nonnegative-integer? = 0
  end : exact-nonnegative-integer? = (bytes-length bstr)
```

Produces a string by decoding the *start* to *end* substring of *bstr* using the current locale's encoding (see also §12.1.1 "Encodings and Locales"). If *err-char* is not `#f`, it is used for each byte in *bstr* that is not part of a valid encoding; if *err-char* is `#f`, and if the *start* to *end* substring of *bstr* is not a valid encoding overall, then the `exn:fail:contract` exception is raised.

```
(bytes->string/latin-1  bstr
                       [err-char
                        start
                        end])     → string?
  bstr : bytes?
  err-char : (or/c false/c char?) = #f
  start : exact-nonnegative-integer? = 0
  end : exact-nonnegative-integer? = (bytes-length bstr)
```

Produces a string by decoding the *start* to *end* substring of *bstr* as a Latin-1 encoding of Unicode code points; i.e., each byte is translated directly to a character using `integer->char`, so the decoding always succeeds. (See also the Latin-1 footnote of §12.1.1 "Encodings and Locales".) The *err-char* argument is ignored, but present for consistency with the other operations.

```
(string->bytes/utf-8 str [err-byte start end]) → bytes?
```

```
str : string?
err-byte : (or/c false/c byte?) = #f
start : exact-nonnegative-integer? = 0
end : exact-nonnegative-integer? = (string-length str)
```

Produces a byte string by encoding the *start* to *end* substring of *str* via UTF-8 (always succeeding). The *err-byte* argument is ignored, but included for consistency with the other operations.

```
(string->bytes/locale str [err-byte start end]) → bytes?
  str : string?
  err-byte : (or/c false/c byte?) = #f
  start : exact-nonnegative-integer? = 0
  end : exact-nonnegative-integer? = (string-length str)
```

Produces a string by encoding the *start* to *end* substring of *str* using the current locale's encoding (see also §12.1.1 "Encodings and Locales"). If *err-byte* is not `#f`, it is used for each character in *str* that cannot be encoded for the current locale; if *err-byte* is `#f`, and if the *start* to *end* substring of *str* cannot be encoded, then the `exn:fail:contract` exception is raised.

```
(string->bytes/latin-1 str
                       [err-byte
                        start
                        end])     → bytes?
  str : string?
  err-byte : (or/c false/c byte?) = #f
  start : exact-nonnegative-integer? = 0
  end : exact-nonnegative-integer? = (string-length str)
```

Produces a string by encoding the *start* to *end* substring of *str* using Latin-1; i.e., each character is translated directly to a byte using `char->integer`. If *err-byte* is not `#f`, it is used for each character in *str* whose value is greater than `255`. (See also the Latin-1 footnote of §12.1.1 "Encodings and Locales". If *err-byte* is `#f`, and if the *start* to *end* substring of *str* has a character with a value greater than `255`, then the `exn:fail:contract` exception is raised.

```
(string-utf-8-length str [start end]) → exact-nonnegative-integer?
  str : string?
  start : exact-nonnegative-integer? = 0
  end : exact-nonnegative-integer? = (string-lenght str)
```

Returns the length in bytes of the UTF-8 encoding of *str*'s substring from *start* to *end*, but without actually generating the encoded bytes.

```
(bytes-utf-8-length bstr [err-char start end])
 → exact-nonnegative-integer?
  bstr : bytes?
  err-char : (or/c false/c char?) = #f
  start : exact-nonnegative-integer? = 0
  end : exact-nonnegative-integer? = (bytes-length bstr)
```

Returns the length in characters of the UTF-8 decoding of *bstr*'s substring from *start* to *end*, but without actually generating the decoded characters. If *err-char* is `#f` and the substring is not a UTF-8 encoding overall, the result is `#f`. Otherwise, *err-char* is used to resolve decoding errors as in `bytes->string/utf-8`.

```
(bytes-utf-8-ref bstr [skip err-char start end]) → char?
  bstr : bytes?
  skip : exact-nonnegative-integer? = 0
  err-char : (or/c false/c char?) = #f
  start : exact-nonnegative-integer? = 0
  end : exact-nonnegative-integer? = (bytes-length bstr)
```

Returns the *skip*th character in the UTF-8 decoding of *bstr*'s substring from *start* to *end*, but without actually generating the other decoded characters. If the substring is not a UTF-8 encoding up to the *skip*th character (when *err-char* is `#f`), or if the substring decoding produces fewer than *skip* characters, the result is `#f`. If *err-char* is not `#f`, it is used to resolve decoding errors as in `bytes->string/utf-8`.

```
(bytes-utf-8-index bstr
                  [skip
                   err-char
                   start
                   end])     → exact-nonnegative-integer?
  bstr : bytes?
  skip : exact-nonnegative-integer? = 0
  err-char : (or/c false/c char?) = #f
  start : exact-nonnegative-integer? = 0
  end : exact-nonnegative-integer? = (bytes-length bstr)
```

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

3.4.4 Bytes to Bytes Encoding Conversion

```
(bytes-open-converter from-name to-name) → bytes-converter?
  from-name : string?
  to-name : string?
```

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

Certain encoding combinations are always available:

- `(bytes-open-converter "UTF-8" "UTF-8")` — the identity conversion, except that encoding errors in the input lead to a decoding failure.
- `(bytes-open-converter "UTF-8-permissive" "UTF-8")` — the identity conversion, except that any input byte that is not part of a valid encoding sequence is effectively replaced by the UTF-8 encoding sequence for `#\`. (This handling of invalid sequences is consistent with the interpretation of port bytes streams into characters; see §12.1 "Ports".)
- `(bytes-open-converter "" "UTF-8")` — converts from the current locale's default encoding (see §12.1.1 "Encodings and Locales") to UTF-8.
- `(bytes-open-converter "UTF-8" "")` — converts from UTF-8 to the current locale's default encoding (see §12.1.1 "Encodings and Locales").
- `(bytes-open-converter "platform-UTF-8" "platform-UTF-16")` — converts UTF-8 to UTF-16 under Unix and Mac OS X, where each UTF-16 code unit is a sequence of two bytes ordered by the current platform's endianess. Under Windows, the input can include encodings that are not valid UTF-8, but which naturally extend the UTF-8 encoding to support unpaired surrogate code units, and the output is a sequence of UTF-16 code units (as little-endian byte pairs), potentially including unpaired surrogates.
- `(bytes-open-converter "platform-UTF-8-permissive" "platform-UTF-16")` — like `(bytes-open-converter "platform-UTF-8" "platform-UTF-16")`, but an input byte that is not part of a valid UTF-8 encoding sequence (or valid for the unpaired-surrogate extension under Windows) is effectively replaced with `(char->integer #\?)`.
- `(bytes-open-converter "platform-UTF-16" "platform-UTF-8")` — converts UTF-16 (bytes orderd by the current platform's endianness) to UTF-8 under

Unix and Mac OS X. Under Windows, the input can include UTF-16 code units that are unpaired surrogates, and the corresponding output includes an encoding of each surrogate in a natural extension of UTF-8. Under Unix and Mac OS X, surrogates are assumed to be paired: a pair of bytes with the bits 55296 starts a surrogate pair, and the 1023 bits are used from the pair and following pair (independent of the value of the 56320 bits). On all platforms, performance may be poor when decoding from an odd offset within an input byte string.

A newly opened byte converter is registered with the current custodian (see §13.6 "Custodians"), so that the converter is closed when the custodian is shut down. A converter is not registered with a custodian (and does not need to be closed) if it is one of the guaranteed combinations not involving `""` under Unix, or if it is any of the guaranteed combinations (including `""`) under Windows and Mac OS X.

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

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

```
(bytes-close-converter converter) → void
  converter : bytes-converter?
```

Closes the given converter, so that it can no longer be used with `bytes-convert` or `bytes-convert-end`.

```
(bytes-convert  converter
                src-bstr
               [src-start-pos
                src-end-pos
                dest-bstr
                dest-start-pos
                dest-end-pos])
 → (or/c bytes? nonnegative-exact-integer?)
   nonnegative-exact-integer?
   (one-of 'complete 'continues 'aborts 'error)
  converter : bytes-converter?
  src-bstr : bytes?
  src-start-pos : nonnegative-exact-integer? = 0
  src-end-pos : nonnegative-exact-integer?
              = (bytes-length src-bstr)
  dest-bstr : (or/c bytes? false/c) = #f
  dest-start-pos : nonnegative-exact-integer? = 0
```

```
dest-end-pos : (or/c nonnegative-exact-integer? false/c)
             = (and dest-bstr
                    (bytes-length dest-bstr))
```

Converts the bytes from `src-start-pos` to `src-end-pos` in `src-bstr`.

If `dest-bstr` is not `#f`, the converted byte are written into `dest-bstr` from `dest-start-pos` to `dest-end-pos`. If `dest-bstr` is `#f`, then a newly allocated byte string holds the conversion results, and if `dest-end-pos` is not `#f`, the size of the result byte string is no more than `(- dest-end-pos dest-start-pos)`.

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

- `result-bstr` or `dest-wrote-amt` — a byte string if `dest-bstr` is `#f` or not provided, or the number of bytes written into `dest-bstr` otherwise.
- `src-read-amt` — the number of bytes successfully converted from `src-bstr`.
- `'complete`, `'continues`, `'aborts`, or `'error` — indicates how conversion terminated:
  - `'complete`: The entire input was processed, and `src-read-amt` will be equal to `(- src-end-pos src-start-pos)`.
  - `'continues`: Conversion stopped due to the limit on the result size or the space in `dest-bstr`; in this case, fewer than `(- dest-end-pos dest-start-pos)` bytes may be returned if more space is needed to process the next complete encoding sequence in `src-bstr`.
  - `'aborts`: The input stopped part-way through an encoding sequence, and more input bytes are necessary to continue. For example, if the last byte of input is `195` for a `"UTF-8-permissive"` decoding, the result is `'aborts`, because another byte is needed to determine how to use the `195` byte.
  - `'error`: The bytes starting at `(+ src-start-pos src-read-amt)` bytes in `src-bstr` do not form a legal encoding sequence. This result is never produced for some encodings, where all byte sequences are valid encodings. For example, since `"UTF-8-permissive"` handles an invalid UTF-8 sequence by dropping characters or generating "?," every byte sequence is effectively valid.

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

```
(bytes-convert-end  converter
                   [dest-bstr
                    dest-start-pos
                    dest-end-pos])
 → (or/c bytes? nonnegative-exact-integer?)
   (one-of 'complete 'continues)
  converter : bytes-converter?
  dest-bstr : (or/c bytes? false/c) = #f
  dest-start-pos : nonnegative-exact-integer? = 0
  dest-end-pos :  (or/c nonnegative-exact-integer? false/c)
               = (and dest-bstr
                      (bytes-length dest-bstr))
```

Like `bytes-convert`, but instead of converting bytes, this procedure generates an ending sequence for the conversion (sometimes called a "shift sequence"), if any. Few encodings use shift sequences, so this function will succeed with no output for most encodings. In any case, successful output of a (possibly empty) shift sequence resets the converter to its initial state.

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

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

```
(bytes-converter? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a byte converter produced by `bytes-open-converter`, `#f` otherwise.

```
(locale-string-encoding) → any
```

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

3.5 Characters

§3.3 "Characters" in §"**Guide**: PLT Scheme" introduces characters.

MzScheme characters range over Unicode scalar values, which includes characters whose values range from #x0 to #x10FFFF, but not including #xD800 to #xDFFF.

Two characters are eqv? if they correspond to the same scalar value. For each scalar value less than 256, character values that are eqv? are also eq?.

3.5.1 Characters and Scalar Values

```
(char? v) → boolean?
  v : any/c
```

Return #t if *v* is a character, #f otherwise.

```
(char->integer char) → exact-integer?
  char : char?
```

Returns a character's code-point number.

Examples:

```
> (char->integer #\A)
65
```

```
(integer->char k) → char?
  k : (and/c exact-integer?
             (or/c (integer-in 0 55295)
                   (integer-in 57344 1114111)))
```

Return the character whose code-point number is *k*. For *k* less than 256, the result is the same object for the same *k*.

Examples:

```
> (integer->char 65)
#\A
```

```
(char-utf-8-length char) → (integer-in 1 6)
  char : char?
```

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

3.5.2 Character Comparisons

```
(char=? char1 char2 ...+) → boolean?
  char1 : char?
  char2 : char?
```

Returns `#t` if all of the arguments are `eqv?`.

Examples:

```
> (char=? #\a #\a)
#t
> (char=? #\a #\A #\a)
#f
```

```
(char<? char1 char2 ...+) → boolean?
  char1 : char?
  char2 : char?
```

Returns `#t` if the arguments are sorted increasing, where two characters are ordered by their scalar values, `#f` otherwise.

Examples:

```
> (char<? #\A #\a)
#t
> (char<? #\a #\A)
#f
> (char<? #\a #\b #\c)
#t
```

```
(char<=? char1 char2 ...+) → boolean?
  char1 : char?
  char2 : char?
```

Like `char<?`, but checks whether the arguments are nondecreasing.

Examples:

```
> (char<=? #\A #\a)
#t
> (char<=? #\a #\A)
#f
> (char<=? #\a #\b #\b)
#t
```

```
(char>? char1 char2 ...+) → boolean?
  char1 : char?
  char2 : char?
```

Like `char<?`, but checks whether the arguments are decreasing.

Examples:

```
> (char>? #\A #\a)
#f
> (char>? #\a #\A)
#t
> (char>? #\c #\b #\a)
#t
```

```
(char>=? char1 char2 ...+) → boolean?
  char1 : char?
  char2 : char?
```

Like `char<?`, but checks whether the arguments are nonincreasing.

Examples:

```
> (char>=? #\A #\a)
#f
> (char>=? #\a #\A)
#t
> (char>=? #\c #\b #\b)
#t
```

```
(char-ci=? char1 char2 ...+) → boolean?
  char1 : char?
  char2 : char?
```

Returns `#t` if all of the arguments are `eqv?` after locale-insensitive case-folding via `char-foldcase`.

Examples:

```
> (char-ci=? #\A #\a)
#t
> (char-ci=? #\a #\a #\a)
#t
```

```
(char-ci<? char1 char2 ...+) → boolean?
  char1 : char?
```

```
char2 : char?
```

Like `char<?`, but checks whether the arguments would be in increasing order if each was first case-folded using `char-foldcase` (which is locale-insensitive).

Examples:

```
> (char-ci<? #\A #\a)
#f
> (char-ci<? #\a #\b)
#t
> (char-ci<? #\a #\b #\c)
#t
```

```
(char-ci<=? char1 char2 ...+) → boolean?
  char1 : char?
  char2 : char?
```

Like `char-ci<?`, but checks whether the arguments would be nondecreasing after case-folding.

Examples:

```
> (char-ci<=? #\A #\a)
#t
> (char-ci<=? #\a #\A)
#t
> (char-ci<=? #\a #\b #\b)
#t
```

```
(char-ci>? char1 char2 ...+) → boolean?
  char1 : char?
  char2 : char?
```

Like `char-ci<?`, but checks whether the arguments would be decreasing after case-folding.

Examples:

```
> (char-ci>? #\A #\a)
#f
> (char-ci>? #\b #\A)
#t
> (char-ci>? #\c #\b #\a)
#t
```

```
(char-ci>=? char1 char2 ...+) → boolean?
  char1 : char?
```

char2 : char?

Like char-ci<?, but checks whether the arguments would be nonincreasing after case-folding.

Examples:

```
> (char-ci>=? #\A #\a)
#t
> (char-ci>=? #\a #\A)
#t
> (char-ci>=? #\c #\b #\b)
#t
```

3.5.3 Classifications

(char-alphabetic? *char*) → boolean?
  char : char?

Returns #t if *char* has the Unicode "Alphabetic" property.

(char-lower-case? *char*) → boolean?
  char : char?

Returns #t if *char* has the Unicode "Lowercase" property.

(char-upper-case? *char*) → boolean?
  char : char?

Returns #t if *char* has the Unicode "Uppercase" property.

(char-title-case? *char*) → boolean?
  char : char?

Returns #t if *char*'s Unicode general category is Lt, #f otherwise.

(char-numeric? *char*) → boolean?
  char : char?

Returns #t if *char* has the Unicode "Numeric" property.

```
(char-symbolic? char) → boolean?
  char : char?
```

Returns `#t` if *char*'s Unicode general category is Sm, Sc, Sk, or So, `#f` otherwise.

```
(char-punctuation? char) → boolean?
  char : char?
```

Returns `#t` if *char*'s Unicode general category is Pc, Pd, Ps, Pe, Pi, Pf, or Po, `#f` otherwise.

```
(char-graphic? char) → boolean?
  char : char?
```

Returns `#t` if *char*'s Unicode general category is Mn, Mc, Me, or if one of the following produces `#t` when applied to *char*: `char-alphabetic?`, `char-numeric?`, `char-symbolic?`, or `char-punctuation?`.

```
(char-whitespace? char) → boolean?
  char : char?
```

Returns `#t` if *char* has the Unicode "White_Space" property.

```
(char-blank? char) → boolean?
  char : char?
```

Returns `#t` if *char*'s Unicode general category is Zs or if *char* is `#\tab`. (These correspond to horizontal whitespace.)

```
(char-iso-control? char) → boolean?
  char : char?
```

Return `#t` if *char* is between `#\nul` and `#\u001F` inclusive or `#\rubout` and `#\u009F` inclusive.

```
(char-general-category char) → symbol?
  char : char?
```

Returns a symbol representing the character's Unicode general category, which is `'lu`, `'ll`, `'lt`, `'lm`, `'lo`, `'mn`, `'mc`, `'me`, `'nd`, `'nl`, `'no`, `'ps`, `'pe`, `'pi`, `'pf`, `'pd`, `'pc`, `'po`, `'sc`, `'sm`, `'sk`, `'so`, `'zs`, `'zp`, `'zl`, `'cc`, `'cf`, `'cs`, `'co`, or `'cn`.

```
(make-known-char-range-list)
 → (listof (list/c nonnegative-integer?
                   nonnegative-integer?
                   boolean?))
```

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

3.5.4 Character Conversions

```
(char-upcase char) → char?
  char : char?
```

Produces a character consistent with the 1-to-1 code point mapping defined by Unicode. If *char* has no upcase mapping, `char-upcase` produces *char*.

String procedures, such as `string-upcase`, handle the case where Unicode defines a locale-independent mapping from the code point to a code-point sequence (in addition to the 1-1 mapping on scalar values).

Examples:

```
> (char-upcase #\a)
#\A
> (char-upcase #\λ)
#\Λ
> (char-upcase #\space)
#\space
```

```
(char-downcase char) → char?
  char : char?
```

Like `char-upcase`, but for the Unicode downcase mapping.

Examples:

```
> (char-downcase #\A)
#\a
> (char-downcase #\Λ)
#\λ
> (char-downcase #\space)
#\space
```

```
(char-titlecase char) → char?
  char : char?
```

Like `char-upcase`, but for the Unicode titlecase mapping.

Examples:

```
> (char-upcase #\a)
#\A
> (char-upcase #\λ)
#\Λ
> (char-upcase #\space)
#\space
```

```
(char-foldcase char) → char?
  char : char?
```

Like `char-upcase`, but for the Unicode case-folding mapping.

Examples:

```
> (char-foldcase #\A)
#\a
> (char-foldcase #\Σ)
#\σ
> (char-foldcase #\ς)
#\σ
> (char-foldcase #\space)
#\space
```

3.6 Symbols

§3.6 "Symbols" in §"**Guide**: PLT Scheme" introduces symbols.

A *symbol* is like an immutable string, but symbols are normally *interned*, so that two symbols with the same character content are normally `eq?`. All symbols produced by the default reader (see §12.6.2 "Reading Symbols") are interned.

The two procedures `string->uninterned-symbol` and `gensym` generate *uninterned* symbols, i.e., symbols that are not `eq?`, `eqv?`, or `equal?` to any other symbol, although they may print the same as other symbols.

Regular (interned) symbols are only weakly held by the internal symbol table. This weakness can never affect the result of an `eq?`, `eqv?`, or `equal?` test, but a symbol may disappear when placed into a weak box (see §15.1 "Weak Boxes") used as the key in a weak hash table (see §3.13 "Hash Tables"), or used as an ephemeron key (see §15.2 "Ephemerons").

```
(symbol? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a symbol, `#f` otherwise.

Examples:

```
> (symbol? 'Apple)
#t
> (symbol? 10)
#f
```

```
(symbol->string sym) → symbol?
  sym : symbol?
```

Returns a freshly allocated mutable string whose characters are the same as in *sym*.

Examples:

```
> (symbol->string 'Apple)
"Apple"
```

```
(string->symbol str) → symbol?
  str : string?
```

Returns an interned symbol whose characters are the same as in *str*.

Examples:

```
> (string->symbol "Apple")
Apple
> (string->symbol "1")
|1|
```

```
(string->uninterned-symbol str) → symbol?
  str : string?
```

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

Examples:

```
> (string->uninterned-symbol "Apple")
Apple
> (eq? 'a (string->uninterned-symbol "a"))
#f
```

```
(gensym [base]) → symbol?
  base : (or/c string? symbol?) = "g"
```

Returns a new uninterned symbol with an automatically-generated name. The optional `base` argument is a prefix symbol or string.

Examples:

```
> (gensym "apple")
apple5347
```

3.7 Regular Expressions

Regular expressions are specified as strings or byte strings, using the same pattern language as the Unix utility `egrep` or Perl. A string-specified pattern produces a character regexp matcher, and a byte-string pattern produces a byte regexp matcher. If a character regexp is used with a byte string or input port, it matches UTF-8 encodings (see §12.1.1 "Encodings and Locales") of matching character streams; if a byte regexp is used with a character string, it matches bytes in the UTF-8 encoding of the string.

Regular expressions can be compiled into a *regexp value* for repeated matches. The `regexp` and `byte-regexp` procedures convert a string or byte string (respectively) into a regexp value using one syntax of regular expressions that is most compatible to `egrep`. The `pregexp` and `byte-pregexp` procedures produce a regexp value using a slightly different syntax of regular expressions that is more compatible with Perl. In addition, Scheme constants written with `#rx` or `#px` (see §12.6 "The Reader") produce compiled regexp values.

The internal size of a regexp value is limited to 32 kilobytes; this limit roughly corresponds to a source string with 32,000 literal characters or 5,000 operators.

3.7.1 Regexp Syntax

The following syntax specifications describe the content of a string that represents a regular expression. The syntax of the corresponding string may involve extra escape characters. For example, the regular expression `(.*)\1` can be represented with the string `"(.*)\\1"` or the regexp constant `#rx"(.*)\\1"`; the `\` in the regular expression must be escaped to include it in a string or regexp constant.

The `regexp` and `pregexp` syntaxes share a common core:

⟨*regexp*⟩	::=	⟨*pces*⟩	Match ⟨*pces*⟩
	|	⟨*regexp*⟩|⟨*regexp*⟩	Match either ⟨*regexp*⟩, try left first
⟨*pces*⟩	::=	⟨*pce*⟩	Match ⟨*pce*⟩
	|	⟨*pce*⟩⟨*pces*⟩	Match ⟨*pce*⟩ followed by ⟨*pces*⟩

⟨*pce*⟩	::=	⟨*repeat*⟩	Match ⟨*repeat*⟩, longest possible
	|	⟨*repeat*⟩`?`	Match ⟨*repeat*⟩, shortest possible
	|	⟨*atom*⟩	Match ⟨*atom*⟩ exactly once
⟨*repeat*⟩	::=	⟨*atom*⟩`*`	Match ⟨*atom*⟩ 0 or more times
	|	⟨*atom*⟩`+`	Match ⟨*atom*⟩ 1 or more times
	|	⟨*atom*⟩`?`	Match ⟨*atom*⟩ 0 or 1 times
⟨*atom*⟩	::=	`(`⟨*regexp*⟩`)`	Match sub-expression ⟨*regexp*⟩ and report
	|	`[`⟨*rng*⟩`]`	Match any character in ⟨*rng*⟩
	|	`[^`⟨*rng*⟩`]`	Match any character not in ⟨*rng*⟩
	|	`.`	Match any (except newline in multi mode)
	|	`^`	Match start (or after newline in multi mode)
	|	`$`	Match end (or before newline in multi mode)
	|	⟨*literal*⟩	Match a single literal character
	|	`(?`⟨*mode*⟩`:`⟨*regexp*⟩`)`	Match ⟨*regexp*⟩ using ⟨*mode*⟩
	|	`(?>`⟨*regexp*⟩`)`	Match ⟨*regexp*⟩, only first possible
	|	⟨*look*⟩	Match empty if ⟨*look*⟩ matches
	|	`(?`⟨*tst*⟩⟨*pces*⟩`|`⟨*pces*⟩`)`	Match 1st ⟨*pces*⟩ if ⟨*tst*⟩, else 2nd ⟨*pces*⟩
	|	`(?`⟨*tst*⟩⟨*pces*⟩`)`	Match ⟨*pces*⟩ if ⟨*tst*⟩, empty if not ⟨*tst*⟩
⟨*rng*⟩	::=	`]`	⟨*rng*⟩ contains `]` only
	|	`-`	⟨*rng*⟩ contains `-` only
	|	⟨*mrng*⟩	⟨*rng*⟩ contains everything in ⟨*mrng*⟩
	|	⟨*mrng*⟩`-`	⟨*rng*⟩ contains `-` and everything in ⟨*mrng*⟩
⟨*mrng*⟩	::=	`]`⟨*lrng*⟩	⟨*mrng*⟩ contains `]` and everything in ⟨*lrng*⟩
	|	`-`⟨*lrng*⟩	⟨*mrng*⟩ contains `-` and everything in ⟨*lrng*⟩
	|	⟨*lrng*⟩	⟨*mrng*⟩ contains everything in ⟨*lrng*⟩
⟨*lrng*⟩	::=	⟨*rliteral*⟩	⟨*lrng*⟩ contains a literal character
	|	⟨*rliteral*⟩`-`⟨*rliteral*⟩	⟨*lrng*⟩ contains Unicode range inclusive
	|	⟨*lrng*⟩⟨*lrng*⟩	⟨*lrng*⟩ contains everything in both
⟨*look*⟩	::=	`(?=`⟨*regexp*⟩`)`	Match if ⟨*regexp*⟩ matches
	|	`(?!`⟨*regexp*⟩`)`	Match if ⟨*regexp*⟩ doesn't match
	|	`(?<=`⟨*regexp*⟩`)`	Match if ⟨*regexp*⟩ matches preceeding
	|	`(?<!`⟨*regexp*⟩`)`	Match if ⟨*regexp*⟩ doesn't match preceeding
⟨*tst*⟩	::=	`(`⟨*n*⟩`)`	True if Nth `(` has a match
	|	⟨*look*⟩	True if ⟨*look*⟩ matches
⟨*mode*⟩	::=		Like the enclosing mode
	|	⟨*mode*⟩`i`	Like ⟨*mode*⟩, but case-insensitive
	|	⟨*mode*⟩`-i`	Like ⟨*mode*⟩, but sensitive
	|	⟨*mode*⟩`s`	Like ⟨*mode*⟩, but not in multi mode
	|	⟨*mode*⟩`-s`	Like ⟨*mode*⟩, but in multi mode
	|	⟨*mode*⟩`m`	Like ⟨*mode*⟩, but in multi mode
	|	⟨*mode*⟩`-m`	Like ⟨*mode*⟩, but not in multi mode

The following completes the grammar for `regexp`, which treats `{` and `}` as literals, `\` as a literal within ranges, and `\` as a literal producer outside of ranges.

⟨*literal*⟩ ::= Any character except `(`, `)`, `*`, `+`, `?`, `[`, `.`, `^`, `\`, or `|`

	|	`\`⟨*aliteral*⟩	Match ⟨*aliteral*⟩
⟨*aliteral*⟩	::=	Any character	
⟨*rliteral*⟩	::=	Any character except `]` or `-`	

The following completes the grammar for `pregexp`, which uses `{` and `}` bounded repetition and uses `\` for meta-characters both inside and outside of ranges.

⟨*repeat*⟩	::=	...	...
	|	⟨*atom*⟩`{`⟨*n*⟩`}`	Match ⟨*atom*⟩ exactly ⟨*n*⟩ times
	|	⟨*atom*⟩`{`⟨*n*⟩`,}`	Match ⟨*atom*⟩ ⟨*n*⟩ or more times
	|	⟨*atom*⟩`{,`⟨*m*⟩`}`	Match ⟨*atom*⟩ between 0 and ⟨*m*⟩ times
	|	⟨*atom*⟩`{`⟨*n*⟩`,`⟨*m*⟩`}`	Match ⟨*atom*⟩ between ⟨*n*⟩ and ⟨*m*⟩ times
⟨*atom*⟩	::=	...	...
	|	`\`⟨*n*⟩	Match latest reported match for ⟨*n*⟩th `(`
	|	⟨*class*⟩	Match any character in ⟨*class*⟩
	|	`\b`	Match \w* boundary
	|	`\B`	Match where \b does not
	|	`\p{`⟨*property*⟩`}`	Match (UTF-8 encoded) in ⟨*property*⟩
	|	`\P{`⟨*property*⟩`}`	Match (UTF-8 encoded) not in ⟨*property*⟩
⟨*literal*⟩	::=	Any character except `(`, `)`, `*`, `+`, `?`, `[`, `]`, `{`, `}`, `.`, `^`, `\`, or `|`	
	|	`\`⟨*aliteral*⟩	Match ⟨*aliteral*⟩
⟨*aliteral*⟩	::=	Any character except `a`-`z`, `A`-`Z`, `0`-`9`	
⟨*lrng*⟩	::=	...	...
	|	⟨*class*⟩	⟨*lrng*⟩ contains all characters in ⟨*class*⟩
	|	⟨*posix*⟩	⟨*lrng*⟩ contains all characters in ⟨*posix*⟩
	|	`\`⟨*eliteral*⟩	⟨*lrng*⟩ contains ⟨*eliteral*⟩
⟨*rliteral*⟩	::=	Any character except `]`, `\`, or `-`	
⟨*eliteral*⟩	::=	Any character except `a`-`z`, `A`-`Z`	
⟨*class*⟩	::=	`\d`	Contains `0`-`9`
	|	`\D`	Contains ASCII other than those in \d
	|	`\w`	Contains `a`-`z`, `A`-`Z`, `0`-`9`, `_`
	|	`\W`	Contains ASCII other than those in \w
	|	`\s`	Contains space, tab, newline, formfeed, return
	|	`\S`	Contains ASCII other than those in \s
⟨*posix*⟩	::=	`[:alpha:]`	Contains `a`-`z`, `A`-`Z`
	|	`[:alnum:]`	Contains `a`-`z`, `A`-`Z`, `0`-`9`
	|	`[:ascii:]`	Contains all ASCII characters
	|	`[:blank:]`	Contains space and tab
	|	`[:cntrl:]`	Contains all characters with scalar value < 32
	|	`[:digit:]`	Contains `0`-`9`
	|	`[:graph:]`	Contains all ASCII characters that use ink
	|	`[:lower:]`	Contains space, tab, and ASCII ink users
	|	`[:print:]`	Contains `A`-`Z`
	|	`[:space:]`	Contains space, tab, newline, formfeed, return
	|	`[:upper:]`	Contains `A`-`Z`
	|	`[:word:]`	Contains `a`-`z`, `A`-`Z`, `0`-`9`, `_`

<table>
<tr><td></td><td>|</td><td><code>[:xdigit:]</code></td><td>Contains <code>0-9</code>, <code>a-f</code>, <code>A-F</code></td></tr>
<tr><td>⟨property⟩</td><td>::=</td><td>⟨category⟩</td><td>Includes all characters in ⟨category⟩</td></tr>
<tr><td></td><td>|</td><td><code>^</code>⟨category⟩</td><td>Includes all characters not in ⟨category⟩</td></tr>
<tr><td>⟨category⟩</td><td>::=</td><td><code>Ll</code> | <code>Lu</code> | <code>Lt</code> | <code>Lm</code></td><td>Unicode general category</td></tr>
<tr><td></td><td>|</td><td><code>L&</code></td><td>Union of Ll, Lu, Lt, and Lm</td></tr>
<tr><td></td><td>|</td><td><code>Lo</code></td><td>Unicode general category</td></tr>
<tr><td></td><td>|</td><td><code>L</code></td><td>Union of L& and Lo</td></tr>
<tr><td></td><td>|</td><td><code>Nd</code> | <code>Nl</code> | <code>No</code></td><td>Unicode general category</td></tr>
<tr><td></td><td>|</td><td><code>N</code></td><td>Union of Nd, Nl, and No</td></tr>
<tr><td></td><td>|</td><td><code>Ps</code> | <code>Pe</code> | <code>Pi</code> | <code>Pf</code></td><td>Unicode general category</td></tr>
<tr><td></td><td>|</td><td><code>Pc</code> | <code>Pd</code> | <code>Po</code></td><td>Unicode general category</td></tr>
<tr><td></td><td>|</td><td><code>P</code></td><td>Union of Ps, Pe, Pi, Pf, Pc, Pd, and Po</td></tr>
<tr><td></td><td>|</td><td><code>Mn</code> | <code>Mc</code> | <code>Me</code></td><td>Unicode general category</td></tr>
<tr><td></td><td>|</td><td><code>M</code></td><td>Union of Mn, Mc, and Me</td></tr>
<tr><td></td><td>|</td><td><code>Sc</code> | <code>Sk</code> | <code>Sm</code> | <code>So</code></td><td>Unicode general category</td></tr>
<tr><td></td><td>|</td><td><code>S</code></td><td>Union of Sc, Sk, Sm, and So</td></tr>
<tr><td></td><td>|</td><td><code>Zl</code> | <code>Zp</code> | <code>Zs</code></td><td>Unicode general category</td></tr>
<tr><td></td><td>|</td><td><code>Z</code></td><td>Union of Zl, Zp, and Zs</td></tr>
<tr><td></td><td>|</td><td><code>.</code></td><td>Union of all general categories</td></tr>
</table>

3.7.2 Additional Syntactic Constraints

In addition to matching a grammars, regular expressions must meet two syntactic restrictions:

- In a ⟨*repeat*⟩ other than ⟨*atom*⟩`?`, then ⟨*atom*⟩ must not match an empty sequence.
- In a `(?<=`⟨*regexp*⟩`)` or `(?<!`⟨*regexp*⟩`)`, the ⟨*regexp*⟩ must match a bounded sequence, only.

These contraints are checked syntactically by the following type system. A type $[n, m]$ corresponds to an expression that matches between n and m characters. In the rule for `(`⟨*Regexp*⟩`)`, N means the number such that the opening parenthesis is the Nth opening parenthesis for collecting match reports. Non-emptiness is inferred for a backreference pattern, `\`⟨N⟩, so that a backreference can be used for repetition patterns; in the case of mutual dependencies among backreferences, the inference chooses the fixpoint that maximizes non-emptiness. Finiteness is not inferred for backreferences (i.e., a backreference is assumed to match an arbitrarily large sequence).

$$\frac{\langle regexp\rangle_1 \;:\; [n_1, m_1] \qquad \langle regexp\rangle_2 \;:\; [n_2, m_2]}{}$$

$$\frac{}{\langle regexp\rangle_1\,\texttt{|}\,\langle regexp\rangle_2 \;:\; [\min(n_1, n_2), \max(m_1, m_2)]}$$

$$\frac{\langle pce\rangle \;:\; [n_1, m_1] \qquad \langle pces\rangle \;:\; [n_2, m_2]}{\langle pce\rangle\langle pces\rangle \;:\; [n_1{+}n_2, m_1{+}m_2]}$$

$$\frac{\langle repeat\rangle \;:\; [n, m]}{\langle repeat\rangle\texttt{?} \;:\; [0, m]} \qquad \frac{\langle atom\rangle \;:\; [n, m] \qquad n > 0}{\langle atom\rangle\texttt{*} \;:\; [0, \infty]}$$

$$\frac{\langle atom\rangle \;:\; [n, m] \qquad n > 0}{\langle atom\rangle\texttt{+} \;:\; [1, \infty]} \qquad \frac{\langle atom\rangle \;:\; [n, m]}{\langle atom\rangle\texttt{?} \;:\; [0, m]}$$

$$\frac{\langle atom\rangle \;:\; [n, m] \qquad n > 0}{\langle atom\rangle\texttt{\{}\langle n\rangle\texttt{\}} \;:\; [n{*}\langle n\rangle, m{*}\langle n\rangle]}$$

$$\frac{\langle atom\rangle \;:\; [n, m] \qquad n > 0}{\langle atom\rangle\texttt{\{}\langle n\rangle\texttt{,\}} \;:\; [n{*}\langle n\rangle, \infty]}$$

$$\frac{\langle atom\rangle \;:\; [n, m] \qquad n > 0}{\langle atom\rangle\texttt{\{,}\langle m\rangle\texttt{\}} \;:\; [0, m{*}\langle m\rangle]}$$

$$\frac{\langle atom\rangle \;:\; [n, m] \qquad n > 0}{\langle atom\rangle\texttt{\{}\langle n\rangle\texttt{,}\langle m\rangle\texttt{\}} \;:\; [n{*}\langle n\rangle, m{*}\langle m\rangle]}$$

$$\frac{\langle regexp\rangle \;:\; [n, m]}{\texttt{(}\langle regexp\rangle\texttt{)} \;:\; [n, m] \qquad \alpha_N{=}n}$$

$$\frac{\langle regexp\rangle \;:\; [n, m]}{\texttt{(?}\langle mode\rangle\texttt{:}\langle regexp\rangle\texttt{)} \;:\; [n, m]}$$

$$\frac{\langle regexp\rangle \;:\; [n, m]}{\texttt{(?=}\langle regexp\rangle\texttt{)} \;:\; [0, 0]} \qquad \frac{\langle regexp\rangle \;:\; [n, m]}{\texttt{(?!}\langle regexp\rangle\texttt{)} \;:\; [0, 0]}$$

$$\frac{\langle regexp\rangle \;:\; [n, m] \qquad m < \infty}{\texttt{(?<=}\langle regexp\rangle\texttt{)} \;:\; [0, 0]} \qquad \frac{\langle regexp\rangle \;:\; [n, m] \qquad m < \infty}{\texttt{(?<!}\langle regexp\rangle\texttt{)} \;:\; [0, 0]}$$

$$\frac{\langle regexp\rangle \;:\; [n, m]}{\texttt{(?>}\langle regexp\rangle\texttt{)} \;:\; [n, m]}$$

$$\frac{\langle tst\rangle \;:\; [n_0, m_0] \qquad \langle pces\rangle_1 \;:\; [n_1, m_1] \qquad \langle pces\rangle_2 \;:\; [n_2, m_2]}{\texttt{(?}\langle tst\rangle\langle pces\rangle_1\texttt{|}\langle pces\rangle_2\texttt{)} \;:\; [\min(n_1, n_2), \max(m_1, m_2)]}$$

$$\frac{\langle tst\rangle \;:\; [n_0, m_0] \qquad \langle pces\rangle \;:\; [n_1, m_1]}{\texttt{(?}\langle tst\rangle\langle pces\rangle\texttt{)} \;:\; [0, m_1]}$$

$(\langle n \rangle)$: $[\alpha_N, \infty]$ $[\langle rng \rangle]$: $[1, 1]$ $[\hat{}\langle rng \rangle]$: $[1, 1]$

$.$: $[1, 1]$ `^` : $[0, 0]$ `$` : $[0, 0]$

$\langle literal \rangle$: $[1, 1]$ $\backslash\langle n \rangle$: $[\alpha_N, \infty]$ $\langle class \rangle$: $[1, 1]$

`\b` : $[0, 0]$ `\B` : $[0, 0]$

`\p{`$\langle property \rangle$`}` : $[1, 6]$ `\P{`$\langle property \rangle$`}` : $[1, 6]$

3.7.3 Regexp Constructors

`(regexp? v) → boolean?`
  `v : any/c`

Returns `#t` if *v* is a regexp value created by `regexp` or `pregexp`, `#f` otherwise.

`(pregexp? v) → boolean?`
  `v : any/c`

Returns `#t` if *v* is a regexp value created by `pregexp` (not `regexp`), `#f` otherwise.

`(byte-regexp? v) → boolean?`
  `v : any/c`

Returns `#t` if *v* is a regexp value created by `byte-regexp` or `byte-pregexp`, `#f` otherwise.

`(byte-pregexp? v) → boolean?`
  `v : any/c`

Returns `#t` if *v* is a regexp value created by `byte-pregexp` (not `byte-regexp`), `#f` otherwise.

`(regexp str) → regexp?`
  `str : string?`

Takes a string representation of a regular expression (using the syntax in §3.7.1 "Regexp Syntax") and compiles it into a regexp value. Other regular expression procedures accept either a string or a regexp value as the matching pattern. If a regular expression string is used multiple times, it is faster to compile the string once to a regexp value and use it for repeated matches instead of using the string each time.

The `object-name` procedure returns the source string for a regexp value.

Examples:

```
> (regexp "ap*le")
#rx"ap*le"
> (object-name #rx"ap*le")
"ap*le"
```

```
(pregexp string) → pregexp?
  string : string?
```

Like `regexp`, except that it uses a slightly different syntax (see §3.7.1 "Regexp Syntax"). The result can be used with `regexp-match`, etc., just like the result from `regexp`.

Examples:

```
> (pregexp "ap*le")
#px"ap*le"
> (regexp? #px"ap*le")
#t
```

```
(byte-regexp bstr) → byte-regexp?
  bstr : bytes?
```

Takes a byte-string representation of a regular expression (using the syntax in §3.7.1 "Regexp Syntax") and compiles it into a byte-regexp value.

The `object-name` procedure returns the source byte string for a regexp value.

Examples:

```
> (byte-regexp #"ap*le")
#rx#"ap*le"
> (object-name #rx#"ap*le")
#"ap*le"
> (byte-regexp "ap*le")
byte-regexp: expects argument of type <byte string>; given
"ap*le"
```

```
(byte-pregexp bstr) → byte-pregexp?
  bstr : bytes?
```

Like `byte-regexp`, except that it uses a slightly different syntax (see §3.7.1 "Regexp Syntax"). The result can be used with `regexp-match`, etc., just like the result from `byte-`

regexp.

Examples:

```
> (byte-pregexp #"ap*le")
#px#"ap*le"
```

```
(regexp-quote str [case-sensitive?]) → string?
  str : string?
  case-sensitive? : any/c = #t
(regexp-quote bstr [case-sensitive?]) → bytes?
  bstr : bytes?
  case-sensitive? : any/c = #t
```

Produces a string or byte string suitable for use with `regexp` to match the literal sequence of characters in *str* or sequence of bytes in *bstr*. If *case-sensitive?* is true, the resulting regexp matches letters in *str* or `bytes` case-insensitively, otherwise it matches case-sensitively.

Examples:

```
> (regexp-match "." "apple.scm")
("a")
> (regexp-match (regexp-quote ".") "apple.scm")
(".")
```

3.7.4 Regexp Matching

```
(regexp-match pattern
              input
             [start-pos
              end-pos
              output-port])
 → (or/c (listof (or/c (cons (or/c string? bytes?)
                             (or/c string? bytes?))
                       false/c))
         false/c)
  pattern : (or/c string? bytes? regexp? byte-regexp?)
  input : (or/c string? bytes? input-port?)
  start-pos : nonnegative-exact-integer? = 0
  end-pos : (or/c nonnegative-exact-integer? false/c) = #f
  output-port : (or/c output-port? false/c) = #f
```

Attempts to match *pattern* (a string, byte string, regexp value, or byte-regexp value) once to a portion of *input*. The matcher finds a portion of *input* that matches and is closest to the start of the input (after `start-pos`).

The optional `start-pos` and `end-pos` arguments select a portion of `input` for matching; the default is the entire string or the stream up to an end-of-file. When `input` is a string, `start-pos` is a character position; when `input` is a byte string, then `start-pos` is a byte position; and when `input` is an input port, `start-pos` is the number of bytes to skip before starting to match. The `end-pos` argument can be `#f`, which corresponds to the end of the string or the end-of-file in the stream; otherwise, it is a character or byte position, like `start-pos`. If `input` is an input port, and if the end-of-file is reached before `start-pos` bytes are skipped, then the match fails.

In `pattern`, a start-of-string `^` refers to the first position of `input` after `start-pos`, and the end-of-input `$` refers to the `end-pos`th position or (in the case of an input port) the end of file, whichever comes first.

If the match fails, `#f` is returned. If the match succeeds, a list containing strings or byte string, and possibly `#f`, is returned. The list contains strings only if `input` is a string and `pattern` is not a byte regexp value. Otherwise, the list contains byte strings (substrings of the UTF-8 encoding of `input`, if `input` is a string).

The first [byte] string in a result list is the portion of `input` that matched `pattern`. If two portions of `input` can match `pattern`, then the match that starts earliest is found.

Additional [byte] strings are returned in the list if `pattern` contains parenthesized sub-expressions (but not when the open parenthesis is followed by `?:`). Matches for the sub-expressions are provided in the order of the opening parentheses in `pattern`. When sub-expressions occur in branches of an `|` "or" pattern, in a `*` "zero or more" pattern, or other places where the overall pattern can succeed without a match for the sub-expression, then a `#f` is returned for the sub-expression if it did not contribute to the final match. When a single sub-expression occurs within a `*` "zero or more" pattern or other multiple-match positions, then the rightmost match associated with the sub-expression is returned in the list.

If the optional `output-port` is provided as an output port, the part of `input` from its beginning (not `start-pos`) that precedes the match is written to the port. All of `input` up to `end-pos` is written to the port if no match is found. This functionality is most useful when `input` is an input port.

When matching an input port, a match failure reads up to `end-pos` bytes (or end-of-file), even if `pattern` begins with a start-of-string `^`; see also `regexp-try-match`. On success, all bytes up to and including the match are eventually read from the port, but matching proceeds by first peeking bytes from the port (using `peek-bytes-avail!`), and then (re-)reading matching bytes to discard them after the match result is determined. Non-matching bytes may be read and discarded before the match is determined. The matcher peeks in blocking mode only as far as necessary to determine a match, but it may peek extra bytes to fill an internal buffer if immediately available (i.e., without blocking). Greedy repeat operators in `pattern`, such as `*` or `+`, tend to force reading the entire content of the port (up to `end-pos`) to determine a match.

If the input port is read simultaneously by another thread, or if the port is a custom port with inconsistent reading and peeking procedures (see §12.1.9 "Custom Ports"), then the bytes that are peeked and used for matching may be different than the bytes read and discarded after the match completes; the matcher inspects only the peeked bytes. To avoid such interleaving, use `regexp-match-peek` (with a `progress-evt` argument) followed by `port-commit-peeked`.

Examples:

```
> (regexp-match #rx"x." "12x4x6")
("x4")
> (regexp-match #rx"y." "12x4x6")
#f
> (regexp-match #rx"x." "12x4x6" 3)
("x6")
> (regexp-match #rx"x." "12x4x6" 3 4)
#f
> (regexp-match #rx#"x." "12x4x6")
(#"x4")
> (regexp-match #rx"x." "12x4x6" 0 #f (current-output-port))
12
("x4")
> (regexp-match #rx"(-[0-9]*)+" "a-12--345b")
("-12--345" "-345")
```

```
(regexp-match* pattern
               input
              [start-pos
               end-pos]) → (listof (or/c string? bytes?))
  pattern : (or/c string? bytes? regexp? byte-regexp?)
  input : (or/c string? bytes? input-port?)
  start-pos : nonnegative-exact-integer? = 0
  end-pos : (or/c nonnegative-exact-integer? false/c) = #f
```

Like `regexp-match`, but the result is a list of strings or byte strings corresponding to a sequence of matches of *pattern* in *input*. (Unlike `regexp-match`, results for parenthesized sub-patterns in *pattern* are not returned.) If *pattern* matches a zero-length string or byte sequence along the way, the `exn:fail` exception is raised.

If *input* contains no matches (in the range *start-pos* to *end-pos*), `null` is returned. Otherwise, each item in the resulting list is a distinct substring or byte sequence from *input* that matches *pattern*. The *end-pos* argument can be `#f` to match to the end of *input* (which corresponds to an end-of-file if *input* is an input port).

Examples:

```
> (regexp-match* #rx"x." "12x4x6")
("x4" "x6")
```

```
(regexp-try-match  pattern
                   input
                  [start-pos
                   end-pos
                   output-port])
 → (or/c (listof (or/c (cons (or/c string? bytes?)
                             (or/c string? bytes?))
                       false/c))
         false/c)
  pattern : (or/c string? bytes? regexp? byte-regexp?)
  input : input-port?
  start-pos : nonnegative-exact-integer? = 0
  end-pos : (or/c nonnegative-exact-integer? false/c) = #f
  output-port : (or/c output-port? false/c) = #f
```

Like `regexp-match` on input ports, except that if the match fails, no characters are read and discarded from `in`.

This procedure is especially useful with a *`pattern`* that begins with a start-of-string `^` or with a non-`#f` *`end-pos`*, since each limits the amount of peeking into the port. Otherwise, beware that a large portion of the stream may be peeked (and therefore pulled into memory) before the match succeeds or fails.

```
(regexp-match-positions  pattern
                         input
                        [start-pos
                         end-pos
                         output-port])
 → (or/c (listof (or/c (cons nonnegative-exact-integer?
                             nonnegative-exact-integer?)
                       false/c))
         false/c)
  pattern : (or/c string? bytes? regexp? byte-regexp?)
  input : (or/c string? bytes? input-port?)
  start-pos : nonnegative-exact-integer? = 0
  end-pos : (or/c nonnegative-exact-integer? false/c) = #f
  output-port : (or/c output-port? false/c) = #f
```

Like `regexp-match`, but returns a list of number pairs (and `#f`) instead of a list of strings. Each pair of numbers refers to a range of characters or bytes in *`input`*. If the result for the same arguments with `regexp-match` would be a list of byte strings, the resulting ranges correspond to byte ranges; in that case, if *`input`* is a character string, the byte ranges correspond to bytes in the UTF-8 encoding of the string.

Range results are returned in a `substring`- and `subbytes`-compatible manner, independent of *start-pos*. In the case of an input port, the returned positions indicate the number of bytes that were read, including *start-pos*, before the first matching byte.

Examples:

```
> (regexp-match-positions #rx"x." "12x4x6")
((2 . 4))
> (regexp-match-positions #rx"x." "12x4x6" 3)
((4 . 6))
> (regexp-match-positions #rx"(-[0-9]*)+" "a-12--345b")
((1 . 9) (5 . 9))
```

```
(regexp-match-positions* pattern
                          input
                         [start-pos
                          end-pos])
 → (listof (cons nonnegative-exact-integer?
                 nonnegative-exact-integer?))
  pattern : (or/c string? bytes? regexp? byte-regexp?)
  input : (or/c string? bytes? input-port?)
  start-pos : nonnegative-exact-integer? = 0
  end-pos : (or/c nonnegative-exact-integer? false/c) = #f
```

Like `regexp-match-positions`, but returns multiple matches like `regexp-match*`.

Examples:

```
> (regexp-match-positions #rx"x." "12x4x6")
((2 . 4))
```

```
(regexp-match? pattern
               input
              [start-pos
               end-pos
               output-port]) → boolean?
  pattern : (or/c string? bytes? regexp? byte-regexp?)
  input : (or/c string? bytes? input-port?)
  start-pos : nonnegative-exact-integer? = 0
  end-pos : (or/c nonnegative-exact-integer? false/c) = #f
  output-port : (or/c output-port? false/c) = #f
```

Like `regexp-match`, but returns merely `#t` when the match succeeds, `#f` otherwise.

Examples:

```
> (regexp-match? #rx"x." "12x4x6")
#t
```

```
> (regexp-match? #rx"y." "12x4x6")
#f
```

```
(regexp-match-exact? pattern input) → boolean?
  pattern : (or/c string? bytes? regexp? byte-regexp?)
  input : (or/c string? bytes? input-port?)
```

Like `regexp-match?`, but `#t` is only returned when the entire content of *`input`* matches *`pattern`*.

Examples:

```
> (regexp-match-exact? #rx"x." "12x4x6")
#f
> (regexp-match-exact? #rx"1.*x." "12x4x6")
#t
```

```
(regexp-match-peek  pattern
                    input
                   [start-pos
                    end-pos
                    progress])
 → (or/c (listof (or/c (cons bytes? bytes?)
                       false/c))
         false/c)
  pattern : (or/c string? bytes? regexp? byte-regexp?)
  input : input-port?
  start-pos : nonnegative-exact-integer? = 0
  end-pos : (or/c nonnegative-exact-integer? false/c) = #f
  progress : (or/c evt false/c) = #f
```

Like `regexp-match` on input ports, but only peeks bytes from `input-port` instead of reading them. Furthermore, instead of an output port, the last optional argument is a progress event for `input-port` (see `port-progress-evt`). If *`progress`* becomes ready, then the match stops peeking from `input-port` and returns `#f`. The *`progress`* argument can be `#f`, in which case the peek may continue with inconsistent information if another process meanwhile reads from `input-port`.

Examples:

```
> (define p (open-input-string "a abcd"))
> (regexp-match-peek ".*bc" p)
(#"a abc")
> (regexp-match-peek ".*bc" p 2)
(#"abc")
> (regexp-match ".*bc" p 2)
(#"abc")
```

```
> (peek-char p)
#\d
> (regexp-match ".*bc" p)
#f
> (peek-char p)
#<eof>
```

```
(regexp-match-peek-positions  pattern
                              input
                             [start-pos
                              end-pos
                              progress])
 → (or/c (listof (or/c (cons nonnegative-exact-integer?
                             nonnegative-exact-integer?)
                       false/c))
         false/c)
  pattern : (or/c string? bytes? regexp? byte-regexp?)
  input : input-port?
  start-pos : nonnegative-exact-integer? = 0
  end-pos : (or/c nonnegative-exact-integer? false/c) = #f
  progress : (or/c evt false/c) = #f
```

Like `regexp-match-positions` on input ports, but only peeks bytes from `input-port` instead of reading them, and with a *progress* argument like `regexp-match-peek`.

```
(regexp-match-peek-immediate  pattern
                              input
                             [start-pos
                              end-pos
                              progress])
 → (or/c (listof (or/c (cons bytes? bytes?)
                       false/c))
         false/c)
  pattern : (or/c string? bytes? regexp? byte-regexp?)
  input : input-port?
  start-pos : nonnegative-exact-integer? = 0
  end-pos : (or/c nonnegative-exact-integer? false/c) = #f
  progress : (or/c evt false/c) = #f
```

Like `regexp-match-peek`, but it attempts to match only bytes that are available from `input-port` without blocking. The match fails if not-yet-available characters might be used to match *pattern*.

```
(regexp-match-peek-positions-immediate  pattern
                                        input
                                       [start-pos
                                        end-pos
                                        progress])
 → (or/c (listof (or/c (cons nonnegative-exact-integer?
                             nonnegative-exact-integer?)
                       false/c))
         false/c)
  pattern : (or/c string? bytes? regexp? byte-regexp?)
  input : input-port?
  start-pos : nonnegative-exact-integer? = 0
  end-pos : (or/c nonnegative-exact-integer? false/c) = #f
  progress : (or/c evt false/c) = #f
```

Like `regexp-match-peek-positions`, but it attempts to match only bytes that are available from `input-port` without blocking. The match fails if not-yet-available characters might be used to match *pattern*.

```
(regexp-match-peek-positions*  pattern
                               input
                              [start-pos
                               end-pos])
 → (listof (cons nonnegative-exact-integer?
                 nonnegative-exact-integer?))
  pattern : (or/c string? bytes? regexp? byte-regexp?)
  input : input-port?
  start-pos : nonnegative-exact-integer? = 0
  end-pos : (or/c nonnegative-exact-integer? false/c) = #f
```

Like `regexp-match-peek-positions`, but returns multiple matches like `regexp-match*`.

3.7.5 Regexp Splitting

```
(regexp-split  pattern
               input
              [start-pos
               end-pos]) → (listof (or/c string? bytes?))
  pattern : (or/c string? bytes? regexp? byte-regexp?)
  input : (or/c string? bytes? input-port?)
  start-pos : nonnegative-exact-integer? = 0
  end-pos : (or/c nonnegative-exact-integer? false/c) = #f
```

The complement of regexp-match*: the result is a list of strings (if *pattern* is a string or character regexp and *input* is a string) or byte strings (otherwise) from in *input* that are separated by matches to *pattern*. Adjacent matches are separated with "" or #"". If *pattern* matches a zero-length string or byte sequence along the way, the exn:fail exception is raised.

If *input* contains no matches (in the range *start-pos* to *end-pos*), the result is a list containing *input*'s content (from *start-pos* to *end-pos*) as a single element. If a match occurs at the beginning of *input* (at *start-pos*), the resulting list will start with an empty string or byte string, and if a match occurs at the end (at *end-pos*), the list will end with an empty string or byte string. The *end-pos* argument can be #f, in which case splitting goes to the end of *input* (which corresponds to an end-of-file if *input* is an input port).

Examples:

```
> (regexp-split #rx"x" "12x4x6")
("12" "4" "6")
> (regexp-split #rx"." "12x4x6")
("" "" "" "" "" "" "")
```

3.7.6 Regexp Substitution

```
(regexp-replace pattern input insert) → (or/c string? bytes?)
  pattern : (or/c string? bytes? regexp? byte-regexp?)
  input : (or/c string? bytes?)
  insert : (or/c string? bytes?
                 (string? . -> . string?)
                 (bytes? . -> . bytes?))
```

Performs a match using *pattern* on *input*, and then returns a string or byte string in which the matching portion of *input* is replaced with *insert*. If *pattern* matches no part of *input*, then iput is returned unmodified.

If *pattern* is a string or character regexp and *input* is a string, then *insert* must be a string or a procedure that accept strings, and the result is a string. If *pattern* is a byte string or byte regexp, or if *input* is a byte string, then *insert* as a string is converted to a byte string, *insert* as a procedure is called with a byte string, and the result is a byte string.

If *insert* contains &, then & is replaced with the matching portion of *input* before it is substituted into the match's place. If *insert* contains \⟨*n*⟩ for some integer ⟨*n*⟩, then it is replaced with the ⟨*n*⟩th matching sub-expression from *input*. A & and \0 are synonymous. If the ⟨*n*⟩th sub-expression was not used in the match, or if ⟨*n*⟩ is greater than the number of sub-expressions in *pattern*, then \⟨*n*⟩ is replaced with the empty string.

To substitute a literal & or \, use \& and \\, respectively, in *insert*. A \$ in *insert* is equivalent to an empty sequence; this can be used to terminate a number ⟨*n*⟩ following `\`. If a `\` in *`insert`* is followed by anything other than a digit, `&`, `\`, or `$`, then the `\` by itself is treated as `\0`.

Note that the `\` described in the previous paragraphs is a character or byte of *`input`*. To write such an *`input`* as a Scheme string literal, an escaping `\` is needed before the `\`. For example, the Scheme constant `"\\1"` is `\1`.

Examples:

```
> (regexp-replace "mi" "mi casa" "su")
"su casa"
> (regexp-replace "mi" "mi casa" string-upcase)
"MI casa"
> (regexp-replace "([Mm])i ([a-zA-Z]*)" "Mi Casa" "\\1y \\2")
"My Casa"
> (regexp-replace "([Mm])i ([a-zA-Z]*)" "mi cerveza Mi Mi Mi"
                  "\\1y \\2")
"my cerveza Mi Mi Mi"
> (regexp-replace #rx"x" "12x4x6" "\\\\")
"12\\4x6"
> (display (regexp-replace #rx"x" "12x4x6" "\\\\"))
12\4x6
```

```
(regexp-replace* pattern input insert) → (or/c string? bytes?)
  pattern : (or/c string? bytes? regexp? byte-regexp?)
  input : (or/c string? bytes?)
  insert : (or/c string? bytes?
                 (string? . -> . string?)
                 (bytes? . -> . bytes?))
```

Like `regexp-replace`, except that every instance of *`pattern`* in *`input`* is replaced with *`insert`*, instead of just the first match. Only non-overlapping instances of *`pattern`* in *`input`* are replaced, so instances of *`pattern`* within inserted strings are *not* replaced recursively. If, in the process of repeating matches, *`pattern`* matches an empty string, the `exn:fail` exception is raised.

Examples:

```
> (regexp-replace* "([Mm])i ([a-zA-Z]*)" "mi cerveza Mi Mi Mi"
                   "\\1y \\2")
"my cerveza My Mi Mi"
> (regexp-replace* "([Mm])i ([a-zA-Z]*)" "mi cerveza Mi Mi Mi"
                   (lambda (all one two)
                     (string-append (string-downcase one) "y"
                                    (string-upcase two))))
"myCERVEZA myMI Mi"
> (display (regexp-replace* #rx"x" "12x4x6" "\\\\"))
```

```
12\4\6
```

```
(regexp-replace-quote str) → string?
  str : string?
(regexp-replace-quote bstr) → bytes?
  bstr : bytes?
```

Produces a string suitable for use as the third argument to `regexp-replace` to insert the literal sequence of characters in *`str`* or bytes in *`bstr`* as a replacement. Concretely, every `\` and `&` in *`str`* or *`bstr`* is protected by a quoting `\`.

Examples:

```
> (regexp-replace "UT" "Go UT!" "A&M")
"Go AUTM!"
> (regexp-replace "UT" "Go UT!" (regexp-replace-quote "A&M"))
"Go A&M!"
```

3.8 Keywords

§3.7 "Keywords" in §"**Guide**: PLT Scheme" introduces keywords.

A *keyword* is like an interned symbol, but its printed form starts with `#:`, and a keyword cannot be used as an identifier. Furthermore, a keyword by itself is not a valid expression, though a keyword can be `quote`d to form an expression that produces the symbol.

Two keywords are `eq?` if and only if they print the same.

Like symbols, keywords are only weakly held by the internal keyword table; see §3.6 "Symbols" for more information.

```
(keyword? v) → boolean?
  v : any/c
```

Returns `#t` if *`v`* is a keyword, `#f` otherwise.

```
(keyword->string keyword) → string?
  keyword : keyword?
```

Returns a string for the `display`ed form of *`keyword`*, not including the leading `#:`.

```
(string->keyword str) → keyword
  str : string?
```

Returns a keyword whose `display`ed form is the same as that of *`str`*, but with a leading

`#:`.

```
(keyword<? a-keyword b-keyword ...+) → boolean?
  a-keyword : keyword?
  b-keyword : keyword?
```

Returns `#t` if the arguments are sorted, where the comparison for each pair of keywords is the same as using `keyword->string` and `string<?`.

3.9 Pairs and Lists

A *pair* combines exactly two values. The first value is accessed with the `car` procedure, and the second value is accessed with the `cdr` procedure. Pairs are not mutable (but see §3.10 "Mutable Pairs and Lists").

A *list* is recursively defined: it is either the constant `null`, or it is a pair whose second value is a list.

A list can be used as a single-valued sequence (see §3.14 "Sequences"). The elements of the list serve as elements of the sequence. See also `in-list`.

Cyclic data structures can be created using only immutable pairs via `read` or `make-reader-graph`. If starting with a pair and using some number of `cdr`s returns to the starting pair, then the pair is not a list.

3.9.1 Pair Constructors and Selectors

```
(pair? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a pair, `#f` otherwise.

```
(null? v) → boolean?
  v : any/c
```

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

```
(cons a d) → pair?
  a : any/c
  d : any/c
```

Returns a pair whose first element is `a` and second element is `d`.

```
(car p) → any/c
  p : pair?
```

Returns the first element of the pair `p`.

```
(cdr p) → any/c
  p : pair?
```

Returns the second element of the pair `p`.

```
null : null?
```

The empty list.

```
(list? v) → boolean?
  v : any/c
```

Returns `#t` if `v` is a list: either the empty list, or a pair whose second element is a list. This procedure takes amortized constant time.

```
(list v ...) → list?
  v : any/c
```

Returns a newly allocated list containing the `v`s as its elements.

```
(list* v ... tail) → any/c
  v : any/c
  tail : any/c
```

Like `list`, but the last argument is used as the tail of the result, instead of the final element. The result is a list only if the last argument is a list.

```
(build-list n proc) → list?
  n : exact-nonnegative-integer?
  proc : (exact-nonnegative-integer? . -> . any)
```

Creates a list of `n` elements by applying `proc` to the integers from `0` to `(sub1 n)` in order. If `lst` is the resulting list, then `(list-ref lst i)` is the value produced by `(proc i)`.

Examples:

```
> (build-list 10 values)
(0 1 2 3 4 5 6 7 8 9)
> (build-list 5 (lambda (x) (* x x)))
(0 1 4 9 16)
```

3.9.2 List Operations

```
(length lst) → nonnegative-exact-integer?
  lst : list?
```

Returns the number of elements in *lst*.

```
(list-ref lst pos) → any/c
  lst : any/c
  pos : nonnegative-exact-integer?
```

Returns the element of *lst* at position *pos*, where the list's first element is position `0`. If the list has *pos* or fewer elements, then the `exn:fail:contract` exception is raised.

The *lst* argument need not actually be a list; *lst* must merely start with a chain of at least *pos* pairs.

```
(list-tail lst pos) → any/c
  lst : any/c
  pos : nonnegative-exact-integer?
```

Returns the list after the first *pos* elements of *lst*. If the list has fewer than *pos* elements, then the `exn:fail:contract` exception is raised.

The *lst* argument need not actually be a list; *lst* must merely start with a chain of at least *pos* pairs.

```
(append lst ...) → list?
  lst : list?
(append lst ... v) → any/c
  lst : list?
  v : any/c
```

When given all list arguments, the result is a lists that contains all of the elements of the given lists in order. The last argument is used directly in the tail of the result.

The last argument need not be a list, in which case the result is an "improper list."

```
(reverse lst) → list?
  lst : list?
```

Returns a list that has the same elements as *lst*, but in reverse order.

3.9.3 List Iteration

```
(map proc lst ...+) → list?
  proc : procedure?
  lst : list?
```

Applies *proc* to the elements of the *lst*s from the first elements to the last, returning `#f` as soon as any application returns `#f`. The *proc* argument must accept the same number of arguments as the number of supplied *lst*s, and all *lst*s must have the same number of elements. The result is a list containing each result of *proc*.

```
(andmap proc lst ...+) → any
  proc : procedure?
  lst : list?
```

Similar to `map`, except that

- the result is `#f` if any application of *proc* produces `#f`, in which case *proc* is not applied to later elements of the *lst*s; or
- the result is that of *proc* applied to the last elements of the `lsts`s; more specifically, the application of *proc* to the last elements in the *lst*s is in tail position with respect to the `andmap` call.

If the *lst*s are empty, then `#t` is returned.

Examples:

```
> (andmap positive? '(1 2 3))
#t
> (andmap positive? '(1 2 a))
positive?: expects argument of type <real number>; given a
> (andmap positive? '(1 -2 a))
#f
> (andmap + '(1 2 3) '(4 5 6))
9
```

```
(ormap proc lst ...+) → any
  proc : procedure?
  lst : list?
```

Similar to `map`, except that

- the result is `#f` if every application of *proc* produces `#f`; or
- the result of the first applciation of *proc* to produces a value other than `#f`, in which case *proc* is not applied to later elements of the *lst*s; more specifically, the application of *proc* to the last elements in the *lst*s is in tail position with respect to the `andmap` call.

If the *lst*s are empty, then `#f` is returned.

Examples:

```
> (ormap eq? '(a b c) '(a b c))
#t
> (ormap positive? '(1 2 a))
#t
> (ormap + '(1 2 3) '(4 5 6))
5
```

```
(for-each proc lst ...+) → void?
  proc : procedure?
  lst : list?
```

Similar to `map`, but *proc* is called only for its effect, and its result (which can be any number of values) is ignored.

```
(foldl proc init lst ...+) → list?
  proc : procedure?
  init : any/c
  lst : list?
```

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

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

Unlike `foldr`, `foldl` processes the *lst*s in constant space (plus the space for each call to *proc*).

Examples:

```
> (foldl cons '() '(1 2 3 4))
(4 3 2 1)
> (foldl + 0 '(1 2 3 4))
10
```

```
(foldr proc init lst ...+) → list?
  proc : procedure?
  init : any/c
  lst : list?
```

Like `foldl`, but the lists are traversed from right to left. Unlike `foldl`, `foldr` processes the *lst*s in space proportional to the length of *lst*s (plus the space for each call to *proc*).

Examples:

```
> (foldr cons '() '(1 2 3 4))
(1 2 3 4)
> (foldr (lambda (v l) (cons (add1 v) l)) '() '(1 2 3 4))
(2 3 4 5)
```

3.9.4 List Filtering

```
(filter pred lst) → list?
  pred : procedure?
  lst : list?
```

Returns a list with the elements of *lst* for which *pred* produces a true value. The *pred* procedure is applied to each element from first to last.

```
(remove v lst [proc]) → list?
  v : any/c
  lst : list?
  proc : procedure? = equal?
```

Returns a list that is like *lst*, omitting the first element of *lst* that is equal to *v* using the comparison procedure *proc* (which must accept two arguments).

```
(remq v lst) → list?
  v : any/c
  lst : list?
```

Returns `(remove v lst eq?)`.

```
(remv v lst) → list?
  v : any/c
  lst : list?
```

Returns `(remove v lst eqv?)`.

```
(remove* v-lst lst [proc]) → list?
  v-lst : list?
  lst : list?
  proc : procedure? = equal?
```

Like `remove`, but removes from *lst* every instance of every element of *v-lst*.

```
(remq* v lst) → list?
  v : any/c
  lst : list?
```

Returns `(remove* v lst eq?)`.

```
(remv* v lst) → list?
  v : any/c
  lst : list?
```

Returns `(remove* v lst eqv?)`.

```
(sort lst
      less-than?
     [#:key extract-key
      #:cache-keys? cache-keys?]) → list?
  lst : list?
  less-than? : (any/c any/c . -> . any/c)
  extract-key : (any/c . -> . any/c) = (lambda (x) x)
  cache-keys? : boolean? = #f
```

Returns a list sorted according to the *less-than?* procedure, which takes two elements of *lst* and returns a true value if the first is less than (i.e., should be sorted earlier) than the second.

The sort is stable; if two elements of *lst* are "equal" (i.e., proc does not return a true value when given the pair in either order), then the elements preserve their relative order from *lst* in the output list. To preserve this guarantee, use sort with a strict comparison functions (e.g., < or string<?; not <= or string<=?).

The #:key argument *extract-key* is used to extract a key value for comparison from each list element. That is, the full comparison procedure is essentially

```
(lambda (x y)
  (less-than? (extract-key x) (extract-key y)))
```

By default, *extract-key* is applied to two list elements for every comparison, but if *cache-keys?* is true, then the *extract-key* function is used exactly once for each list item. Supply a true value for *cache-keys?* when *extract-key* is an expensive operation; for example, if file-or-directory-modify-seconds is used to extract a timestamp for every file in a list, then *cache-keys?* should be #t to minimize file-system calls, but if *extract-key* is car, then *cache-keys?* should be #f. As another example, providing *extract-key* as (lambda (x) (random)) and #t for *cache-keys?* effectively shuffles the list.

Examples:

```
> (sort '(1 3 4 2) <)
(1 2 3 4)
> (sort '("aardvark" "dingo" "cow" "bear") string<?)
("aardvark" "bear" "cow" "dingo")
> (sort '(("aardvark") ("dingo") ("cow") ("bear"))
        #:key car string<?)
(("aardvark") ("bear") ("cow") ("dingo"))
```

3.9.5 List Searching

```
(member v lst) → (or/c list? false/c)
  v : any/c
  lst : list?
```

Locates the first element of *lst* that is equal? to *v*. If such an element exists, the tail of *lst* starting with that element is returned. Otherwise, the result is #f.

```
(memv v lst) → (or/c list? false/c)
  v : any/c
  lst : list?
```

Like `member`, but finds an element using `eqv?`.

```
(memq v lst) → (or/c list? false/c)
  v : any/c
  lst : list?
```

Like `member`, but finds an element using `eq?`.

```
(memf proc lst) → (or/c list? false/c)
  proc : procedure?
  lst : list?
```

Like `member`, but finds an element using the predicate *`proc`*; an element is found when *`proc`* applied to the element returns a true value.

```
(findf proc lst) → any/c
  proc : procedure?
  lst : list?
```

Like `memf`, but returns the element or `#f` instead of a tail of *`lst`* or `#f`.

```
(assoc v lst) → (or/c pair? false/c)
  v : any/c
  lst : (listof pair?)
```

Locates the first element of *`lst`* whose `car` is `equal?` to *`v`*. If such an element exists, the pair (i.e., an element of *`lst`*) is returned. Otherwise, the result is `#f`.

```
(assv v lst) → (or/c pair? false/c)
  v : any/c
  lst : (listof pair?)
```

Like `assoc`, but finds an element using `eqv?`.

```
(assq v lst) → (or/c pair? false/c)
  v : any/c
  lst : (listof pair?)
```

Like `assoc`, but finds an element using `eq?`.

```
(assf proc lst) → (or/c list? false/c)
```

```
proc : procedure?
lst : list?
```

Like `assoc`, but finds an element using the predicate *`proc`*; an element is found when *`proc`* applied to the `car` of an *`lst`* element returns a true value.

3.9.6 Pair Accessor Shorthands

```
(caar v) → any/c
  v : (cons/c pair? any/c)
```

Returns `(car (car p))`

```
(cadr v) → any/c
  v : (cons/c any/c pair?)
```

Returns `(car (cdr p))`

```
(cdar v) → any/c
  v : (cons/c pair? any/c)
```

Returns `(cdr (car p))`

```
(cddr v) → any/c
  v : (cons/c any/c pair?)
```

Returns `(cdr (cdr p))`

```
(caaar v) → any/c
  v : (cons/c (cons/c pair? any/c) any/c)
```

Returns `(car (car (car p)))`

```
(caadr v) → any/c
  v : (cons/c any/c (cons/c pair? any/c))
```

Returns `(car (car (cdr p)))`

```
(cadar v) → any/c
```

```
v : (cons/c (cons/c any/c pair?) any/c)
```

Returns `(car (cdr (car p)))`

```
(caddr v) → any/c
  v : (cons/c any/c (cons/c any/c pair?))
```

Returns `(car (cdr (cdr p)))`

```
(cdaar v) → any/c
  v : (cons/c (cons/c pair? any/c) any/c)
```

Returns `(cdr (car (car p)))`

```
(cdadr v) → any/c
  v : (cons/c any/c (cons/c pair? any/c))
```

Returns `(cdr (car (cdr p)))`

```
(cddar v) → any/c
  v : (cons/c (cons/c any/c pair?) any/c)
```

Returns `(cdr (cdr (car p)))`

```
(cdddr v) → any/c
  v : (cons/c any/c (cons/c any/c pair?))
```

Returns `(cdr (cdr (cdr p)))`

```
(caaaar v) → any/c
  v : (cons/c (cons/c (cons/c pair? any/c) any/c) any/c)
```

Returns `(car (car (car (car p))))`

```
(caaadr v) → any/c
  v : (cons/c any/c (cons/c (cons/c pair? any/c) any/c))
```

Returns `(car (car (car (cdr p))))`

```
(caadar v) → any/c
```

```
  v : (cons/c (cons/c any/c (cons/c pair? any/c)) any/c)
```

Returns `(car (car (cdr (car p))))`

```
(caaddr v) → any/c
  v : (cons/c any/c (cons/c any/c (cons/c pair? any/c)))
```

Returns `(car (car (cdr (cdr p))))`

```
(cadaar v) → any/c
  v : (cons/c (cons/c (cons/c any/c pair?) any/c) any/c)
```

Returns `(car (cdr (car (car p))))`

```
(cadadr v) → any/c
  v : (cons/c any/c (cons/c (cons/c any/c pair?) any/c))
```

Returns `(car (cdr (car (cdr p))))`

```
(caddar v) → any/c
  v : (cons/c (cons/c any/c (cons/c any/c pair?)) any/c)
```

Returns `(car (cdr (cdr (car p))))`

```
(cadddr v) → any/c
  v : (cons/c any/c (cons/c any/c (cons/c any/c pair?)))
```

Returns `(car (cdr (cdr (cdr p))))`

```
(cdaaar v) → any/c
  v : (cons/c (cons/c (cons/c pair? any/c) any/c) any/c)
```

Returns `(cdr (car (car (car p))))`

```
(cdaadr v) → any/c
  v : (cons/c any/c (cons/c (cons/c pair? any/c) any/c))
```

Returns `(cdr (car (car (cdr p))))`

```
(cdadar v) → any/c
```

```
v : (cons/c (cons/c any/c (cons/c pair? any/c)) any/c)
```

Returns `(cdr (car (cdr (car p))))`

```
(cdaddr v) → any/c
  v : (cons/c any/c (cons/c any/c (cons/c pair? any/c)))
```

Returns `(cdr (car (cdr (cdr p))))`

```
(cddaar v) → any/c
  v : (cons/c (cons/c (cons/c any/c pair?) any/c) any/c)
```

Returns `(cdr (cdr (car (car p))))`

```
(cddadr v) → any/c
  v : (cons/c any/c (cons/c (cons/c any/c pair?) any/c))
```

Returns `(cdr (cdr (car (cdr p))))`

```
(cdddar v) → any/c
  v : (cons/c (cons/c any/c (cons/c any/c pair?)) any/c)
```

Returns `(cdr (cdr (cdr (car p))))`

```
(cddddr v) → any/c
  v : (cons/c any/c (cons/c any/c (cons/c any/c pair?)))
```

Returns `(cdr (cdr (cdr (cdr p))))`

3.9.7 Additional List Functions and Synonyms

```
(require scheme/list)
```

The bindings documented in this section are provided by the `scheme/list` and `scheme` libraries, but not `scheme/base`.

```
empty : null?
```

The empty list.

```
(cons? v) → boolean?
  v : any/c
```

The same as `(pair? v)`.

```
(empty? v) → boolean?
  v : any/c
```

The same as `(null? v)`.

```
(first lst) → any/c
  lst : list?
```

The same as `(car lst)`, but only for lists (that are not empty).

```
(rest lst) → list?
  lst : list?
```

The same as `(cdr lst)`, but only for lists (that are not empty).

```
(second lst) → any
  lst : list?
```

Returns the second element of the list.

```
(third lst) → any
  lst : list?
```

Returns the third element of the list.

```
(fourth lst) → any
  lst : list?
```

Returns the fourth element of the list.

```
(fifth lst) → any
  lst : list?
```

Returns the fifth element of the list.

```
(sixth lst) → any
  lst : list?
```

Returns the sixth element of the list.

```
(seventh lst) → any
  lst : list?
```

Returns the seventh element of the list.

```
(eighth lst) → any
  lst : list?
```

Returns the eighth element of the list.

```
(ninth lst) → any
  lst : list?
```

Returns the ninth element of the list.

```
(tenth lst) → any
  lst : list?
```

Returns the tenth element of the list.

```
(last lst) → any
  lst : list?
```

Returns the last element of the list.

```
(last-pair p) → pair?
  p : pair?
```

Returns the last pair of a (possibly improper) list.

```
(drop lst pos) → list?
  lst : any/c
  pos : nonnegative-exact-integer?
```

Synonym for list-tail.

```
(take lst pos) → list?
  lst : any/c
  pos : nonnegative-exact-integer?
```

Returns a fresh list whose elements are the first *pos* elements of *lst*. If *lst* has fewer than *pos* elements, the exn:fail:contract exception is raised.

The *lst* argument need not actually be a list; *lst* must merely start with a chain of at least *pos* pairs.

Examples:

```
> (take '(1 2 3 4) 2)
(1 2)
> (take 'non-list 0)
()
```

```
(add-between lst v) → list?
  lst : list?
  v : any/c
```

Returns a list with the same elements as *lst*, but with *v* between each pair of items in *lst*.

Examples:

```
> (add-between '(x y z) 'or)
(x or y or z)
> (add-between '(x) 'or)
(x)
```

```
(append* lst ... lsts) → list?
  lst : list?
  lsts : (listof list?)
(append* lst ... lsts) → any/c
  lst : list?
  lsts : list?
```

Like append, but the last argument is used as a list of arguments for append, so (append* lst ... lsts) is the same as (apply append lst ... lsts). In other words, the relationship between append and append* is similar to the one between list and list*.

Examples:

```
> (append* '(a) '(b) '((c) (d)))
(a b c d)
```

```
> (cdr (append* (map (lambda (x) (list ", " x))
                     '("Alpha" "Beta" "Gamma"))))
("Alpha" ", " "Beta" ", " "Gamma")
```

```
(flatten v) → list?
  v : any/c
```

Flattens an arbitrary S-expression structure of pairs into a list. More precisely, `v` is treated as a binary tree where pairs are interior nodes, and the resulting list contains all of the non-`null` leaves of the tree in the same order as an inorder traversal.

Examples:

```
> (flatten '((a) b (c (d) . e) ()))
(a b c d e)
> (flatten 'a)
(a)
```

```
(remove-duplicates lst [same?]) → list?
  lst : list?
  same? : (any/c any/c . -> . any/c) = equal?
```

Returns a list that has all items in `lst`, but without duplicate items, where `same?` determines whether two elements of the list are equivalent. The resulting list is in the same order as `lst`, and for any item that occurs multiple times, the first one is kept.

Examples:

```
> (remove-duplicates '(a b b a))
(a b)
> (remove-duplicates '(1 2 1.0 0))
(1 2 1.0 0)
> (remove-duplicates '(1 2 1.0 0) =)
(1 2 0)
```

```
(filter-map proc lst ...+) → list?
  proc : procedure?
  lst : list?
```

Returns `(filter (lambda (x) x) (map proc lst ...))`, but without building the intermediate list.

```
(partition pred lst) → list? list?
  pred : procedure?
  lst : list?
```

Similar to `filter`, except that two values are returned: the items for which *pred* returns a true value, and the items for which *pred* returns `#f`.

The result is the same as

```
(values (filter pred lst) (filter (negate pred) lst))
```

but *pred* is applied to each item in *lst* only once.

Examples:

```
> (partition even? '(1 2 3 4 5 6))
(2 4 6)
(1 3 5)
```

```
(append-map proc lst ...+) → list?
  proc : procedure?
  lst : list?
```

Returns `(append* (map proc lst ...))`.

Examples:

```
> (append-map vector->list '(#(1) #(2 3) #(4)))
(1 2 3 4)
```

```
(filter-not pred lst) → list?
  pred : (any/c . -> . any/c)
  lst : list?
```

Like `filter`, but the meaning of the *pred* predicate is reversed: the result is a list of all items for which *pred* returns `#f`.

Examples:

```
> (filter-not even? '(1 2 3 4 5 6))
(1 3 5)
```

3.9.8 Immutable Cyclic Data

```
(make-reader-graph v) → any/c
  v : any/c
```

Returns a value like *v*, with placeholders created by `make-placeholder` replaced with the values that they contain, and with placeholders created by `make-hash-placeholder` with an immutable hash table. No part of *v* is mutated; instead, parts of *v* are copied as necessary to construct the resulting graph, where at most one copy is created for any given value.

Since the copied vales can be immutable, and since the copy is also immutable, `make-reader-graph` can cycles involving only immutable pairs, vectors, boxes, and hash tables.

Only the following kinds of values are copied and traversed to detect placeholders:

- pairs
- immutable pairs (as created by `mcons`)
- vectors, both mutable and immutable
- boxes, both mutable and immutable
- hash tables, both mutable and immutable
- instances of a prefab structure type
- placeholders created by `make-placeholder` and `make-hash-placeholder`

Due to these restrictions, `make-reader-graph` creates exactly the same sort of cyclic values as `read`.

Examples:

```
> (let* ([ph (make-placeholder #f)]
         [x (cons 1 ph)])
    (placeholder-set! ph x)
    (make-reader-graph x))
#0=(1 . #0#)
```

```
(placeholder? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a placeholder created by `make-placeholder`, `#f` otherwise.

```
(make-placeholder v) → placeholder?
  v : any/c
```

Returns a placeholder for use with `placeholder-set!` and `make-reader-graph`. The *v* argument supplies the initial value for the placeholder.

```
(placeholder-set! ph datum) → void?
  ph : placeholder?
  datum : any/c
```

Changes the value of *ph* to `v`.

`(placeholder-get ph) → any/c`
  `ph : placeholder?`

Returns the value of *ph*.

`(hash-placeholder? v) → boolean?`
  `v : any/c`

Returns `#t` if *v* is a placeholder created by `make-hash-placeholder`, `#f` otherwise.

`(make-hash-placeholder assocs) → hash-placeholder?`
  `assocs : (listof pair?)`

Like `make-immutable-hash`, but produces a table placeholder for use with `make-reader-graph`.

`(make-hasheq-placeholder assocs) → hash-placeholder?`
  `assocs : (listof pair?)`

Like `make-immutable-hasheq`, but produces a table placeholder for use with `make-reader-graph`.

3.10 Mutable Pairs and Lists

A *mutable pair* is like a pair created by `cons`, but it supports `set-mcar!` and `set-mcdr!` mutation operations to change the parts of the pair (like traditional Lisp and Scheme pairs).

A *mutable list* is analogous to a list created with pairs, but instead created with mutable pairs.

3.10.1 Mutable Pair Constructors and Selectors

`(mpair? v) → boolean?`
  `v : any/c`

Returns `#t` if *v* is a mutable pair, `#f` otherwise.

```
(mcons a d) → pair?
  a : any/c
  d : any/c
```

Returns a mutable pair whose first element is `a` and second element is `d`.

```
(mcar p) → any/c
  p : mpair?
```

Returns the first element of the mutable pair `p`.

```
(mcdr p) → any/c
  p : mpair?
```

Returns the second element of the mutable pair `p`.

```
(set-mcar! p v) → void?
  p : mpair?
  v : any/v
```

Changes the mutable pair `p` so that its first element is `v`.

```
(set-mcdr! p v) → void?
  p : mpair?
  v : any/v
```

Changes the mutable pair `p` so that its second element is `v`.

3.10.2 Mutable List Functions

```
(require scheme/mpair)
```

The bindings documented in this section are provided by the `scheme/mpair` library, not `scheme/base` or `scheme`.

For functions described in this section, contracts are not directly enforced. In particular, when a mutable list is expected, supplying any other kind of value (or mutating a value that starts as a list) tends to produce an exception from `mcar` or `mcdr`.

```
(mlist? v) → boolean?
  v : any/c
```

Returns #t if *v* is a mutable list: either the empty list, or a mutable pair whose second element is a mutable list.

```
(mlist v ...) → mlist?
  v : any/c
```

Returns a newly allocated mutable list containing the *v*s as its elements.

```
(list->mlist lst) → mlist?
  lst : list?
```

Returns a newly allocated mutable list with the same elements as *lst*.

```
(mlist->list mlst) → list?
  mlst : mlist?
```

Returns a newly allocated list with the same elements as nlst.

```
(mlength mlst) → nonnegative-exact-integer?
  mlst : mlist?
```

Returns the number of elements in *mlst*.

```
(mlist-ref mlst pos) → any/c
  mlst : mlist?
  pos : nonnegative-exact-integer?
```

Like list-ref, but for mutable lists.

```
(mlist-tail mlst pos) → any/c
  mlst : mlist?
  pos : nonnegative-exact-integer?
```

Like list-tail, but for mutable lists.

```
(mappend mlst ...) → mlist?
  mlst : mlist?
(mappend mlst ... v) → any/c
  mlst : mlist?
  v : any/c
```

Like `append`, but for mutable lists.

`(mappend! mlst ...) → mlist?`
  `mlst : mlist?`
`(mappend! mlst ... v) → any/c`
  `mlst : mlist?`
  `v : any/c`

The `mappend!` procedure appends the given lists by mutating the tail of each to refer to the next, using `set-mcdr!`. Empty lists are dropped; in particular, the result of calling `mappend!` with one or more empty lists is the same as the result of the call with the empty lists removed from the set of arguments.

`(mreverse mlst) → mlist?`
  `mlst : mlist?`

Like `reverse`, but for mutable lists.

`(mreverse! mlst) → mlist?`
  `mlst : mlist?`

Like `mreverse`, but destructively reverses the list by using all of the mutable pairs in *`mlst`* and changing them with `set-mcdr!`.

`(mmap proc mlst ...+) → mlist?`
  `proc : procedure?`
  `mlst : mlist?`

Like `map`, but for mutable lists.

`(mfor-each proc mlst ...+) → void?`
  `proc : procedure?`
  `mlst : mlist?`

Like `for-each`, but for mutable lists.

`(mmember v mlst) → (or/c mlist? false/c)`
  `v : any/c`
  `mlst : mlist?`

Like `member`, but for mutable lists.

```
(mmemv v mlst) → (or/c mlist? false/c)
  v : any/c
  mlst : mlist?
```

Like `memv`, but for mutable lists.

```
(mmemq v mlst) → (or/c list? false/c)
  v : any/c
  mlst : mlist?
```

Like `memq`, but for mutable lists.

```
(massoc v mlst) → (or/c mpair? false/c)
  v : any/c
  mlst : (mlistof mpair?)
```

Like `assoc`, but for mutable lists of mutable pairs.

```
(massv v mlst) → (or/c mpair? false/c)
  v : any/c
  mlst : (mlistof mpair?)
```

Like `assv`, but for mutable lists of mutable pairs.

```
(massq v mlst) → (or/c mpair? false/c)
  v : any/c
  mlst : (mlistof mpair?)
```

Like `assq`, but for mutable lists of mutable pairs.

```
(mlistof pred) → (any/c . -> . boolean?)
  pred : (any/c . -> . any/c)
```

Returns a procedure that returns `#t` when given a mutable list for which *`pred`* returns a true value for all elements.

3.11 Vectors

A *vector* is a fixed-length array with constant-time access and update of the vector slots, which are numbered from `0` to one less than the number of slots in the vector.

Two vectors are `equal?` if they have the same length, and if the values in corresponding slots of the the vectors are `equal?`.

A vector can be *mutable* or *immutable*. When an immutable vector is provided to a procedure like `vector-set!`, the `exn:fail:contract` exception is raised. Vectors generated by the default reader (see §12.6.6 "Reading Strings") are immutable.

A vector can be used as a single-valued sequence (see §3.14 "Sequences"). The elements of the vector serve as elements of the sequence. See also `in-vector`.

```
(vector? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a vector, `#f` otherwise.

```
(make-vector size [v]) → vector?
  size : nonnegative-exact-integer?
  v : any/c = 0
```

Returns a mutable vector with *size* slots, where all slots are initialized to contain *v*.

```
(vector v ...) → vector?
  v : any/c
```

Returns a mutable vector with as many slots as provided *v*s, where the slots are initialized to contain the given *v*s in order.

```
(vector-immutable v ...) → (and/c vector?
                                  immutable?)
  v : any/c
```

Returns an immutable vector with as many slots as provided *v*s, where the slots are contain the given *v*s in order.

```
(vector-length vec) → nonnegative-exact-integer?
  vec : vector?
```

Returns the length of `vec` (i.e., the number of slots in the vector).

```
(vector-ref vec pos) → any/c
  vec : vector?
  pos : nonnegative-exact-integer?
```

Returns the element in slot `pos` of `vec`. The first slot is position `0`, and the last slot is one less than `(vector-length vec)`.

```
(vector-set! vec pos v) → void?
  vec : (and/c vector? (not/c immutable?))
  pos : nonnegative-exact-integer?
  v : any/c
```

Updates the slot `pos` of `vec` to contain `v`.

```
(vector->list vec) → list?
  vec : vector?
```

Returns a list with the same length and elements as `vec`.

```
(list->vector lst) → vector?
  lst : list?
```

Returns a mutable vector with the same length and elements as `lst`.

```
(vector->immutable-vector vec) → (and/c vector? immutable?)
  vec : vector?
```

Returns an immutable vector with the same length and elements as `vec`. If `vec` is itself immutable, then it is returned as the result.

```
(vector-fill! vec v) → void?
  vec : (and/c vector? (not/c immutable?))
  v : any/c
```

Changes all slots of `vec` to contain `v`.

```
(vector-copy! dest
              dest-start
              src
             [src-start
              src-end])   → void?
  dest : (and/c vector? (not/c immutable?))
  dest-start : exact-nonnegative-integer?
  src : vector?
  src-start : exact-nonnegative-integer? = 0
  src-end : exact-nonnegative-integer? = (vector-length src)
```

Changes the elements of `dest` starting at position `dest-start` to match the elements in `src` from `src-start` (inclusive) to `src-end` (exclusive). The vectors `dest` and `src` can be the same vector, and in that case the destination region can overlap with the source region; the destination elements after the copy match the source elements from before the copy. If any of `dest-start`, `src-start`, or `src-end` are out of range (taking into account the sizes of the vectors and the source and destination regions), the `exn:fail:contract` exception is raised.

Examples:

```
> (define v (vector 'A 'p 'p 'l 'e))
> (vector-copy! v 4 #(y))
> (vector-copy! v 0 v 3 4)
> v
#(l p p l y)
```

```
(vector->values vec [start-pos end-pos]) → any
  vec : vector?
  start-pos : nonnegative-exact-integer? = 0
  end-pos : nonnegative-exact-integer? = (vector-length vec)
```

Returns `end-pos` - `start-pos` values, which are the elements of `vec` from `start-pos` (inclusive) to `end-pos` (exclusive). If `start-pos` or `end-pos` are greater than `(vector-length vec)`, or if `end-pos` is less than `start-pos`, the `exn:fail:contract` exception is raised.

```
(build-vector n proc) → vector?
  n : exact-nonnegative-integer?
  proc : (exact-nonnegative-integer? . -> . any.c)
```

Creates a vector of `n` elements by applying `proc` to the integers from `0` to `(sub1 n)` in order. If `vec` is the resulting vector, then `(vector-ref vec i)` is the value produced by `(proc i)`.

Examples:

```
> (build-vector 5 add1)
#(1 2 3 4 5)
```

3.12 Boxes

```
(box? v) → boolean?
  v : any/c
```

Returns `#t` if `v` is a box, `#f` otherwise.

```
(box v) → box?
  v : any/c
```

Returns a new mutable box that contains `v`.

```
(box-immutable v) → (and/c box? immutable?)
  v : any/c
```

Returns a new immutable box that contains `v`.

```
(unbox box) → any/c
  box : box?
```

Returns the content of `box`. For any `v`, `(unbox (box (scheme v)))` returns `v`.

```
(set-box! box v) → void?
  box : (and/c box? (not/c immutable?))
  v : any/c
```

Sets the content of `box` to `v`.

3.13 Hash Tables

A *hash table* (or simply *hash*) maps each of its keys to a single value. For a given hash table, keys are equivalent via `equal?` or `eq?`, and keys are retained either strongly or weakly (see §15.1 "Weak Boxes"). A hash table is also either mutable or immutable; immutable tables support constant-time functional update.

A hash table can be used as a two-valued sequence (see §3.14 "Sequences"). The keys and values of the hash table serve as elements of the sequence (i.e., each element is a key and its associated value). If a mapping is added to or removed from the hash table during iteration, then an iteration step may fail with `exn:fail:contract`, or the iteration may skip or duplicate keys and values. See also `in-hash`, `in-hash-keys`, `in-hash-values`, and `in-hash-pairs`.

Two hash tables are `equal?` if they use the same key-comparison procedure (`equal?` or `eq?`), if they both hold keys strongly or weakly, and if they have the same mutability.

Caveats concerning concurrent modification: A mutable hash table can be manipulated with `hash-ref`, `hash-set!`, and `hash-remove!` concurrently by multiple threads, and the operations are protected by a table-specific semaphore as needed. Two caveats apply, however:

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

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

```
(hash? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a hash table, `#f` otherwise.

```
(hash-eq? hash) → boolean?
  hash : hash?
```

Returns `#t` if *hash* compares keys with `eq?`, `#f` if it compares with `equal?`.

```
(hash-weak? hash) → boolean?
  hash : hash?
```

Returns `#t` if *hash* retains its keys weakly, `#f` if it retains keys strongly.

```
(make-hash) → hash?
```

Creates an empty mutable hash table that holds keys strongly and that uses `equal?` to compare keys. See also `make-custom-hash`.

```
(make-hasheq) → (and/c hash? hash-eq?)
```

Creates an empty mutable hash table that holds keys strongly and that uses `eq?` to compare keys.

```
(make-weak-hash) → (and/c hash? hash-weak?)
```

Creates an empty mutable hash table that holds keys weakly and that uses `equal?` to compare keys. See also `make-weak-custom-hash`.

```
(make-weak-hasheq) → (and/c hash? hash-eq? hash-weak?)
```

Creates an empty mutable hash table that holds keys weakly and that uses `eq?` to compare keys.

```
(make-immutable-hash assocs) → (and/c hash? immutable?)
  assocs : (listof pair?)
```

Creates an immutable hash table that compares keys with `equal?`. In each element of *assocs*, the `car` of each pair is a key, and the `cdr` is the corresponding value. The mappings are added to the table in the order that they appear in *assocs*, so later mappings can hide earlier mappings.

```
(make-immutable-hasheq assocs)
 → (and/c hash? hash-eq? immutable?)
  assocs : (listof pair?)
```

Like `make-immutable-hash`, but the resulting hash table compares keys with `eq?`.

```
(hash-set! hash key v) → void?
  hash : (and/c hash? (not/c immutable?))
  key : any/c
  v : any/c
```

Maps *key* to *v* in *hash*, overwriting any existing mapping for *key*.

```
(hash-set hash key v) → (and/c hash? immutable?)
  hash : (and/c hash? immutable?)
  key : any/c
  v : any/c
```

Functionally extends *hash* by mapping *key* to *v*, overwriting any existing mapping for *key*, and returning an extended hash table.

```
(hash-ref hash key [failure-result]) → any
  hash : hash?
  key : any/c
  failure-result : any/c
                 = (lambda () (raise (make-exn:fail ....)))
```

Returns the value for *key* in *hash*. If no value is found for *key*, then *failure-result* determines the result:

- If *failure-result* is a procedure, it is called (through a tail call) with no arguments to produce the result.
- Otherwise, *failure-result* is returned as the result.

```
(hash-remove! hash key) → void?
  hash : (and/c hash? (not/c immutable?))
  key : any/c
```

Removes any existing mapping for *key* in *hash*.

```
(hash-remove hash key) → (and/c hash? immutable?)
  hash : (and/c hash? immutable?)
  key : any/c
```

Functionally removes any existing mapping for *key* in *hash*, returning the updated hash table.

```
(hash-map hash proc) → (listof any/c)
  hash : hash?
  proc : (any/c any/c . -> . any/c)
```

Applies the procedure *proc* to each element in *hash* in an unspecified order, accumulating the results into a list. The procedure *proc* is called each time with a key and its value. See the caveat above about concurrent modification.

```
(hash-for-each hash proc) → void?
  hash : hash?
  proc : (any/c any/c . -> . any)
```

Applies *proc* to each element in *hash* (for the side-effects of *proc*) in an unspecified order. The procedure *proc* is called each time with a key and its value. See the caveat above about concurrent modification.

```
(hash-count hash) → nonnegative-exact-integer?
  hash : hash?
```

Returns the number of keys mapped by *hash*. If *hash* is not created with `'weak`, then the result is computed in constant time and atomically. If *hash* is created with `'weak`, see the caveat above about concurrent modification.

```
(hash-iterate-first hash)
 → (or/c false/c nonnegative-exact-integer?)
  hash : hash?
```

Returns `#f` if *hash* contains no elements, otherwise it returns an integer that is a index to the first element in the hash table; "first" refers to an unspecified ordering of the table elements, and the index values are not necessarily consecutive integers. For a mutable *hash*, this index is guaranteed to refer to the first item only as long as no items are added to or removed from *hash*.

```
(hash-iterate-next hash pos)
 → (or/c false/c nonnegative-exact-integer?)
  hash : hash?
  pos : nonnegative-exact-integer?
```

Returns either an integer that is an index to the element in *hash* after the element indexed by *pos* (which is not necessarily one more than *pos*) or `#f` if *pos* refers to the last element in *hash*. If *pos* is not a valid index, then the `exn:fail:contract` exception is raised. For a mutable *hash*, the result index is guaranteed to refer to its item only as long as no items are added to or removed from *hash*.

```
(hash-iterate-key hash pos) → any
  hash : hash?
```

`pos : nonnegative-exact-integer?`

Returns the key for the element in *hash* at index *pos*. If *pos* is not a valid index for *hash*, the `exn:fail:contract` exception is raised.

```
(hash-iterate-value hash pos) → any
  hash : hash?
  pos : nonnegative-exact-integer?
```

Returns the value for the element in *hash* at index *pos*. If *pos* is not a valid index for *hash*, the `exn:fail:contract` exception is raised.

```
(hash-copy hash) → (and/c hash? (not/c immutable?))
  hash : hash?
```

Returns a mutable hash table with the same mappings, same key-comparison mode, and same key-holding strength as *hash*.

```
(eq-hash-code v) → exact-integer?
  v : any/c
```

Returns an exact integer; for any two `eq?` values, the returned integer is the same. Furthermore, for the result integer `k` and any other exact integer `j`, `(= k j)` implies `(eq? k j)`.

```
(equal-hash-code v) → exact-integer?
  v : any/c
```

Returns an exact integer; for any two `equal?` values, the returned integer is the same. Furthermore, for the result integer `k` and any other exact integer `j`, `(= k j)` implies `(eq? k j)`. A has code is computed even when *v* contains a cycle through pairs, vectors, boxes, and/or inspectable structure fields. See also `prop:equal+hash`.

```
(equal-secondary-hash-code v) → exact-integer?
  v : any/c
```

Like `equal-hash-code`, but computes a secondary value suitable for use in double hashing.

3.14 Sequences

§11.1 "Sequence Constructors" in §"**Guide**: PLT Scheme" introduces sequences.

A *sequence* encapsulates an ordered stream of values. The elements of a sequence can be extracted with one of the `for` syntactic forms or with the procedures returned by `sequence-generate`.

The sequence datatype overlaps with many other datatypes. Among built-in datatypes, the sequence datatype includes the following:

- lists (see §3.9 "Pairs and Lists")
- vectors (see §3.11 "Vectors")
- hash tables (see §3.13 "Hash Tables")
- strings (see §3.3 "Strings")
- byte strings (see §3.4 "Byte Strings")
- input ports (see §12.1 "Ports")

In addition, `make-do-sequence` creates a sequence given a thunk that returns procedures to implement a generator, and the `prop:sequence` property can be associated with a structure type.

For most sequence types, extracting elements from a sequence has no side-effect on the original sequence value; for example, extracting the sequence of elements from a list does not change the list. For other sequence types, each extraction implies a side effect; for example, extracting the sequence of bytes from a port cause the bytes to be read from the port.

Inidvidual elements of a sequence typically correspond to single values, but an element may also correspond to multiple values. For example, a hash table generates two values—a key and its value—for each element in the sequence.

3.14.1 Sequence Predicate and Constructors

```
(sequence? v) → boolean?
  v : any/c
```

Return `#t` if *v* can be used as a sequence, `#f` otherwise.

```
(in-range end) → sequence?
```

```
  end : number?
(in-range start end [step]) → sequence?
  start : number?
  end : number?
  step : number? = 1
```

Returns a sequence whose elements are numbers. The single-argument case `(in-range end)` is equivalent to `(in-range 0 end 1)`. The first number in the sequence is `start`, and each successive element is generated by adding `step` to the previous element. The sequence stops before an element that would be greater or equal to `end` if `step` is non-negative, or less or equal to `end` if `step` is negative.

A `in-range` application can provide better performance for number iteration when it appears directly in a `for` clause.

```
(in-naturals [start]) → sequence?
  start : exact-nonnegative-integer? = 0
```

Returns an infinite sequence of exact integers starting with `start`, where each element is one more than the preceeding element.

A `in-naturals` application can provide better performance for integer iteration when it appears directly in a `for` clause.

```
(in-list lst) → sequence?
  lst : list?
```

Returns a sequence equivalent to `lst`.

A `in-list` application can provide better performance for list iteration when it appears directly in a `for` clause.

```
(in-vector vec) → sequence?
  vec : vector?
```

Returns a sequence equivalent to `vec`.

A `in-vector` application can provide better performance for vector iteration when it appears directly in a `for` clause.

```
(in-string str) → sequence?
  str : string?
```

Returns a sequence equivalent to `str`.

A `in-string` application can provide better performance for string iteration when it appears directly in a `for` clause.

```
(in-bytes bstr) → sequence?
  bstr : bytes?
```

Returns a sequence equivalent to `bstr`.

A `in-bytes` application can provide better performance for byte string iteration when it appears directly in a `for` clause.

```
(in-input-port-bytes in) → sequence?
  in : input-port?
```

Returns a sequence equivalent to `in`.

```
(in-input-port-chars in) → sequence?
  in : input-port?
```

Returns a sequence whose elements are read as characters form `in` (as opposed to using `in` directly as a sequence to get bytes).

```
(in-lines [in mode]) → sequence?
  in : input-port? = (current-input-port)
  mode : (one-of 'linefeed 'return 'return-linefeed 'any 'any-one)
       = 'any
```

Returns a sequence whose elements are the result of `(read-line in mode)` until an end-of-line is encountered. Note that the default mode is `'any`, whereas the default mode of `read-line` is `'linefeed`.

```
(in-hash hash) → sequence?
  hash : hash?
```

Returns a sequence equivalent to `hash`.

```
(in-hash-keys hash) → sequence?
  hash : hash?
```

Returns a sequence whose elements are the keys of *hash*.

`(in-hash-values hash) → sequence?`
`  hash : hash?`

Returns a sequence whose elements are the values of *hash*.

`(in-hash-pairs hash) → sequence?`
`  hash : hash?`

Returns a sequence whose elements are pairs, each containing a key and its value from *hash* (as opposed to using *hash* directly as a sequence to get the key and value as separate values for each element).

`(in-indexed seq) → sequence?`
`  seq : sequence?`

Returns a sequence where each element has two values: the value produced by *seq*, and a non-negative exact integer starting with `0`. The elements of *seq* must be single-valued.

`(in-parallel seq ...) → sequence?`
`  seq : sequence?`

Returns a sequence where each element has as many values as the number of supplied *seq*s; the values, in order, are the values of each *seq*. The elements of each *seq* must be single-valued.

```
(stop-before seq pred) → sequence?
  seq : sequence?
  pred : (any/c . -> . any)
```

Returns a sequence that contains the elements of *seq* (which must be single-valued), but only until the last element for which applying *pred* to the element produces `#t`, after which the sequence ends.

```
(stop-after seq pred) → sequence?
  seq : sequence?
  pred : (any/c . -> . any)
```

Returns a sequence that contains the elements of *seq* (which must be single-valued), but only until the element (inclusive) for which applying *pred* to the element produces `#t`, after which the sequence ends.

```
(make-do-sequence thunk) → sequence?
  thunk : (->* ()
               (values (any/c . -> . any)
                       (any/c . -> . any/c)
                       any/c
                       (any/c . -> . any/c)
                       (() () #:rest list? . ->* . any/c)
                       ((any/c) () #:rest list? . ->* . any/c)))
```

Returns a sequence whose elements are generated by the procedures and initial value returned by the thunk. The generator is defined in terms of a *position*, which is initialized to the third result of the thunk, and the *element*, which may consist of multiple values.

The `thunk` results define the generated elements as follows:

- The first result is a `pos->element` procedure that takes the current position and returns the value(s) for the current element.
- The second result is a `next-pos` procedure that takes the current position and returns the next position.
- The third result is the initial position.
- The fourth result takes the current position and returns a true result if the sequence includes the value(s) for the current position, and false if the sequence should end instead of including the value(s).
- The fifth result is like the fourth result, but it takes the current element value(s) instead of the current position.
- The sixth result is like the fourth result, but it takes both the current position and the current element values(s) and determines a sequence end after the current element is already included in the sequence.

Each of the procedures listed above is called only once per position. Among the last three procedures, as soon as one of the procedures returns `#f`, the sequence ends, and none are called again. Typically, one of the functions determines the end condition, and the other two functions always return `#t`.

```
prop:sequence : struct-type-property?
```

Associates a procedure to a structure type that takes an instance of the structure and returns a sequence. If `v` is an instance of a structure type with this property, then `(sequence? v)` produces `#t`.

Examples:

```
> (define-struct train (car next)
    #:property prop:sequence (lambda (t)
                               (make-do-sequence
                                (lambda ()
                                  (values train-car
                                          train-next
                                          t
                                          (lambda (t) t)
                                          (lambda (v) #t)
                                          (lambda (t v) #t))))))
> (for/list ([c (make-train 'engine
                            (make-train 'boxcar
                                        (make-train 'caboose
                                                    #f)))])
    c)
'(engine boxcar caboose)
```

3.14.2 Sequence Generators

`(sequence-generate seq) → (-> boolean?) (-> any)`
  `seq : sequence?`

Returns two thunks to extract elements from the sequence. The first returns `#t` if more values are available for the sequence. The second returns the next element (which may be multiple values) from the sequence; if no more elements are available, the `exn:fail:contract` exception is raised.

3.15 Dictionaries

A *dictionary* is an instance of a datatype that maps keys to values. The following datatypes are all dictionaries:

- hash tables;
- vectors (using only exact integers as keys);
- lists of pairs (an *association list* using `equal?` to compare keys); and
- structures whose types have the `prop:dict` property.

```
(require scheme/dict)
```

The bindings documented in this section are provided by the `scheme/dict` and `scheme` libraries, but not `scheme/base`.

```
(dict? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a dictionary, `#f` otherwise.

Beware that `dict?` is not a constant-time test on pairs, since checking that *v* is an association list may require traversing the list.

Examples:

```
> (dict? #hash((a . "apple")))
#t
> (dict? '#("apple" "banana"))
#t
> (dict? '("apple" "banana"))
#f
> (dict? '((a . "apple") (b . "banana")))
#t
```

```
(dict-mutable? d) → boolean?
  d : dict?
```

Returns `#t` if *d* is mutable via `dict-set!` and maybe `dict-remove!`, `#f` otherwise.

Examples:

```
> (dict-mutable? #hash((a . "apple")))
#f
> (dict-mutable? (make-hash))
#t
> (dict-mutable? '#("apple" "banana"))
#f
> (dict-mutable? (vector "apple" "banana"))
#t
> (dict-mutable? '((a . "apple") (b . "banana")))
#f
```

```
(dict-can-remove-keys? d) → boolean?
  d : dict?
```

Returns #t if *d* supports removing mappings via dict-remove! and/or dict-remove, #f otherwise.

Examples:

```
> (dict-can-remove-keys? #hash((a . "apple")))
#t
> (dict-can-remove-keys? '#("apple" "banana"))
#f
> (dict-can-remove-keys? '((a . "apple") (b . "banana")))
#t
```

```
(dict-can-functional-set? d) → boolean?
  d : dict?
```

Returns #t if *d* supports functional update via dict-set and maybe dict-remove, #f otherwise.

Examples:

```
> (dict-can-functional-set? #hash((a . "apple")))
#t
> (dict-can-functional-set? (make-hash))
#f
> (dict-can-functional-set? '#("apple" "banana"))
#f
> (dict-can-functional-set? '((a . "apple") (b . "banana")))
#t
```

```
(dict-set! dict key v) → void?
  dict : (and/c dict? (not/c immutable?))
  key : any/c
  v : any/c
```

Maps *key* to *v* in *dict*, overwriting any existing mapping for *key*. The update can fail with a exn:fail:contract exception if *dict* is not mutable or if *key* is not an allowed key for the dictionary (e.g., not an exact integer in the appropriate range when *dict* is a vector).

Examples:

```
> (define h (make-hash))
> (dict-set! h 'a "apple")
> h
#hash((a . "apple"))
> (define v (vector #f #f #f))
> (dict-set! v 0 "apple")
> v
#("apple" #f #f)
```

```
(dict-set dict key v) → (and/c dict? immutable?)
  dict : (and/c dict? immutable?)
  key : any/c
  v : any/c
```

Functionally extends *dict* by mapping *key* to *v*, overwriting any existing mapping for *key*, and returning an extended dictionary. The update can fail with a `exn:fail:contract` exception if *dict* does not support functional extension or if *key* is not an allowed key for the dictionary.

Examples:

```
> (dict-set #hash() 'a "apple")
#hash((a . "apple"))
> (dict-set #hash((a . "apple") (b . "beer")) 'b "banana")
#hash((a . "apple") (b . "banana"))
> (dict-set '() 'a "apple")
((a . "apple"))
> (dict-set '((a . "apple") (b . "beer")) 'b "banana")
((a . "apple") (b . "banana"))
```

```
(dict-ref dict key [failure-result]) → any
  dict : dict?
  key : any/c
  failure-result : any/c
                 = (lambda () (raise (make-exn:fail ....)))
```

Returns the value for *key* in *dict*. If no value is found for *key*, then *failure-result* determines the result:

- If *failure-result* is a procedure, it is called (through a tail call) with no arguments to produce the result.
- Otherwise, *failure-result* is returned as the result.

Examples:

```
> (dict-ref #hash((a . "apple") (b . "beer")) 'a)
"apple"
> (dict-ref #hash((a . "apple") (b . "beer")) 'c)
hash-ref: no value found for key: c
> (dict-ref #hash((a . "apple") (b . "beer")) 'c #f)
#f
> (dict-ref '((a . "apple") (b . "banana")) 'b)
"banana"
```

```
> (dict-ref #("apple" "banana") 1)
"banana"
> (dict-ref #("apple" "banana") 3 #f)
#f
> (dict-ref #("apple" "banana") -3 #f)
#f
```

```
(dict-remove! dict key) → void?
  dict : (and/c dict? (not/c immutable?))
  key : any/c
```

Removes any existing mapping for *key* in *dict*. The update can fail if *dict* is not mutable or does not support removing keys (as is the case for vectors, for example).

Examples:

```
> (define h (make-hash))
> (dict-set! h 'a "apple")
> h
#hash((a . "apple"))
> (dict-remove! h 'a)
> h
#hash()
```

```
(dict-remove dict key) → (and/c dict? immutable?)
  dict : (and/c dict? immutable?)
  key : any/c
```

Functionally removes any existing mapping for *key* in *dict*, returning the updated dictionary. The update can fail if *dict* does not support functional update or does not support removing keys.

Examples:

```
> (define h #hash())
> (define h (dict-set h 'a "apple"))
> h
#hash((a . "apple"))
> (dict-remove h 'a)
#hash()
> h
#hash((a . "apple"))
> (dict-remove h 'z)
#hash((a . "apple"))
> (dict-remove '((a . "apple") (b . "banana")) 'a)
((b . "banana"))
```

```
(dict-map dict proc) → (listof any/c)
  dict : dict?
  proc : (any/c any/c . -> . any/c)
```

Applies the procedure *proc* to each element in *dict* in an unspecified order, accumulating the results into a list. The procedure *proc* is called each time with a key and its value.

Examples:

```
> (dict-map #hash((a . "apple") (b . "banana")) vector)
(#(a "apple") #(b "banana"))
```

```
(dict-for-each dict proc) → void?
  dict : dict?
  proc : (any/c any/c . -> . any)
```

Applies *proc* to each element in *dict* (for the side-effects of *proc*) in an unspecified order. The procedure *proc* is called each time with a key and its value.

Examples:

```
> (dict-for-each #hash((a . "apple") (b . "banana"))
                 (lambda (k v)
                   (printf "~a = ~s\n" k v)))
a = "apple"
b = "banana"
```

```
(dict-count dict) → nonnegative-exact-integer?
  dict : dict?
```

Returns the number of keys mapped by *dict*, usually in constant time.

Examples:

```
> (dict-count #hash((a . "apple") (b . "banana")))
2
> (dict-count #("apple" "banana"))
2
```

```
(dict-iterate-first dict) → any/c
  dict : dict?
```

Returns `#f` if *dict* contains no elements, otherwise it returns a non-`#f` value that is a index to the first element in the dict table; "first" refers to an unspecified ordering of the dictionary elements. For a mutable *dict*, this index is guaranteed to refer to the first item only as long as no mappings are added to or removed from *dict*.

Examples:

```
> (dict-iterate-first #hash((a . "apple") (b . "banana")))
0
> (dict-iterate-first #hash())
#f
> (dict-iterate-first #("apple" "banana"))
0
> (dict-iterate-first '((a . "apple") (b . "banana")))
#<assoc-iter>
```

```
(dict-iterate-next dict pos) → any/c
  dict : dict?
  pos : any/c
```

Returns either a non-`#f` that is an index to the element in *dict* after the element indexed by *pos* or `#f` if *pos* refers to the last element in *dict*. If *pos* is not a valid index, then the `exn:fail:contract` exception is raised. For a mutable *dict*, the result index is guaranteed to refer to its item only as long as no items are added to or removed from *dict*. The `dict-iterate-next` operation should take constant time.

Examples:

```
> (define h #hash((a . "apple") (b . "banana")))
> (define i (dict-iterate-first h))
> i
0
> (dict-iterate-next h i)
1
> (dict-iterate-next h (dict-iterate-next h i))
#f
```

```
(dict-iterate-key dict pos) → any
  dict : dict?
  pos : any/c
```

Returns the key for the element in *dict* at index *pos*. If *pos* is not a valid index for *dict*, the `exn:fail:contract` exception is raised. The `dict-iterate-key` operation should take constant time.

Examples:

```
> (define h '((a . "apple") (b . "banana")))
> (define i (dict-iterate-first h))
> (dict-iterate-key h i)
a
> (dict-iterate-key h (dict-iterate-next h i))
b
```

```
(dict-iterate-value dict pos) → any
  dict : dict?
  pos : any/c
```

Returns the value for the element in *dict* at index *pos*. If *pos* is not a valid index for *dict*, the exn:fail:contract exception is raised. The dict-iterate-key operation should take constant time.

Examples:

```
> (define h '((a . "apple") (b . "banana")))
> (define i (dict-iterate-first h))
> (dict-iterate-value h i)
"apple"
> (dict-iterate-value h (dict-iterate-next h i))
"banana"
```

```
(in-dict dict) → sequence?
  dict : dict?
```

Returns a sequence whose each element is two values: a key and corresponding value from *dict*.

Examples:

```
> (define h #hash((a . "apple") (b . "banana")))
> (for/list ([(k v) (in-dict h)])
    (format "~a = ~s" k v))
("a = \"apple\"" "b = \"banana\"")
```

```
(in-dict-keys dict) → sequence?
  dict : dict?
```

Returns a sequence whose elements are the keys of *dict*.

Examples:

```
> (define h #hash((a . "apple") (b . "banana")))
> (for/list ([k (in-dict-keys h)])
    k)
(a b)
```

```
(in-dict-values dict) → sequence?
  dict : dict?
```

Returns a sequence whose elements are the values of *dict*.

Examples:

```
> (define h #hash((a . "apple") (b . "banana")))
> (for/list ([v (in-dict-values h)])
    v)
'("apple" "banana")
```

`(in-dict-pairs dict) → sequence?`
`  dict : dict?`

Returns a sequence whose elements are pairs, each containing a key and its value from `dict` (as opposed to using `in-dict`, which gets the key and value as separate values for each element).

Examples:

```
> (define h #hash((a . "apple") (b . "banana")))
> (for/list ([p (in-dict-pairs h)])
    p)
'((a . "apple") (b . "banana"))
```

`prop:dict : struct-type-property?`

A structure type property (see §4.3 "Structure Type Properties") that supplies dictionary-operation implementations for a structure type. The property value must be a vector of ten procedures (some optional) that are applied only to instances of the structure type that has the property:

- `ref` : a procedure like `dict-ref` that accepts either two or three arguments
- `set!` : a procedure like `dict-set!` that accepts three arguments, or `#f` if mutation is not supported
- `set` : a procedure like `dict-set` that accepts three arguments and returns an updated dictionary, or `#f` if functional update is not supported
- `remove!` : a procedure like `dict-remove!` that accepts two arguments, or `#f` if mutation is not supported or if key removal is not supported
- `remove` : a procedure like `dict-remove` that accepts two arguments and returns an updated dictionary, or `#f` if functional update or key removal is not supported
- `count` : a procedure like `dict-count` that accepts one argument
- `iterate-first` : a procedure like `dict-iterate-first` that accepts one argument
- `iterate-next` : a procedure like `dict-iterate-next` that accepts two arguments; the procedure is responsible for checking that the second argument is a valid position for the first argument

- *iterate-key* : a procedure like `dict-iterate-key` that accepts two arguments; the procedure is responsible for checking that the second argument is a valid position for the first argument
- *iterate-value* : a procedure like `dict-iterate-value` that accepts two arguments; the procedure is responsible for checking that the second argument is a valid position for the first argument

---

```
(make-custom-hash eql? hash-proc hash2-proc) → dict?
  eql? : (any/c any/c . -> . any/c)
  hash-proc : (any/c . -> . exact-integer?)
  hash2-proc : (any/c . -> . exact-integer?)
(make-immutable-custom-hash eql?
                            hash-proc
                            hash2-proc) → dict?
  eql? : (any/c any/c . -> . any/c)
  hash-proc : (any/c . -> . exact-integer?)
  hash2-proc : (any/c . -> . exact-integer?)
(make-weak-custom-hash eql?
                       hash-proc
                       hash2-proc) → dict?
  eql? : (any/c any/c . -> . any/c)
  hash-proc : (any/c . -> . exact-integer?)
  hash2-proc : (any/c . -> . exact-integer?)
```

Creates a dictionary that is implemented in terms of a hash table where keys are compared with *eql?* and hashed with *hash-proc* and *hash2-proc*. See `prop:equal+hash` for information on suitable equality and hashing functions.

The `make-custom-hash` and `make-weak-custom-hash` functions create a mutable dictionary that does not support functional update, while `make-immutable-custom-hash` creates an immutable dictionary that supports functional update. The dictionary created by `make-weak-custom-hash` retains its keys weakly, like the result of `make-weak-hash`.

Dictionaries created by `make-custom-hash` and company are `equal?` when they have the same mutability and key strength, the associated procedures are `equal?`, and the key–value mappings are the same when keys and values are compared with `equal?`.

Examples:

```
> (define h (make-custom-hash (lambda (a b)
                                (string=? (format "~a" a)
                                          (format "~a" b)))
                              (lambda (a)
                                (equal-hash-code
                                 (format "~a" a)))))
```

```
> (dict-set! h 1 'one)
> (dict-ref h "1")
one
```

3.16 Procedures

```
(procedure? v) → boolean
  v : any/c
```

Returns `#t` if `v` is a procedure, `#f` otherwise.

```
(apply proc v ... lst #:<kw> kw-arg ...) → any
  proc : procedure?
  v : any/c
  lst : list?
  kw-arg : any/c
```

§4.3.3 "The `apply` Function" in §"**Guide**: PLT Scheme" introduces `apply`.

Applies `proc` using the content of `(list* v ... lst)` as the (by-position) arguments. The `#:<kw> kw-arg` sequence is also supplied as keyword arguments to `proc`, where `#:<kw>` stands for any keyword.

The given `proc` must accept as many arguments as the number of `v`s plus length of `lst`, it must accept the supplied keyword arguments, and it must not require any other keyword arguments; otherwise, the `exn:fail:contract` exception is raised. The given `proc` is called in tail position with respect to the `apply` call.

Examples:

```
> (apply + '(1 2 3))
6
> (apply + 1 2 '(3))
6
> (apply + '())
0
> (apply sort (list (list '(2) '(1)) <) #:key car)
((1) (2))
```

```
(compose proc ...) → procedure?
  proc : procedure?
```

Returns a procedure that composes the given functions, applying the last `proc` first and the first `proc` last. The composed functions can consume and produce any number of values, as long as each function produces as many values as the preceding function consumes.

Examples:

```
> ((compose - sqrt) 10)
-3.1622776601683795
> ((compose sqrt -) 10)
0+3.1622776601683795i
> ((compose list split-path) (bytes->path #"/a" 'unix))
(#<path:/> #<path:a> #f)
```

3.16.1 Keywords and Arity

```
(keyword-apply proc
               kw-lst
               kw-val-lst
               v ...
               lst
               #:<kw> kw-arg ...) → any
  proc : procedure?
  kw-lst : (listof keyword?)
  kw-val-lst : list?
  v : any/c
  lst : list?
  kw-arg : any/c
```

§4.3.3 "The `apply` Function" in §"**Guide**: PLT Scheme" introduces `keyword-apply`.

Like `apply`, but *`kw-lst`* and *`kw-val-lst`* supply by-keyword arguments in addition to the by-position arguments of the *`v`*s and *`lst`*, and in addition to the directly supplied keyword arguments in the `#:<kw>` *`kw-arg`* sequence, where `#:<kw>` stands for any keyword.

The given *`kw-lst`* must be sorted using `keyword<?`. No keyword can appear twice in *`kw-lst`* or in both `kw-list` and as a `#:<kw>`, otherwise, the `exn:fail:contract` exception is raised. The given *`kw-val-lst`* must have the same length as *`kw-lst`*, otherwise, the `exn:fail:contract` exception is raised. The given *`proc`* must accept all of the keywords in *`kw-lst`* plus the `#:<kw>`s, it must not require any other keywords, and it must accept as many by-position arguments as supplied via the *`v`*s and *`lst`*; otherwise, the `exn:fail:contract` exception is raised.

Examples:

```
(define (f x #:y y #:z [z 10])
  (list x y z))

> (keyword-apply f '(#:y) '(2) '(1))
(1 2 10)
> (keyword-apply f '(#:y #:z) '(2 3) '(1))
(1 2 3)
> (keyword-apply f #:z 7 '(#:y) '(2) '(1))
```

```
(1 2 7)
```

```
(procedure-arity proc) → procedure-arity?
  proc : procedure?
```

Returns information about the number of by-position arguments accepted by `proc`. See also `procedure-arity?`.

```
(procedure-arity? v) → boolean?
  v : any/c
```

A valid arity `a` is one of the following:

- An exact non-negative integer, which means that the procedure accepts `a` arguments, only.
- A `arity-at-least` instance, which means that the procedure accepts `(arity-at-least-value a)` or more arguments.
- A list containing integers and `arity-at-least` instances, which means that the procedure accepts any number of arguments that can match one of the elements of `a`.

Examples:

```
> (procedure-arity cons)
2
> (procedure-arity list)
#(struct:arity-at-least 0)
> (arity-at-least? (procedure-arity list))
#t
> (arity-at-least-value (procedure-arity list))
0
> (arity-at-least-value (procedure-arity (lambda (x . y) x)))
1
> (procedure-arity (case-lambda [(x) 0] [(x y) 1]))
(1 2)
```

```
(procedure-arity-includes? proc k) → boolean?
  proc : procedure?
  k : exact-nonnegative-integer?
```

Returns `#t` if the procedure can accept `k` arguments when no keyword arguments are supplied, `#f` otherwise.

Examples:

```
> (procedure-arity-includes? cons 2)
#t
> (procedure-arity-includes? display 3)
#f
```

```
(procedure-reduce-arity proc arity) → procedure?
  proc : procedure?
  arity : procedure-arity?
```

Returns a procedure that is the same as *proc* (including the same name returned by `object-name`), but that accepts only arguments consistent with *arity*. In particular, when `procedure-arity` is applied to the generated procedure, it returns a value that is `equal?` to *arity*.

If the *arity* specification allows arguments that are not in `(procedure-arity proc)`, the `exn:fail:contract` exception is raised.

Examples:

```
> (define my+ (procedure-reduce-arity + 2))
> (my+ 1 2)
3
> (my+ 1 2 3)
+: expects 2 arguments, given 3: 1 2 3
```

```
(procedure-keywords proc) → (listof keyword?)
                            (or/c (listof keyword?)
                                  false/c)
  proc : procedure?
```

Returns information about the keyword arguments required and accepted by a procedure. The first result is a list of keywords (sorted by `keyword<?`) that are required when applying *proc*. The second result is a list of accepted keywords (sorted by `keyword<?`), or `#f` to mean that any keyword is accepted. When the second result is a list, every element in the first list is also in the second list.

Examples:

```
> (procedure-keywords +)
()
()
> (procedure-keywords (lambda (#:tag t #:mode m) t))
(#:mode #:tag)
(#:mode #:tag)
> (procedure-keywords (lambda (#:tag t #:mode [m #f]) t))
(#:tag)
```

```
(#:mode #:tag)
```

```
(make-keyword-procedure proc [plain-proc]) → procedure?
  proc : (((listof keyword?) list?) list? . ->* . any)
  plain-proc : procedure?
             = (lambda args (keyword-apply proc null null args))
```

Returns a procedure that accepts all keyword arguments (without requiring any keyword arguments). See also `procedure-reduce-keyword-arity`.

When the result is called with keyword arguments, then *`proc`* is called; the first argument is a list of keywords sorted by `keyword<?`, the second argument is a parallel list containing a value for each keyword, and the remaining arguments are the by-position arguments.

When the result is called without keyword arguments, then *`plain-proc`* is called. Furthermore, `procedure-arity` obtains its result from *`plain-proc`*.

Examples:

```
(define show
  (make-keyword-procedure (lambda (kws kw-args . rest)
                            (list kws kw-args rest))))

> (show 1)
(() () (1))
> (show #:init 0 1 2 3 #:extra 4)
((#:extra #:init) (4 0) (1 2 3))
```

```
(procedure-reduce-keyword-arity proc
                                arity
                                required-kws
                                allowed-kws) → procedure?
  proc : procedure?
  arity : procedure-arity?
  required-kws : (listof keyword?)
  allowed-kws : (or/c (listof keyword?)
                      false/c)
```

Like `procedure-reduce-arity`, but constrains the keyword arguments according to *`required-kws`* and *`allowed-kws`*, which must be sorted using `keyword<?`. If *`allowed-kws`* is `#f`, then the resulting procedure still accepts any keyword, otherwise the keywords in *`required-kws`* must be a subset of those in *`allowed-kws`*. The original *`proc`* must require no more keywords than the ones listed din *`required-kws`*, and it must allow at least the keywors in *`allowed-kws`* (or it must allow all keywords if *`allowed-kws`* is `#f`).

Examples:

```
(define orig-show
  (make-keyword-procedure (lambda (kws kw-args . rest)
                           (list kws kw-args rest))))

(define show (procedure-reduce-keyword-arity
              orig-show 3 '(#:init) '(#:extra #:init)))

> (show #:init 0 1 2 3 #:extra 4)
((#:extra #:init) (4 0) (1 2 3))
> (show 1)
procedure ...me/private/kw.ss:124:25: expects 3 arguments
plus an argument with keyword #:init plus an optional
argument with keyword #:extra, given 1: 1
> (show #:init 0 1 2 3 #:extra 4 #:more 7)
procedure application: procedure:
#<procedure:...me/private/kw.ss:124:25>; does not expect an
argument with keyword #:more; arguments were: 1 2 3 #:extra
4 #:init 0 #:more 7
```

```
(struct arity-at-least (value))
  value : nonnegative-exact-integer?
```

This structure type is used for the result of `procedure-arity`. See also `procedure-arity?`.

```
prop:procedure : struct-type-property?
```

A structure type property to indentify structure types whose instances can be applied as procedures. In particular, when `procedure?` is applied to the instance, the result will be `#t`, and when an instance is used in the function position of an application expression, a procedure is extracted from the instance and used to complete the procedure call.

If the `prop:procedure` property value is an integer, it designates a field within the structure that should contain a procedure. The integer must be between `0` (inclusive) and the number of non-automatic fields in the structure type (exclusive, not counting supertype fields). The designated field must also be specified as immutable, so that after an instance of the structure is created, its procedure cannot be changed. (Otherwise, the arity and name of the instance could change, and such mutations are generally not allowed for procedures.) When the instance is used as the procedure in an application expression, the value of the designated field in the instance is used to complete the procedure call. (This procedure can be another structure that acts as a procedure; the immutability of procedure fields disallows cycles in the procedure graph, so that the procedure call will eventually continue with a non-structure procedure.) That procedure receives all of the arguments from the application expression. The procedure's name (see `object-name`) and arity (see `procedure-arity`) are also used for the name and arity of the structure. If the value in the designated field is not a procedure, then the instance behaves like (case-lambda) (i.e., a procedure which does not accept any number of arguments). See also procedure-extract-target.

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

Examples:

```
> (define-struct annotated-proc (base note)
                 #:property prop:procedure
                            (struct-field-index base))
> (define plus1 (make-annotated-proc
                  (lambda (x) (+ x 1))
                  "adds 1 to its argument"))
> (procedure? plus1)
#t
> (annotated-proc? plus1)
#t
> (plus1 10)
11
> (annotated-proc-note plus1)
"adds 1 to its argument"
```

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

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

Examples:

```
> (define-struct fish (weight color)
                 #:mutable
                 #:property
                 prop:procedure
                 (lambda (f n)
                   (let ([w (fish-weight f)])
                    (set-fish-weight! f (+ n w)))))
> (define wanda (make-fish 12 'red))
```

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

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

```
(procedure-struct-type? type) → boolean?
  type : struct-type?
```

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

```
(procedure-extract-target proc) → (or/c false/c procedure?)
  proc : procedure?
```

If *`proc`* is an instance of a structure type with property `prop:procedure`, and if the property value indicates a field of the structure, and if the field value is a procedure, then `procedure-extract-target` returns the field value. Otherwise, the result if `#f`.

```
prop:arity-string : struct-type-property?
```

This property is used for reporting arity-mismatch errors when a structure type with the `prop:procedure` property is applied to the wrong number of arguments. The value of the `prop:arity-string` property must be a procedure that takes a single argument, which is the misapplied structure, and returns a string. The result string is used after the word "expects," and it is followed in the error message by the number of actual arguments.

Arity-mismatch reporting automatically uses `procedure-extract-target` when the `prop:arity-string` property is not associated with a procedure structure type.

Examples:

```
> (define-struct evens (proc)
    #:property prop:procedure (struct-field-index proc)
    #:property prop:arity-string
    (lambda (p)
```

```
       "an even number of arguments"))
> (define pairs
    (make-evens
     (case-lambda
      [() null]
      [(a b . more)
       (cons (cons a b)
             (apply pairs more))])))
> (pairs 1 2 3 4)
((1 . 2) (3 . 4))
> (pairs 5)
#<procedure>: expects an even number of arguments, given 1:
5
```

3.16.2 Reflecting on Primitives

A *primitive procedure* is a built-in procedure that is implemented in low-level language. Not all procedures of `scheme/base` are primitives, but many are. The distinction is mainly useful to other low-level code.

`(primitive? v) → boolean?`
  `v : any/c`

Returns `#t` if *v* is a primitive procedure, `#f` otherwise.

`(primitive-closure? v) → boolean`
  `v : any/c`

Returns `#t` if *v* is internally implemented as a primitive closure rather than a simple primitive procedure, `#f` otherwise.

`(primitive-result-arity prim) → procedure-arity?`
  `prim : primitive?`

Returns the arity of the result of the primitive procedure *prim* (as opposed to the procedure's input arity as returned by `arity`). For most primitives, this procedure returns `1`, since most primitives return a single value when applied.

3.16.3 Additional Procedure Functions

```
(require scheme/function)
```

The bindings documented in this section are provided by the `scheme/function` and `scheme` libraries, but not `scheme/base`.

`(negate proc) → procedure?`
`  proc : procedure?`

Returns a procedure that is just like *proc*, except that it returns the `not` of *proc*'s result.

Examples:

```
> (filter (negate symbol?) '(1 a 2 b 3 c))
(1 2 3)
> (map (negate =) '(1 2 3) '(1 1 1))
(#f #t #t)
```

`(curry proc) → procedure?`
`  proc : procedure?`
`(curry proc v ...+) → any/c`
`  proc : procedure?`
`  v : any/c`

Returns a procedure that is a curried version of *proc*. When the resulting procedure is first applied, unless it is given the maximum number of arguments that it can accept, the result is a procedure to accept additional arguments.

Examples:

```
> ((curry list) 1 2)
#<procedure:curried>
> ((curry cons) 1)
#<procedure:curried>
> ((curry cons) 1 2)
(1 . 2)
```

After the first application of the result of `curry`, each further application accumulates arguments until an acceptable number of arguments have been accumulated, at which point the original *proc* is called.

Examples:

```
> (((curry list) 1 2) 3)
(1 2 3)
> (((curry list) 1) 3)
(1 3)
> ((((curry foldl) +) 0) '(1 2 3))
6
```

A function call `(curry proc v ...)` is equivalent to `((curry proc) v ...)`. In other words, curry itself is curried.

The curry function provides limited support for keyworded functions: only the curry call itself can receive keyworded arguments to be propagated eventually to *proc*.

Examples:

```
> (map ((curry +) 10) '(1 2 3))
(11 12 13)
> (map (curry + 10) '(1 2 3))
(11 12 13)
> (map (compose (curry * 2) (curry + 10)) '(1 2 3))
(22 24 26)
> (define foo (curry (lambda (x y z) (list x y z))))
> (foo 1 2 3)
(1 2 3)
> (((((foo) 1) 2)) 3)
(1 2 3)
```

```
(curryr proc) → procedure?
  proc : procedure?
(curryr proc v ...+) → any/c
  proc : procedure?
  v : any/c
```

Like curry, except that the arguments are collected in the opposite direction: the first step collects the rightmost group of arguments, and following steps add arguments to the left of these.

Examples:

```
> (map (curryr list 'foo) '(1 2 3))
((1 foo) (2 foo) (3 foo))
```

3.17 Void and Undefined

The constant #<void> is returned by most forms and procedures that have a side-effect and no useful result. The constant #<undefined> is used as the initial value for letrec bindings.

```
(void? v) → void?
  v : any/c
```

Returns #t if *v* is the constant #<void>, #f otherwise.

```
(void v ...) → void?
  v : any/c
```

Returns the constant `#<void>`. Each `v` argument is ignored.

4 Structures

§5 "Programmer-Defined Datatypes" in §"**Guide**: PLT Scheme" introduces structure types via `define-struct`.

A *structure type* is a record datatype composing a number of *fields*. A *structure*, an instance of a structure type, is a first-class value that contains a value for each field of the structure type. A structure instance is created with a type-specific constructor procedure, and its field values are accessed and changed with type-specific accessor and mutator 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.

A structure type's fields are essentially unnamed, though names are supported for error-reporting purposes. The constructor procedure takes one value for each field of the structure type, except that some of the fields of a structure type can be *automatic fields*; the automatic fields are initialized to a constant that is associated with the structure type, and the corresponding arguments are omitted for the constructor procedure. All automatic fields in a structure type follow the non-automatic fields.

A structure type can be created as 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 it may have its own fields in addition to the fields of the base type.

A structure subtype "inherits" the fields of its base type. If the base type has m fields, and if n fields are specified for the new structure subtype, then the resulting structure type has $m+n$ fields. The value for automatic fields can be different in a subtype than in its base type.

If m' of the original m fields are non-automatic (where $m'<m$), and n' of the new fields are automatic (where $n'<n$), then $m'+n'$ 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 (or its supertypes), and the last n are accessed with subtype-specific selectors. Subtype-specific accessors and mutators for the first m fields do not exist.

The `define-struct` form and `make-struct-type` procedure typically create a new structure type, but they can also access *prefab* (i.e., previously fabricated) structure types that are globally shared, and whose instances can be parsed and written by the default reader (see §12.6 "The Reader") and printer (see §12.7 "The Printer"). Prefab structure types can inherit only from other prefab structure types, and they cannot have guards (see §4.2 "Creating Structure Types") or properties (see §4.3 "Structure Type Properties"). Exactly one prefab structure type exists for each combination of name, supertype, field count, automatic field count, automatic field value (when there is at least one automatic field), and field mutability.

§12.11 "Serialization" also provides information on reading and writing structures.

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

4.1 Defining Structure Types: define-struct

§5 "Programmer-Defined Datatypes" in §"Guide: PLT Scheme" introduces define-struct.

```
(define-struct id-maybe-super (field ...)
               struct-option ...)

id-maybe-super = id
               | (id super-id)

         field = field-id
               | [field-id field-option ...]

 struct-option = #:mutable
               | #:super super-expr
               | #:inspector inspector-expr
               | #:auto-value auto-expr
               | #:guard guard-expr
               | #:property prop-expr val-exr
               | #:transparent
               | #:prefab
               | #:omit-define-syntaxes
               | #:omit-define-values

  field-option = #:mutable
               | #:auto
```

Creates a new structure type (or uses a pre-existing structure type if `#:prefab` is specified), and binds transformers and variables related to the structure type.

A `define-struct` form with *n* `fields` defines up to 4+2*n* names:

- `struct:id`, a *structure type descriptor* value that represents the structure type.
- `make-id`, a *constructor* procedure that takes *m* arguments and returns a new instance of the structure type, where *m* is the number of `fields` that do not include an `#:auto` option.
- `id?`, a *predicate* procedure that returns `#t` for instances of the structure type (constructed by `make-id` or the constructor for a subtype) and `#f` for any other value.
- `id-field-id`, for each `field`; an *accessor* procedure that takes an instance of the structure type and extracts the value for the corresponding field.
- `set-id-field-id!`, for each `field` that includes a `#:mutable` option, or when the `#:mutable` option is specified as a `struct-option`; a *mutator* procedure that takes an instance of the structure type and a new field value. The structure is destructively updated with the new value, and `#<void>` is returned.

- *id*, a transformer binding that encapsulates information about the structure type declaration. This binding is used to define subtypes, and it also works with the shared and match forms. For detailed information about the binding of *id*, see §4.6 "Structure Type Transformer Binding".

If *super-id* is provided, it must have a transformer binding of the same sort bound to *id* (see §4.6 "Structure Type Transformer Binding"), and it specifies a supertype for the structure type. Alternately, the #:super option can be used to specify an expression that must produce a structure type descriptor. See §4 "Structures" for more information on structure subtypes and supertypes. If both *super-id* and #:super are provided, a syntax error is reported.

If the #:mutable option is specified for an individual field, then the field can be mutated in instances of the structure type, and a mutator procedure is bound. Supplying #:mutable as a *struct-option* is the same as supplying it for all *field*s. If #:mutable is specified as both a *field-option* and *struct-option*, a syntax error is reported.

The #:inspector, #:auto-value, and #:guard options specify an inspector, value for automatic fields, and guard procedure, respectively. See make-struct-type for more information on these attributes of a structure type. The #:property option, which is the only one that can be supplied multiple times, attaches a property value to the structure type; see §4.3 "Structure Type Properties" for more information on properties. The #:transparent option is a shorthand for #:inspector #f.

The #:prefab option obtains a prefab (pre-defined, globally shared) structure type, as opposed to creating a new structure type. Such a structure type is inherently transparent and cannot have a guard or properties, so using #:prefab with #:transparent, #:inspector, #:guard, or #:property is a syntax error. If a supertype is specified, it must also be a prefab structure type.

If the #:omit-define-syntaxes option is supplied, then *id* is not bound as a transformer. If the #:omit-define-values option is supplied, then none of the usual variables are bound. If both are supplied, then the define-struct form is equivalent to (begin).

If #:auto is supplied as a *field-option*, then the constructor procedure for the structure type does not accept an argument corresponding to the field. Instead, the structure type's automatic value is used for the field, as specified by the #:auto-value option, or as defaults to #f when #:auto-value is not supplied.

If a *field* includes the #:auto option, then all fields after it must also include #:auto, otherwise a syntax error is reported. If any *field-option* or *struct-option* keyword is repeated, other than #:property, a syntax error is reported.

For serialization, see define-serializable-struct.

Examples:

```
(define-struct posn (x y [z #:auto]))
```

```
                #:auto-value 0
                #:transparent)
```

```
> (make-posn 1 2)
#(struct:posn 1 2 0)
> (posn? (make-posn 1 2))
#t
> (posn-y (make-posn 1 2))
2
```

```
(define-struct (color-posn posn) (hue) #:mutable)
(define cp (make-color-posn 1 2 "blue"))
```

```
> (color-posn-hue cp)
"blue"
> cp
#(struct:color-posn 1 2 0 ...)
> (set-posn-z! cp 3)
```

reference to undefined identifier: set-posn-z!

```
(struct-field-index field-id)
```

This form can only appear as an expression within a `define-struct` form; normally, it is used with `#:property`, especially for a property like `prop:procedure`. The result of

Examples:

```
(define-struct mood-procedure ([base] rating)
                #:property prop:procedure (struct-field-index base))
```

```
(define happy+ (make-mood-procedure add1 10))
```

```
> (happy+ 2)
3
> (mood-procedure-rating happy+)
10
```

```
(define-struct/derived (id . rest-form)
  id-maybe-super (field ...) struct-option ...)
```

Like `define-struct`, but intended for use by macros that expand to `define-struct`. The form immediately after `define-struct/derived` is used for all syntax-error reporting, and the only constraint on the form is that it starts with some *id*.

Examples:

```
> (define-syntax (define-xy-struct stx)
```

```
    (syntax-case stx ()
     [(ds name . rest)
      (with-syntax ([orig stx])
        #'(define-struct/derived orig name (x y) . rest))]))
> (define-xy-struct posn)
> (posn-x (make-posn 1 2))
1
> (define-xy-struct posn #:mutable)
> (set-posn-x! (make-posn 1 2) 0)
> (define-xy-struct posn #:bad-option)
```

eval:52:0: define-xy-struct: unrecognized struct-specification keyword at: #:bad-option in: (define-xy-struct posn #:bad-option)

4.2 Creating Structure Types

```
(make-struct-type name
                  super-type
                  init-field-cnt
                  auto-field-cnt
                 [auto-v
                  props
                  inspector
                  proc-spec
                  immutables
                  guard])
 → struct-type?
   struct-constructor-procedure?
   struct-predicate-procedure?
   struct-accessor-procedure?
   struct-mutator-procedure?
  name : symbol?
  super-type : (or/c struct-type? false/c)
  init-field-cnt : exact-nonnegative-integer?
  auto-field-cnt : exact-nonnegative-integer?
  auto-v : any/c = #f
  props : (listof (cons/c struct-type-property? = null
                          any/c))
  inspector : (or/c inspector? false/c (one-of/c 'prefab))
            = (current-inspector)
  proc-spec : (or/c procedure?                = #f
                    exact-nonnegative-integer?
                    false/c)
```

```
immutables : (listof exact-nonnegative-integer?) = null
guard : (or/c procedure? false/c) = #f
```

Creates a new structure type, unless *inspector* is 'prefab, in which case make-struct-type accesses a prefab structre type. The *name* argument is used as the type name. If *super-type* is not #f, the resulting type is a subtype of the corresponding structure type.

The resulting structure type has *init-field-cnt*+*auto-field-cnt* fields (in addition to any fields from *super-type*), but only *init-field-cnt* constructor arguments (in addition to any constructor arguments from *super-type*). The remaining fields are initialized with *auto-v*. The total field count (including *super-type* fields) must be no more than 32768.

The *props* argument is a list of pairs, where the car of each pair is a structure type property descriptor, and the cdr is an arbitrary value. See §4.3 "Structure Type Properties" for more information about properties. When *inspector* is 'prefab, then *props* must be null.

The *inspector* argument normally controls access to reflective information about the structure type and its instances; see §13.8 "Structure Inspectors" for more information. If *inspector* is 'prefab, then the resulting prefab structure type and its instances are always transparent.

If *proc-spec* is an integer or procedure, instances of the structure type act as procedures. See prop:procedure for further information. Providing a non-#f value for *proc-spec* is the same as pairing the value with prop:procedure in *props*, plus including *proc-spec* in *immutables* when *proc-spec* is an integer.

The *immutables* argument provides a list of field positions. Each element in the list must be unique, otherwise exn:fail:contract exception is raised. Each element must also fall in the range 0 (inclusive) to *init-field-cnt* (exclusive), otherwise exn:fail:contract exception is raised.

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

The result of make-struct-type is five values:

- a structure type descriptor,

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

Examples:

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

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

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

```
> (a-ref a-c 1)
2.0

> (define p1 #s(p a b c))
> (define-values (struct:p make-p p? p-ref p-set!)
    (make-struct-type 'p #f 3 0 #f null 'prefab #f '(0 1 2)))
> (p? p1)
#t
> (p-ref p1 0)
a
> (make-p 'x 'y 'z)
#s(p x y z)
```

```
(make-struct-field-accessor accessor-proc
                            field-pos
                            field-name)    → procedure?
  accessor-proc : struct-accessot-procedure?
  field-pos : exact-nonnegative-integer?
  field-name : symbol?
```

Returns a field accessor that is equivalent to `(lambda (s) (accessor-proc s field-pos))`. The *`accessor-proc`* must be an accessor returned by `make-struct-type`. The name of the resulting procedure for debugging purposes is derived from *`field-name`* and the name of *`accessor-proc`*'s structure type.

For examples, see `make-struct-type`.

```
(make-struct-field-mutator mutator-proc
                           field-pos
                           field-name)    → procedure?
  mutator-proc : struct-mutator-procedure?
  field-pos : exact-nonnegative-integer?
  field-name : symbol?
```

Returns a field mutator that is equivalent to `(lambda (s v) (mutator-proc s field-pos v))`. The *`mutator-proc`* must be a mutator returned by `make-struct-type`. The name of the resulting procedure for debugging purposes is derived from *`field-name`* and the name of *`mutator-proc`*'s structure type.

For examples, see `make-struct-type`.

4.3 Structure Type Properties

A *structure type property* allows per-type information to be associated with a structure type (as opposed to per-instance information associated with a structure value). A property value is associated with a structure type through the make-struct-type procedure (see §4.2 "Creating Structure Types") or through the #:property option of define-struct. Subtypes inherit the property values of their parent types, and subtypes can override an inherited property value with a new value.

```
(make-struct-type-property name [guard]) → struct-type-property?
                                           procedure?
                                           procedure?
  name : symbol?
  guard : (or/c procedure? false/c) = #f
```

Creates a new structure type property and returns three values:

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

If the optional *guard* is supplied as a procedure, it is called by make-struct-type before attaching the property to a new structure type. The guard-proc must accept two arguments: a value for the property supplied to make-struct-type, and a list containing information about the new structure type. The list contains the values that struct-type-info would return for the new structure type if it skipped the immediate current-inspector control check (but not the check for exposing an ancestor structure type, if any; see §13.8 "Structure Inspectors").

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

Examples:

```
> (define-values (prop:p p? p-ref) (make-struct-type-property 'p))
> (define-values (struct:a make-a a? a-ref a-set!)
    (make-struct-type 'a #f 2 1 'uninitialized
                      (list (cons prop:p 8))))
> (p? struct:a)
#t
> (p? 13)
#f
> (define an-a (make-a 'x 'y))
> (p? an-a)
#t
> (p-ref an-a)
8
> (define-values (struct:b make-b b? b-ref b-set!)
    (make-struct-type 'b #f 0 0 #f))
> (p? struct:b)
#f
```

```
(struct-type-property? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a structure type property descriptor value, `#f` otherwise.

4.4 Copying and Updating Structures

```
(struct-copy id struct-expr [field-id expr] ...)
```

Creates a new instance of the structure type *id* with the same field values as the structure produced by *struct-expr*, except that the value of each supplied *field-id* is instead determined by the corresponding *expr*.

The *id* must have a transformer binding that encapsulates information about a structure type (i.e., like the initial identifier bound by `define-struct`), and the binding must supply a constructor, a predicate, and all field accessors.

Each *field-id* is combined with *id* to form *id-field-id* (using the lexical context of *field-id*), which must be one of the accessor bindings in *id*. The accessor bindings determined by different *field-id*s must be distinct. The order of the *field-id*s need not match the order of the corresponding fields in the structure type.

The *struct-expr* is evaluated first. The result must be an instance of the *id* structure type, otherwise the `exn:fail:contract` exception is raised. Next, the field *expr*s are evaluated in order (even if the fields that correspond to the *field-id*s are in a different order). Finally, the new structure instance is created.

The result of *struct-expr* can be an instance of a sub-type of *id*, but the resulting copy is an immediate instance of *id* (not the sub-type).

4.5 Structure Utilities

```
(struct->vector v [opaque-v]) → vector?
  v : any/c
  opaque-v : any/c = '...
```

Creates a vector representing *v*. The first slot of the result vector contains a symbol whose printed name has the form `struct:id`. Each remaining slot contains either the value of a field in *v*, if it is accessible via the current inspector, or *opaque-v* for a field that is not accessible. A single *opaque-v* value is used in the vector for contiguous inaccessible fields. (Consequently, the size of the vector does not match the size of the `struct` if more than one field is inaccessible.)

```
(struct? v) → any
  v : any/c
```

Returns `#t` if `struct->vector` exposes any fields of *v* with the current inspector, `#f` otherwise.

```
(struct-type? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a structure type descriptor value, `#f` otherwise.

```
(struct-constructor-procedure? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a constructor procedure generated by `define-struct` or `make-struct-type`, `#f` otherwise.

```
(struct-predicate-procedure? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a predicate procedure generated by `define-struct` or `make-struct-type`, `#f` otherwise.

```
(struct-accessor-procedure? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is an accessor procedure generated by `define-struct`, `make-struct-type`, or `make-struct-field-accessor`, `#f` otherwise.

```
(struct-mutator-procedure? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a mutator procedure generated by `define-struct`, `make-struct-type`, or `make-struct-field-mutator`, `#f` otherwise.

```
(prefab-struct-key v) → (or/c false/c symbol? list?)
  v : any/c
```

Returns `#f` if *v* is not an instance of a prefab structure type. Otherwise, the result is the shorted key that could be with `make-prefab-struct` to create an instance of the structure type.

Examples:

```
> (prefab-struct-key #s(cat "Garfield"))
cat
> (define-struct cat (name) #:prefab)
> (define-struct (cute-cat cat) (shipping-dest) #:prefab)
> (make-cute-cat "Nermel" "Abu Dhabi")
#s((cute-cat cat 1) "Nermel" "Abu Dhabi")
```

```
(make-prefab-struct key v ...) → struct?
  key : (or/c symbol? list?)
  v : any/c
```

Creates an instance of a prefab structure type, using the *v*s as field values. The *key* and the number of *v*s determine the prefab structure type.

A *key* identifies a structure type based on a list with the following items:

- A symbol for the structure type's name.
- An exact, nonnegative integer representing the number of non-automatic fields in the structure type, not counting fields from the supertype (if any).
- A list of two items, where the first is an exact, nonnegative integer for the number of automatic fields in the structure type that are not from the supertype (if any), and the second element is an arbitrary value that is the value for the automatic fields.

- A vector of exact, nonnegative integers that indicate mutable non-automatic fields in the structure type, counting from 0 and not including fields from the supertype (if any).
- Nothing else, if the structure type has no supertype. Otherwise, the rest of the list matches is the key for the supertype.

An empty vector and an auto-field list that starts with 0 can be omitted. Furthermore, the first integer (which indicates the number of non-automatic fields) can be omitted, since it can be inferred from the number of supplied *v*s. Finally, a single symbol can be used instead of a list that contains only a symbol (in the case that the structure type has no supertype, no automatic fields, and no mutable fields).

The total field count must be no more than 32768. If the number of fields indicated by *key* is inconsistent with the number of supplied *v*s, the `exn:fail:contract` exception is raised.

Examples:

```
> (make-prefab-struct 'clown "Binky" "pie")
#s(clown "Binky" "pie")
> (make-prefab-struct '(clown 2) "Binky" "pie")
#s(clown "Binky" "pie")
> (make-prefab-struct '(clown 2 (0 #f) #()) "Binky" "pie")
#s(clown "Binky" "pie")
> (make-prefab-struct '(clown 1 (1 #f) #()) "Binky" "pie")
#s((clown (1 #f)) "Binky" "pie")
> (make-prefab-struct '(clown 1 (1 #f) #(0)) "Binky" "pie")
#s((clown (1 #f) #(0)) "Binky" "pie")
```

```
(prefab-key->struct-type key field-count) → struct-type?
  key : (or/c symbol? list?)
  field-count : (integer-in 0 32768)
```

Returns a structure type descriptor for the prefab structure type specified by the combination of *key* and *field-count*.

4.6 Structure Type Transformer Binding

The `define-struct` form binds the name of a structure type as a transformer binding that records the other identifiers bound to the structure type, the constructor procedure, the predicate procedure, and the field accessor and mutator procedures. This information can be used during the expansion of other expressions via `syntax-local-value`.

For example, the `define-struct` variant for subtypes uses the base type name *t* to find the variable `struct:t` containing the base type's descriptor; it also folds the field accessor and mutator information for the base type into the information for the subtype. As another example, the `match` form uses a type name to find the predicates and field accessors for the structure type. The `struct` form in an imported signature for `unit` causes the `unit` transformer to generate information about imported structure types, so that `match` and subtyping `define-struct` forms work within the unit.

The expansion-time information for a structure type is represented either as a structure that encapsulates a procedure that takes no arguments and returns a list of six element, or it can be represented directly as a list of six elements (of the same sort that the encapsulated procedure must return):

- an identifier that is bound to the structure type's descriptor, or `#f` it none is known;
- an identifier that is bound to the structure type's constructor, or `#f` it none is known;
- an identifier that is bound to the structure type's predicate, or `#f` it none is known;
- a list of identifiers bound to the field accessors of the structure type, optionally with `#f` as the list's last element. A `#f` as the last element indicates that the structure type may have additional fields, otherwise the list is a reliable indicator of the number of fields in the structure type. Furthermore, the accessors are listed in reverse order for the corresponding constructor arguments. (The reverse order enables sharing in the lists for a subtype and its base type.)
- a list of identifiers bound to the field mutators of the structure type, or `#f` for each field that has no known mutator, and optionally with an extra `#f` as the list's last element (if the accessor list has such a `#f`). The list's order and the meaning of a final `#f` are the same as for the accessor identifiers, and the length of the mutator list is the same as the accessor list's length.
- an identifier that determines a super-type for the structure type, `#f` if the super-type (if any) is unknown, or `#t` if there is no super-type. If a super-type is specified, the identifier is also bound to structure-type expansion-time information.

Use `struct-info?` to recognize both forms of information, and use `extract-struct-info` to obtain a list from either representation. Use `make-struct-info` to encapsulate a procedure that represents structure type information.

The implementor of a syntactic form can expect users of the form to know what kind of information is available about a structure type. For example, the `match` implementation works with structure information containing an incomplete set of accessor bindings, because the user is assumed to know what information is available in the context of the `match` expression. In particular, the `match` expression can appear in a `unit` form with an imported structure type, in which case the user is expected to know the set of fields that are listed in the signature for the structure type.

```
(require scheme/struct-info)
```

The bindings documented in this section are provided by the `scheme/struct-info` library, not `scheme/base` or `scheme`.

```
(struct-info? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is either a six-element list with the correct shape for representing structure-type information, or a procedure encapsulated by `make-struct-info`.

```
(checked-struct-info? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a procedure encapsulated by `make-struct-info` and produced by `define-struct`, but only when no parent type is specified or the parent type is also specified through a transformer binding to such a value).

```
(make-struct-info thunk) → struct-info?
  thunk : (-> (and/c struct-info? list?))
```

Encapsulates a thunk that returns structure-type information in list form.

```
(extract-struct-info v) → (and/c struct-info? list?)
  v : struct-info?
```

Extracts the list form of the structure type information represented by *v*.

```
struct:struct-info : struct-type?
```

The structure type descriptor for the structure type returned by `make-struct-info`. This structure type descriptor is mostly useful for creating structure subtypes. The structure type includes a guard that checks an instance's first field in the same way as `make-struct-info`.

5 Classes and Objects

```
(require scheme/class)
```

The bindings documented in this section are provided by the `scheme/class` and `scheme` libraries, but not `scheme/base`.

A *class* specifies

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

In the context of the class system, an *object* is a collection of bindings for fields that are instantiated according to a class description.

The class system allows a program to define a new class (a *derived class*) in terms of an existing class (the *superclass*) using inheritance, overriding, and augmenting:

- *inheritance*: An object of a derived class supports methods and instantiates fields declared by the derived class's superclass, as well as methods and fields declared in the derived class expression.
- *overriding*: Some methods declared in a superclass can be replaced in the derived class. References to the overridden method in the superclass use the implementation in the derived class.
- *augmenting*: Some methods declared in a superclass can be merely extended in the derived class. The superclass method specifically delegates to the augmenting method in the derived class.

An *interface* is a collection of method names to be implemented by a class, combined with a derivation requirement. A class *implements* an interface when it

- declares (or inherits) a public method for each variable in the interface;
- is derived from the class required by the interface, if any; and
- specifically declares its intention to implement the interface.

A class can implement any number of interfaces. A derived class automatically implements any interface that its superclass implements. Each class also implements an implicitly-defined interface that is associated with the class. The implicitly-defined interface contains all of the class's public method names, and it requires that all other implementations of the interface are derived from the class.

A new interface can *extend* one or more interfaces with additional method names; each class that implements the extended interface also implements the original interfaces. The derivation requirements of the original interface must be consistent, and the extended interface inherits the most specific derivation requirement from the original interfaces.

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

5.1 Creating Interfaces

```
(interface (super-interface-expr ...) id ...)
```

Produces an interface. The `id`s must be mutually distinct.

Each `super-interface-expr` is evaluated (in order) when the `interface` expression is evaluated. The result of each `super-interface-expr` must be an interface value, otherwise the `exn:fail: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 `id`s, plus all identifiers from the superinterfaces. Duplicate identifier names among the superinterfaces are ignored, but if a superinterface contains one of the `id`s in the `interface` expression, the `exn:fail:object` exception is raised.

If no `super-interface-expr`s are provided, then the derivation requirement of the resulting interface is trivial: any class that implements the interface must be derived from `object%`. Otherwise, the implementation requirement of the resulting interface is the most specific requirement from its superinterfaces. If the superinterfaces specify inconsistent derivation requirements, the `exn:fail:object` exception is raised.

5.2 Creating Classes

```
object% : class?
```

A built-in class that has no methods fields, implements only its own interface (`(class->interface object%)`), and is transparent (i.e,. its inspector is `#f`, so all immediate instances are `equal?`). All other classes are derived from `object%`.

```
(class* superclass-expr (interface-expr ...)
  class-clause
  ...)
```

```
      class-clause = (inspect inspector-expr)
                   | (init init-decl ...)
                   | (init-field init-decl ...)
                   | (field field-decl ...)
                   | (inherit-field maybe-renamed ...)
                   | (init-rest id)
                   | (init-rest)
                   | (public maybe-renamed ...)
                   | (pubment maybe-renamed ...)
                   | (public-final maybe-renamed ...)
                   | (override maybe-renamed ...)
                   | (overment maybe-renamed ...)
                   | (override-final maybe-renamed ...)
                   | (augment maybe-renamed ...)
                   | (augride maybe-renamed ...)
                   | (augment-final maybe-renamed ...)
                   | (private id ...)
                   | (inherit maybe-renamed ...)
                   | (inherit/super maybe-renamed ...)
                   | (inherit/inner maybe-renamed ...)
                   | (rename-super renamed ...)
                   | (rename-inner renamed ...)
                   | method-definition
                   | definition
                   | expr
                   | (begin class-clause ...)

         init-decl = id
                   | (maybe-renamed)
                   | (maybe-renamed default-value-expr)

        field-decl = (maybe-renamed default-value-expr)

     maybe-renamed = id
                   | renamed

           renamed = (internal-id external-id)

 method-definition = (define-values (id) method-procedure)

  method-procedure = (lambda kw-formals expr ...+)
                   | (case-lambda (formals expr ...+) ...)
                   | (#%plain-lambda formals expr ...+)
                   | (let-values (((id) method-procedure) ...)
                       method-procedure)
                   | (letrec-values (((id) method-procedure) ...)
                       method-procedure)
                   | (let-values (((id) method-procedure) ...+)
                       id)
                   | (letrec-values (((id) method-procedure) ...+)
                       id)
```

Produces a class value.

The *superclass-expr* expression is evaluated when the `class*` expression is evaluated. The result must be a class value (possibly `object%`), otherwise the `exn:fail: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*` expression is evaluated, after *superclass-expr* is evaluated. The result of each *interface-expr* must be an interface value, otherwise the `exn:fail:object` exception is raised. The interfaces returned by the *interface-expr*s are all implemented by the class. For each identifier in each interface, the class (or one of its ancestors) must declare a public method with the same name, otherwise the `exn:fail:object` exception is raised. The class's superclass must satisfy the implementation requirement of each interface, otherwise the `exn:fail:object` exception is raised.

An `inspect` *class-clause* selects an inspector (see §13.8 "Structure Inspectors") for the class extension. The *inspector-expr* must evaluate to an inspector or `#f` when the `class*` form is evaluated. Just as for structure types, an inspector controls access to the class's fields, including private fields, and also affects comparisons using `equal?`. If no `inspect` clause is provided, access to the class is controlled by the parent of the current inspector (see §13.8 "Structure Inspectors"). A syntax error is reported if more than one `inspect` clause is specified.

The other *class-clause*s define initialization arguments, public and private fields, and public and private methods. For each *id* or *maybe-renamed* in a `public`, `override`, `augment`, `pubment`, `overment`, `augride`, `public-final`, `override-final`, `augment-final`, or `private` clause, there must be one *method-definition*. All other definition *class-clause*s create private fields. All remaining *expr*s are initialization expressions to be evaluated when the class is instantiated (see §5.3 "Creating Objects").

The result of a `class*` expression is a new class, derived from the specified superclass and implementing the specified interfaces. Instances of the class are created with the `instantiate` form or `make-object` procedure, as described in §5.3 "Creating Objects".

Each *class-clause* is (partially) macro-expanded to reveal its shapes. If a *class-clause* is a `begin` expression, its sub-expressions are lifted out of the `begin` and treated as *class-clause*s, in the same way that `begin` is flattened for top-level and embedded definitions.

Within a `class*` form for instances of the new class, `this` is bound to the object itself; `super-instantiate`, `super-make-object`, and `super-new` are bound to forms to initialize fields in the superclass (see §5.3 "Creating Objects"); `super` is available for calling superclass methods (see §5.2.3.1 "Method Definitions"); and `inner` is available for calling subclass augmentations of methods (see §5.2.3.1 "Method Definitions").

```
(class superclass-expr class-clause ...)
```

Like `class*`, but omits the `interface-expr`s, for the case that none are needed.

```
this
```

Within a `class*` form, refers to the current object (i.e., the object being initialized or whose method was called). Use outside the body of a `class*` form is a syntax error.

```
(inspect inspector-expr)
```

See `class*`; use outside the body of a `class*` form is a syntax error.

```
(init init-decl ...)
```

See `class*` and §5.2.1 "Initialization Variables"; use outside the body of a `class*` form is a syntax error.

```
(init-field init-decl ...)
```

See `class*`, §5.2.1 "Initialization Variables", and §5.2.2 "Fields"; use outside the body of a `class*` form is a syntax error.

```
(field field-decl ...)
```

See `class*` and §5.2.2 "Fields"; use outside the body of a `class*` form is a syntax error.

```
(inherit-field maybe-renamed ...)
```

See `class*` and §5.2.2 "Fields"; use outside the body of a `class*` form is a syntax error.

```
(init-rest id)
(init-rest)
```

See `class*` and §5.2.1 "Initialization Variables"; use outside the body of a `class*` form is a syntax error.

```
(public maybe-renamed ...)
```

See `class*` and §5.2.3.1 "Method Definitions"; use outside the body of a `class*` form is a syntax error.

```
(pubment maybe-renamed ...)
```

See `class*` and §5.2.3.1 "Method Definitions"; use outside the body of a `class*` form is a syntax error.

```
(public-final maybe-renamed ...)
```

See `class*` and §5.2.3.1 "Method Definitions"; use outside the body of a `class*` form is a syntax error.

```
(override maybe-renamed ...)
```

See `class*` and §5.2.3.1 "Method Definitions"; use outside the body of a `class*` form is a syntax error.

```
(overment maybe-renamed ...)
```

See `class*` and §5.2.3.1 "Method Definitions"; use outside the body of a `class*` form is a syntax error.

```
(override-final maybe-renamed ...)
```

See `class*` and §5.2.3.1 "Method Definitions"; use outside the body of a `class*` form is a syntax error.

```
(augment maybe-renamed ...)
```

See `class*` and §5.2.3.1 "Method Definitions"; use outside the body of a `class*` form is a syntax error.

```
(augride maybe-renamed ...)
```

See `class*` and §5.2.3.1 "Method Definitions"; use outside the body of a `class*` form is a syntax error.

```
(augment-final maybe-renamed ...)
```

See `class*` and §5.2.3.1 "Method Definitions"; use outside the body of a `class*` form is a syntax error.

`(private id ...)`

See `class*` and §5.2.3.1 “Method Definitions”; use outside the body of a `class*` form is a syntax error.

`(inherit maybe-renamed ...)`

See `class*` and §5.2.3.2 “Inherited and Superclass Methods”; use outside the body of a `class*` form is a syntax error.

`(inherit/super maybe-renamed ...)`

See `class*` and §5.2.3.2 “Inherited and Superclass Methods”; use outside the body of a `class*` form is a syntax error.

`(inherit/inner maybe-renamed ...)`

See `class*` and §5.2.3.2 “Inherited and Superclass Methods”; use outside the body of a `class*` form is a syntax error.

`(rename-super renamed ...)`

See `class*` and §5.2.3.2 “Inherited and Superclass Methods”; use outside the body of a `class*` form is a syntax error.

`(rename-inner renamed ...)`

See `class*` and §5.2.3.2 “Inherited and Superclass Methods”; use outside the body of a `class*` form is a syntax error.

`(public* (id expr) ...)`

Shorthand for `(begin (public id) ... (define id expr) ...)`.

`(pubment* (id expr) ...)`

Shorthand for `(begin (pubment id) ... (define id expr) ...)`.

`(public-final* (id expr) ...)`

Shorthand for `(begin (public-final id) ... (define id expr) ...)`.

`(override* (id expr) ...)`

Shorthand for `(begin (override id) ... (define id expr) ...)`.

`(overment* (id expr) ...)`

Shorthand for `(begin (overment id) ... (define id expr) ...)`.

`(override-final* (id expr) ...)`

Shorthand for `(begin (override-final id) ... (define id expr) ...)`.

`(augment* (id expr) ...)`

Shorthand for `(begin (augment id) ... (define id expr) ...)`.

`(augride* (id expr) ...)`

Shorthand for `(begin (augride id) ... (define id expr) ...)`.

`(augment-final* (id expr) ...)`

Shorthand for `(begin (augment-final id) ... (define id expr) ...)`.

`(private* (id expr) ...)`

Shorthand for `(begin (private id) ... (define id expr) ...)`.

```
(define/public id expr)
(define/public (id . formals) body ...+)
```

Shorthand for `(begin (public id) (define id expr))` or `(begin (public id) (define (id . formals) body ...+))`

```
(define/pubment id expr)
(define/pubment (id . formals) body ...+)
```

Shorthand for `(begin (pubment id) (define id expr))` or `(begin (pubment id) (define (id . formals) body ...+))`

```
(define/public-final id expr)
(define/public-final (id . formals) body ...+)
```

Shorthand for `(begin (public-final id) (define id expr))` or `(begin (public-final id) (define (id . formals) body ...+))`

```
(define/override id expr)
(define/override (id . formals) body ...+)
```

Shorthand for `(begin (override id) (define id expr))` or `(begin (override id) (define (id . formals) body ...+))`

```
(define/overment id expr)
(define/overment (id . formals) body ...+)
```

Shorthand for `(begin (overment id) (define id expr))` or `(begin (overment id) (define (id . formals) body ...+))`

```
(define/override-final id expr)
(define/override-final (id . formals) body ...+)
```

Shorthand for `(begin (override-final id) (define id expr))` or `(begin (override-final id) (define (id . formals) body ...+))`

```
(define/augment id expr)
(define/augment (id . formals) body ...+)
```

Shorthand for `(begin (augment id) (define id expr))` or `(begin (augment id) (define (id . formals) body ...+))`

```
(define/augride id expr)
(define/augride (id . formals) body ...+)
```

Shorthand for `(begin (augride id) (define id expr))` or `(begin (augride id) (define (id . formals) body ...+))`

```
(define/augment-final id expr)
(define/augment-final (id . formals) body ...+)
```

Shorthand for `(begin (augment-final id) (define id expr))` or `(begin (augment-final id) (define (id . formals) body ...+))`

```
(define/private id expr)
(define/private (id . formals) body ...+)
```

Shorthand for `(begin (private id) (define id expr))` or `(begin (private id) (define (id . formals) body ...+))`

```
(class/derived original-datum
  (name-id super-expr (interface-expr ...) deserialize-id-expr)
  class-clause
  ...)
```

Like `class*`, but includes a sub-expression to use used as the source for all syntax errors within the class definition. For example, `define-serializable-class` expands to `class/derived` so that error in the body of the class are reported in terms of `define-serializable-class` instead of `class`.

The *`original-datum`* is the original expression to use for reporting errors.

The *`name-id`* is used to name the resulting class; if it is `#f`, the class name is inferred.

The *`super-expr`*, *`interface-expr`*s, and *`class-clause`*s are as for `class*`.

If the *`deserialize-id-expr`* is not literally `#f`, then a serializable class is generated, and the result is two values instead of one: the class and a deserialize-info structure produced by `make-deserialize-info`. The *`deserialize-id-expr`* should produce a value suitable as the second argument to `make-serialize-info`, and it should refer to an export whose value is the deserialize-info structure.

Future optional forms may be added to the sequence that currently ends with *`deserialize-id-expr`*.

5.2.1 Initialization Variables

A class's initialization variables, declared with `init`, `init-field`, and `init-rest`, are instantiated for each object of a class. Initialization variables can be used in the initial value expressions of fields, default value expressions for initialization arguments, and in initialization expressions. Only initialization variables declared with `init-field` can be accessed from methods; accessing any other initialization variable from a method is a syntax error.

The values bound to initialization variables are

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

If an initialization argument is not provided for an initialization variable that has an associated `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, all of the fields, and all of the methods of the class. If multiple `default-value-expr`s are evaluated, they are evaluated from left to right. Object creation and field initialization are described in detail in §5.3 "Creating Objects".

If an initialization variable has no `default-value-expr`, then the object creation or superclass initialization call must supply an argument for the variable, otherwise the `exn:fail:object` exception is raised.

Initialization arguments can be provided by name or by position. The external name of an initialization variable can be used with `instantiate` or with the superclass initialization form. Those forms also accept by-position arguments. The `make-object` procedure and the superclass initialization procedure accept only by-position arguments.

Arguments provided by position are converted into by-name arguments using the order of `init` and `init-field` clauses and the order of variables within each clause. When a `instantiate` form provides both by-position and by-name arguments, the converted arguments are placed before by-name arguments. (The order can be significant; see also §5.3 "Creating Objects".)

Unless a class contains an `init-rest` clause, when the number of by-position arguments exceeds the number of declared initialization variables, the order of variables in the superclass (and so on, up the superclass chain) determines the by-name conversion.

If a class expression contains an `init-rest` clause, there must be only one, and it must be last. If it declares a variable, then the variable receives extra by-position initialization arguments as a list (similar to a dotted "rest argument" in a procedure). An `init-rest` variable can receive by-position initialization arguments that are left over from a by-name conversion for a derived class. When a derived class's superclass initialization provides even more by-position arguments, they are prefixed onto the by-position arguments accumulated so far.

If too few or too many by-position initialization arguments are provided to an object creation or superclass initialization, then the `exn:fail:object` exception is raised. Similarly, if extra by-position arguments are provided to a class with an `init-rest` clause, the `exn:fail:object` exception is raised.

Unused (by-name) arguments are to be propagated to the superclass, as described in §5.3

"Creating Objects". Multiple initialization arguments can use the same name if the class derivation contains multiple declarations (in different classes) of initialization variables with the name. See §5.3 "Creating Objects" for further details.

See also §5.2.3.3 "Internal and External Names" for information about internal and external names.

5.2.2 Fields

Each `field`, `init-field`, and non-method `define-values` clause in a class declares one or more new fields for the class. Fields declared with `field` or `init-field` are public. Public fields can be accessed and mutated by subclasses using `inherit-field`. Public fields are also accessible outside the class via `class-field-accessor` and mutable via `class-field-mutator` (see §5.4 "Field and Method Access"). Fields declared with `define-values` are accessible only within the class.

A field declared with `init-field` is both a public field and an initialization variable. See §5.2.1 "Initialization Variables" for information about initialization variables.

An `inherit-field` declaration makes a public field defined by a superclass directly accessible in the class expression. If the indicated field is not defined in the superclass, the `exn:fail:object` exception is raised when the class expression is evaluated. Every field in a superclass is present in a derived class, even if it is not declared with `inherit-field` in the derived class. The `inherit-field` clause does not control inheritance, but merely controls lexical scope within a class expression.

When an object is first created, all of its fields have the `#<undefined>` value (see §3.17 "Void and Undefined"). The fields of a class are initialized at the same time that the class's initialization expressions are evaluated; see §5.3 "Creating Objects" for more information.

See also §5.2.3.3 "Internal and External Names" for information about internal and external names.

5.2.3 Methods

[

Method Definitions]Method Definitions

Each `public`, `override`, `augment`, `pubment`, `overment`, `augride`, `public-final`, `override-final`, `augment-final`, and `private` clause in a class declares one or more method names. Each method name must have a corresponding `method-definition`. The order of `public`, etc. clauses and their corresponding definitions (among themselves, and with respect to other clauses in the class) does not matter.

As shown in the grammar for `class*`, a method definition is syntactically restricted to certain procedure forms, as defined by the grammar for `method-procedure`; in the last two forms of `method-procedure`, the body `id` must be one of the `id`s bound by `let-values` or `letrec-values`. A `method-procedure` expression is not evaluated directly. Instead, for each method, a class-specific method procedure is created; it takes an initial object argument, in addition to the arguments the procedure would accept if the `method-procedure` expression were evaluated directly. The body of the procedure is transformed to access methods and fields through the object argument.

A method declared with `public`, `pubment`, or `public-final` introduces a new method into a class. The method must not be present already in the superclass, otherwise the `exn:fail:object` exception is raised when the class expression is evaluated. A method declared with `public` can be overridden in a subclass that uses `override`, `overment`, or `override-final`. A method declared with `pubment` can be augmented in a subclass that uses `augment`, `augride`, or `augment-final`. A method declared with `public-final` cannot be overridden or augmented in a subclass.

A method declared with `override`, `overment`, or `override-final` overrides a definition already present in the superclass. If the method is not already present, the `exn:fail:object` exception is raised when the class expression is evaluated. A method declared with `override` can be overridden again in a subclass that uses `override`, `overment`, or `override-final`. A method declared with `overment` can be augmented in a subclass that uses `augment`, `augride`, or `augment-final`. A method declared with `override-final` cannot be overridden further or augmented in a subclass.

A method declared with `augment`, `augride`, or `augment-final` augments a definition already present in the superclass. If the method is not already present, the `exn:fail:object` exception is raised when the class expression is evaluated. A method declared with `augment` can be augmented further in a subclass that uses `augment`, `augride`, or `augment-final`. A method declared with `augride` can be overridden in a subclass that uses `override`, `overment`, or `override-final`. (Such an override merely replaces the augmentation, not the method that is augmented.) A method declared with `augment-final` cannot be overridden or augmented further in a subclass.

A method declared with `private` is not accessible outside the class expression, cannot be overridden, and never overrides a method in the superclass.

When a method is declared with `override`, `overment`, or `override-final`, then the superclass implementation of the method can be called using `super` form.

When a method is declared with `pubment`, `augment`, or `overment`, then a subclass augmenting method can be called using the `inner` form. The only difference between `public-final` and `pubment` without a corresponding `inner` is that `public-final` prevents the declaration of augmenting methods that would be ignored.

```
(super id arg ...)
(super id arg ... . arg-list-expr)
```

Always accesses the superclass method, independent of whether the method is overridden again in subclasses. Using the `super` form outside of `class*` is an syntax error. Each *arg* is as for #%app: either *arg-expr* or *keyword arg-expr*.

The second form is analogous to using `apply` with a procedure; the *arg-list-expr* must not be a parenthesized expression.

```
(inner default-expr id arg ...)
(inner default-expr id arg ... . arg-list-expr)
```

If the object's class does not supply an augmenting method, then *default-expr* is evaluated, and the *arg* expressions are not evaluated. Otherwise, the augmenting method is called with the *arg* results as arguments, and *default-expr* is not evaluated. If no `inner` call is evaluated for a particular method, then augmenting methods supplied by subclasses are never used. Using the `inner` form outside of `class*` is an syntax error.

The second form is analogous to using `apply` with a procedure; the *arg-list-expr* must not be a parenthesized expression.

[

Inherited and Superclass Methods]Inherited and Superclass Methods

Each `inherit`, `inherit/super`, `inherit/inner`, `rename-super`, and `rename-inner` clause declares one or more methods that are defined in the class, but must be present in the superclass. The `rename-super` and `rename-inner` declarations are rarely used, since `inherit/super` and `inherit/inner` provide the same access. Also, superclass and augmenting methods are typically accessed through `super` and `inner` in a class that also declares the methods, instead of through `inherit/super`, `inherit/inner`, `rename-super`, or `rename-inner`.

Method names declared with `inherit`, `inherit/super`, or `inherit/inner` access overriding declarations, if any, at run time. Method names declared with `inherit/super` can also be used with the `super` form to access the superclass implementation, and method names declared with `inherit/inner` can also be used with the `inner` form to access an augmenting method, if any.

Method names declared with `rename-super` always access the superclass's implementation at run-time. Methods declared with `rename-inner` access a subclass's augmenting method, if any, and must be called with the form

```
(id (lambda () default-expr) arg ...)
```

so that a `default-expr` is available to evaluate when no augmenting method is available. In such a form, `lambda` is a literal identifier to separate the `default-expr` from the `arg`. When an augmenting method is available, it receives the results of the `arg` expressions as arguments.

Methods that are present in the superclass but not declared with `inherit`, `inherit/super`, or `inherit/inner` or `rename-super` are not directly accessible in the class (through they can be called with `send`). Every public method in a superclass is present in a derived class, even if it is not declared with `inherit` in the derived class; the `inherit` clause does not control inheritance, but merely controls lexical scope within a class expression.

If a method declared with `inherit`, `inherit/super`, `inherit/inner`, `rename-super`, or `rename-inner` is not present in the superclass, the `exn:fail:object` exception is raised when the class expression is evaluated.

[

Internal and External Names]Internal and External Names

Each method declared with `public`, `override`, `augment`, `pubment`, `overment`, `augride`, `public-final`, `override-final`, `augment-final`, `inherit`, `inherit/super`, `inherit/inner`, `rename-super`, and `rename-inner` can have separate internal and external names when `(internal-id external-id)` is used for declaring the method. The internal name is used to access the method directly within the class expression (including within `super` or `inner` forms), while the external name is used with `send` and `generic` (see §5.4 "Field and Method Access"). If a single `id` is provided for a method declaration, the identifier is used for both the internal and external names.

Method inheritance, overriding, and augmentation are based external names, only. Separate internal and external names are required for `rename-super` and `rename-inner` (for historical reasons, mainly).

Each `init`, `init-field`, `field`, or `inherit-field` variable similarly has an internal and an external name. The internal name is used within the class to access the variable, while the external name is used outside the class when providing initialization arguments (e.g., to `instantiate`), inheriting a field, or accessing a field externally (e.g., with `class-field-accessor`). As for methods, when inheriting a field with `inherit-field`, the external name is matched to an external field name in the superclass, while the internal name is bound in the `class` expression.

A single identifier can be used as an internal identifier and an external identifier, and it is possible to use the same identifier as internal and external identifiers for different bindings. Furthermore, within a single class, a single name can be used as an external method name, an external field name, and an external initialization argument name. Overall, each internal identifier must be distinct from all other internal identifiers, each external method name must be distinct from all other method names, each external field name must be distinct from all other field names, and each initialization argument name must be distinct from all other initialization argument names

By default, external names have no lexical scope, which means, for example, that an external method name matches the same syntactic symbol in all uses of `send`. The `define-local-member-name` and `define-member-name` forms introduce scoped external names.

When a `class` expression is compiled, identifiers used in place of external names must be symbolically distinct (when the corresponding external names are required to be distinct), otherwise a syntax error is reported. When no external name is bound by `define-member-name`, then the actual external names are guaranteed to be distinct when `class` expression is evaluated. When any external name is bound by `define-member-name`, the `exn:fail:object` exception is raised by `class` if the actual external names are not distinct.

```
(define-local-member-name id ...)
```

Unless it appears as the top-level definition, binds each *`id`* so that, within the scope of the definition, each use of each *`id`* as an external name is resolved to a hidden name generated by the `define-local-member-name` declaration. Thus, methods, fields, and initialization arguments declared with such external-name *`id`*s are accessible only in the scope of the `define-local-member-name` declaration. As a top-level definition, `define-local-member-name` binds *`id`* to its symbolic form.

The binding introduced by `define-local-member-name` is a syntax binding that can be exported and imported with `module`s. Each execution of a `define-local-member-name` declaration generates a distinct hidden name (except as a top-level definitions). The `interface->method-names` procedure does not expose hidden names.

Examples:

```
(define-values (r o)
  (let ()
    (define-local-member-name m)
    (define c% (class object%
                 (define/public (m) 10)
                 (super-new)))
    (define o (new c%))

    (values (send o m)
            o)))

> r
10
> (send o m)
```

send: no such method: m for class: c%

`(define-member-name id key-expr)`

Maps a single external name to an external name that is determined by an expression. The value of *key-expr* must be the result of either a `member-name-key` expression or a `generate-member-key` call.

`(member-name-key identifier)`

Produces a representation of the external name for `id` in the environment of the `member-name-key` expression.

`(generate-member-key) → member-name-key?`

Produces a hidden name, just like the binding for `define-local-member-name`.

```
(member-name-key? v) → boolean?
  v : any/c
```

Returns `#t` for values produced by `member-name-key` and `generate-member-key`, `#f` otherwise.

```
(member-name-key=? a-key b-key) → boolean?
  a-key : member-name-key?
  b-key : member-name-key?
```

Produces `#t` if member-name keys *a-key* and *b-key* represent the same external name, `#f` otherwise.

```
(member-name-key-hash-code a-key) → integer?
  a-key : member-name-key?
```

Produces an integer hash code consistent with `member-name-key=?` comparisons, analogous to `equal-hash-code`.

Examples:

```
(define (make-c% key)
  (define-member-name m key)
  (class object%
    (define/public (m) 10)
```

```
    (super-new)))

> (send (new (make-c% (member-name-key m))) m)
10
> (send (new (make-c% (member-name-key p))) m)
send: no such method: m for class: eval:11:0
> (send (new (make-c% (member-name-key p))) p)
10

(define (fresh-c%)
  (let ([key (generate-member-key)])
    (values (make-c% key) key)))

(define-values (fc% key) (fresh-c%))

> (send (new fc%) m)
send: no such method: m for class: eval:11:0
> (let ()
    (define-member-name p key)
    (send (new fc%) p))
10
```

5.3 Creating Objects

The `make-object` procedure creates a new object with by-position initialization arguments, the `new` form creates a new object with by-name initialization arguments, and the `instantiate` form creates a new object with both by-position and by-name initialization arguments.

All fields in the newly created object are initially bound to the special `#<undefined>` value (see §3.17 "Void and Undefined"). Initialization variables with default value expressions (and no provided value) are also initialized to `#<undefined>`. After argument values are assigned to initialization variables, expressions in `field` clauses, `init-field` clauses with no provided argument, `init` clauses with no provided argument, private field definitions, and other expressions are evaluated. Those expressions are evaluated as they appear in the class expression, from left to right.

Sometime during the evaluation of the expressions, superclass-declared initializations must be executed once by using the `super-make-object` procedure, `super-new` form, or `super-instantiate` form.

By-name initialization arguments to a class that have no matching initialization variable are implicitly added as by-name arguments to a `super-make-object`, `super-new`, or `super-instantiate` invocation, after the explicit arguments. If multiple initialization arguments are provided for the same name, the first (if any) is used, and the unused arguments are propagated to the superclass. (Note that converted by-position arguments are always placed before explicit by-name arguments.) The initialization procedure for the `object%` class accepts zero initialization arguments; if it receives any by-name initialization arguments, then `exn:fail:object` exception is raised.

If the end of initialization is reached for any class in the hierarchy without invoking the superclass's initialization, the `exn:fail:object` exception is raised. Also, if superclass initialization is invoked more than once, the `exn:fail:object` exception is raised.

Fields inherited from a superclass are not initialized until the superclass's initialization procedure is invoked. In contrast, all methods are available for an object as soon as the object is created; the overriding of methods is not affect by initialization (unlike objects in C++).

```
(make-object class init-v ...) → object?
  class : class?
  init-v : any/c
```

Creates an instance of *class*. The *init-v*s are passed as initialization arguments, bound to the initialization variables of *class* for the newly created object as described in §5.2.1 "Initialization Variables". If *class* is not a class, the `exn:fail:contract` exception is raised.

```
(new class-expr (id by-name-expr) ...)
```

Creates an instance of the value of *class-expr* (which must be a class), and the value of each *by-name-expr* is provided as a by-name argument for the corresponding *id*.

```
(instantiate class-expr (by-pos-expr ...) (id by-name-expr) ...)
```

Creates an instance of the value of *class-expr* (which must be a class), and the values of the *by-pos-expr*s are provided as by-position initialization arguments. In addition, the value of each *by-name-expr* is provided as a by-name argument for the corresponding *id*.

```
super-make-object
```

Produces a procedure that takes by-position arguments an invokes superclass initialization. See §5.3 "Creating Objects" for more information.

```
(super-instantiate (by-pos-expr ...) (id by-expr ...) ...)
```

Invokes superclass initialization with the specified by-position and by-name arguments. See §5.3 "Creating Objects" for more information.

```
(super-new (id by-name-expr ...) ...)
```

Invokes superclass initialization with the specified by-name arguments. See §5.3 "Creating Objects" for more information.

5.4 Field and Method Access

In expressions within a class definition, the initialization variables, fields, and methods of the class all part of the environment. Within a method body, only the fields and other methods of the class can be referenced; a reference to any other class-introduced identifier is a syntax error. Elsewhere within the class, all class-introduced identifiers are available, and fields and initialization variables can be mutated with `set!`.

5.4.1 Methods

Method names within a class can only be used in the procedure position of an application expression; any other use is a syntax error.

To allow methods to be applied to lists of arguments, a method application can have the following form:

```
(method-id arg ... . arg-list-expr)
```

This form calls the method in a way analogous to `(apply method-id arg ... arg-list-expr)`. The `arg-list-expr` must not be a parenthesized expression.

Methods are called from outside a class with the `send` and `send/apply` forms.

```
(send obj-expr method-id arg ...)
(send obj-expr method-id arg ... . arg-list-expr)
```

Evaluates *obj-expr* to obtain an object, and calls the method with (external) name *method-id* on the object, providing the *arg* results as arguments. Each *arg* is as for `#%app`: either *arg-expr* or *keyword arg-expr*. In the second form, *arg-list-expr* cannot be a parenthesized expression.

If *obj-expr* does not produce an object, the `exn:fail:contract` exception is raised. If the object has no public method named *method-id*, the `exn:fail:object` exception is raised.

```
(send/apply obj-expr method-id arg ... arg-list-expr)
```

Like the dotted form of send, but *arg-list-expr* can be any expression.

```
(send* obj-expr msg ...)

msg = (method-id arg ...)
    | (method-id arg ... . arg-list-expr)
```

Calls multiple methods (in order) of the same object. Each *msg* corresponds to a use of send.

For example,

```
(send* edit (begin-edit-sequence)
            (insert "Hello")
            (insert #\newline)
            (end-edit-sequence))
```

is the same as

```
(let ([o edit])
  (send o begin-edit-sequence)
  (send o insert "Hello")
  (send o insert #\newline)
  (send o end-edit-sequence))
```

```
(with-method ((id (obj-expr method-id)) ...)
  body ...+)
```

Extracts methods from an object and binds a local name that can be applied directly (in the same way as declared methods within a class) for each method. The each *obj-expr* must produce an object, which must have a public method named by the corresponding *method-id*. The corresponding *id* is bound so that it can be applied directly (see §5.4.1 "Methods").

Example:

```
(let ([s (new stack%)])
  (with-method ([push (s push!)]
                [pop (s pop!)])
    (push 10)
    (push 9)
    (pop)))
```

is the same as

```
(let ([s (new stack%)])
  (send s push! 10)
  (send s push! 9)
  (send s pop!))
```

5.4.2 Fields

```
(get-field id obj-expr)
```

Extracts the field with (external) name *id* from the value of *obj-expr*.

If *obj-expr* does not produce an object, the `exn:fail:contract` exception is raised. If the object has no *id* method, the `exn:fail:object` exception is raised.

```
(field-bound? id obj-expr)
```

Produces `#t` if the object result of *obj-expr* has an field with (external) name *id*, `#f` otherwise.

If *obj-expr* does not produce an object, the `exn:fail:contract` exception is raised.

```
(class-field-accessor class-expr field-id)
```

Returns an accessor procedure that takes an instance of the class produced by *class-expr* and returns the value of the object's field with (external) name *field-id*.

If `obj-expr` does not produce an object, the `exn:fail:contract` exception is raised. If the object has no *field-id* field, the `exn:fail:object` exception is raised.

```
(class-field-mutator class-expr field-id)
```

Returns a mutator procedure that takes an instance of the class produced by *class-expr* and a value, and sets the value of the object's field with (external) name *field-id* to the given value. The result is `#<void>`.

If `obj-expr` does not produce an object, the `exn:fail:contract` exception is raised. If the object has no *field-id* field, the `exn:fail:object` exception is raised.

5.4.3 Generics

A *generic* can be used instead of a method name to avoid the cost of relocating a method by name within a class.

```
(generic class-or-interface-expr id)
```

Produces a generic that works on instances of the class or interface produced by `class-or-interface-expr` (or an instance of a class/interface derived from `class-or-interface`) to call the method with (external) name `id`.

If `class-or-interface-expr` does not produce a class or interface, the `exn:fail:contract` exception is raised. If the resulting class or interface does not contain a method named `id`, the `exn:fail:object` exception is raised.

```
(send-generic obj-expr generic-expr arg ...)
(send-generic obj-expr generic-expr arg ... . arg-list-expr)
```

Calls a method of the object produced by `obj-expr` as indicated by the generic produced by `generic-expr`. Each `arg` is as for #%app: either `arg-expr` or `keyword arg-expr`. The second form is analogous to calling a procedure with `apply`, where `arg-list-expr` is not a parenthesized expression.

If `obj-expr` does not produce a object, or if `generic-expr` does not produce a generic, the `exn:fail:contract` exception is raised. If the result of `obj-expr` is not an instance of the class or interface encapsulated by the result of `generic-expr`, the `exn:fail:object` exception is raised.

```
(make-generic type method-name) → generic?
  type : (or/c class? interface?)
  method-name : symbol?
```

Like the `generic` form, but as a procedure that accepts a symbolic method name.

5.5 Mixins

```
(mixin (interface-expr ...) (interface-expr ...)
  class-clause ...)
```

Produces a *mixin*, which is a procedure that encapsulates a class extension, leaving the superclass unspecified. Each time that a mixin is applied to a specific superclass, it produces a new derived class using the encapsulated extension.

The given class must implement interfaces produced by the first set of *interface-expr*s. The result of the procedure is a subclass of the given class that implements the interfaces produced by the second set of *interface-expr*s. The *class-clause*s are as for `class*`, to define the class extension encapsulated by the mixin.

Evaluation of a `mixin` form checks that the *class-clause*s are consistent with both sets of *interface-expr*s.

5.6 Traits

```
(require scheme/trait)
```

The bindings documented in this section are provided by the `scheme/trait` library, not `scheme/base` or `scheme`.

A *trait* is a collection of methods that can be converted to a mixin and then applied to a class. Before a trait is converted to a mixin, the methods of a trait can be individually renamed, and multiple traits can be merged to form a new trait.

```
(trait trait-clause ...)

trait-clause = (public maybe-renamed ...)
             | (pubment maybe-renamed ...)
             | (public-final maybe-renamed ...)
             | (override maybe-renamed ...)
             | (overment maybe-renamed ...)
             | (override-final maybe-renamed ...)
             | (augment maybe-renamed ...)
             | (augride maybe-renamed ...)
             | (augment-final maybe-renamed ...)
             | (inherit maybe-renamed ...)
             | (inherit/super maybe-renamed ...)
             | (inherit/inner maybe-renamed ...)
             | method-definition
             | (field field-declaration ...)
             | (inherit-field maybe-renamed ...)
```

Creates a trait. The body of a `trait` form is similar to the body of a `class*` form, but restricted to non-private method definitions. In particular, the grammar of *maybe-renamed*, *method-definition*, and *field-declaration* are the same as for `class*`, and every *method-definition* must have a corresponding declaration (one of `public`, `override`, etc.). As in `class`, uses of method names in direct calls, `super` calls, and `inner` calls depend on bringing method names into scope via `inherit`, `inherit/super`, `inherit/inner`, and other method declarations in the same trait; an exception, compared to `class` is that `overment` binds a method name only in the corresponding method, and not in other methods of the same trait. Finally, macros such as `public*` and `define/public` work in `trait` as in `class`.

External identifiers in `trait`, `trait-exclude`, `trait-exclude-field`, `trait-alias`, `trait-rename`, and `trait-rename-field` forms are subject to binding via `define-member-name` and `define-local-member-name`. Although `private` methods or fields are not allowed in a `trait` form, they can be simulated by using a `public` or *`field`* declaration and a name whose scope is limited to the `trait` form.

```
(trait? v) → boolean?
  v : any/c
```

Returns `#t` if *`v`* is a trait, `#f` otherwise.

```
(trait->mixin tr) → (class? . -> . class?)
  tr : trait?
```

Converts a trait to a mixin, which can be applied to a class to produce a new class. An expression of the form

```
(trait->mixin
 (trait
   trait-clause ...))
```

is equivalent to

```
(lambda (%)
  (class %
    trait-clause ...
    (super-new)))
```

Normally, however, a trait's methods are changed and combined with other traits before converting to a mixin.

```
(trait-sum tr ...+) → trait?
  tr : trait?
```

Produces a trait that combines all of the methods of the given *`tr`*s. For example,

```
(define t1
 (trait
```

```
    (define/public (m1) 1)))
  (define t2
   (trait
    (define/public (m2) 2)))
  (define t3 (trait-sum t1 t2))
```

creates a trait `t3` that is equivalent to

```
  (trait
   (define/public (m1) 1)
   (define/public (m2) 2))
```

but `t1` and `t2` can still be used individually or combined with other traits.

When traits are combined with `trait-sum`, the combination drops `inherit`, `inherit/super`, `inherit/inner`, and `inherit-field` declarations when a definition is supplied for the same method or field name by another trait. The `trait-sum` operation fails (the `exn:fail:contract` exception is raised) if any of the traits to combine define a method or field with the same name, or if an `inherit/super` or `inherit/inner` declaration to be dropped is inconsistent with the supplied definition. In other words, declaring a method with `inherit`, `inherit/super`, or `inherit/inner`, does not count as defining the method; at the same time, for example, a trait that contains an `inherit/super` declaration for a method `m` cannot be combined with a trait that defines `m` as `augment`, since no class could satisfy the requirements of both `augment` and `inherit/super` when the trait is later converted to a mixin and applied to a class.

```
(trait-exclude trait-expr id)
```

Produces a new trait that is like the trait result of *`trait-expr`*, but with the definition of a method named by *`id`* removed; as the method definition is removed, either a `inherit`, `inherit/super`, or `inherit/inner` declaration is added:

- A method declared with `public`, `pubment`, or `public-final` is replaced with a `inherit` declaration.
- A method declared with `override` or `override-final` is replaced with a `inherit/super` declaration.
- A method declared with `augment`, `augride`, or `augment-final` is replaced with a `inherit/inner` declaration.
- A method declared with `overment` is not replaced with any `inherit` declaration.

If the trait produced by *`trait-expr`* has no method definition for *`id`*, the `exn:fail:contract` exception is raised.

```
(trait-exclude-field trait-expr id)
```

Produces a new trait that is like the trait result of *trait-expr*, but with the definition of a field named by *id* removed; as the field definition is removed, an `inherit-field` declaration is added.

```
(trait-alias trait-expr id new-id)
```

Produces a new trait that is like the trait result of *trait-expr*, but the definition and declaration of the method named by *id* is duplicated with the name *new-id*. The consistency requirements for the resulting trait are the same as for `trait-sum`, otherwise the `exn:fail:contract` exception is raised. This operation does not rename any other use of *id*, such as in method calls (even method calls to `identifier` in the cloned definition for *new-id*).

```
(trait-rename trait-expr id new-id)
```

Produces a new trait that is like the trait result of *trait-expr*, but all definitions and references to methods named *id* are replaced by definitions and references to methods named by *new-id*. The consistency requirements for the resulting trait is the same as for `trait-sum`, otherwise the `exn:fail:contract` exception is raised.

```
(trait-rename-field trait-expr id new-id)
```

Produces a new trait that is like the trait result of *trait-expr*, but all definitions and references to fields named *id* are replaced by definitions and references to fields named by *new-id*. The consistency requirements for the resulting trait is the same as for `trait-sum`, otherwise the `exn:fail:contract` exception is raised.

5.7 Object and Class Contracts

```
(object-contract member-spec ...)
```

```
          member-spec = (method-id method-contract)
                      | (field field-id contract-expr)

      method-contract = (-> dom ... range)
                      | (->* (mandatory-dom ...)
                             (optional-dom ...)
                             rest
                             range)
                      | (->d (mandatory-dependent-dom ...)
                             (optional-dependent-dom ...)
                             dependent-rest
                             pre-cond
                             dep-range)

                  dom = dom-expr
                      | keyword dom-expr

                range = range-expr
                      | (values range-expr ...)
                      | any

        mandatory-dom = dom-expr
                      | keyword dom-expr

         optional-dom = dom-expr
                      | keyword dom-expr

                 rest =
                      | #:rest rest-expr

mandatory-dependent-dom = [id dom-expr]
                      | keyword [id dom-expr]

 optional-dependent-dom = [id dom-expr]
                      | keyword [id dom-expr]

       dependent-rest =
                      | #:rest id rest-expr

             pre-cond =
                      | #:pre-cond boolean-expr

            dep-range = any
                      | [id range-expr] post-cond
                      | (values [id range-expr] ...) post-cond

            post-cond =
                      | #:post-cond boolean-expr
```

Produces a contract for an object.

Each of the contracts for a method has the same semantics as the corresponding function contract, but the syntax of the method contract must be written directly in the body of the object-contract—much like the way that methods in class definitions use the same syntax as regular function definitions, but cannot be arbitrary procedures. The only exception is that `->d` contracts implicitly bind `this` to the object itself.

```
mixin-contract : contract?
```

A function contract that recognizes mixins. It guarantees that the input to the function is a class and the result of the function is a subclass of the input.

```
(make-mixin-contract type ...) → contract?
  type : (or/c class? interface?)
```

Produces a function contract that guarantees the input to the function is a class that implements/subclasses each *type*, and that the result of the function is a subclass of the input.

```
(is-a?/c type) → flat-contract?
  type : (or/c class? interface?)
```

Accepts a class or interface and returns a flat contract that recognizes objects that instantiate the class/interface.

```
(implementation?/c interface) → flat-contract?
  interface : interface?
```

Returns a flat contract that recognizes classes that implement *interface*.

```
(subclass?/c class) → flat-contract?
  class : class?
```

Returns a flat-contract that recognizes classes that are subclasses of *class*.

5.8 Object Serialization

```
(define-serializable-class* class-id superclass-expr
                                     (interface-expr ...)
  class-clause ...)
```

Binds *class-id* to a class, where *superclass-expr*, the *interface-expr*s, and the *class-clause*s are as in class*.

This forms can only be used at the top level, either within a module or outside. The *class-id* identifier is bound to the new class, and deserialize-info:@scheme [*class-id*] is also defined; if the definition is within a module, then the latter is provided from the module.

Serialization for the class works in one of two ways:

- If the class implements the built-in interface externalizable<%>, then an object is serialized by calling its externalize method; the result can be anything that is serializable (but, obviously, should not be the object itself). Deserialization creates an instance of the class with no initialization arguments, and then calls the object's internalize method with the result of externalize (or, more precisely, a deserialized version of the serialized result of a previous call).

  To support this form of serialization, the class must be instantiable with no initialization arguments. Furthermore, cycles involving only instances of the class (and other such classes) cannot be serialized.

- If the class does not implement externalizable<%>, then every superclass of the class must be either serializable or transparent (i.e,. have #f as its inspector). Serialization and deserialization are fully automatic, and may involve cycles of instances.

  To support cycles of instances, deserialization may create an instance of the call with all fields as the undefined value, and then mutate the object to set the field values. Serialization support does not otherwise make an object's fields mutable.

In the second case, a serializable subclass can implement externalizable<%>, in which case the externalize method is responsible for all serialization (i.e., automatic serialization is lost for instances of the subclass). In the first case, all serializable subclasses implement externalizable<%>, since a subclass implements all of the interfaces of its parent class.

In either case, if an object is an immediate instance of a subclass (that is not itself serializable), the object is serialized as if it was an immediate instance of the serializable class. In particular, overriding declarations of the externalize method are ignored for instances of non-serializable subclasses.

```
(define-serializable-class class-id superclass-expr
  class-clause ...)
```

Like define-serializable-class*, but with not interface expressions (analogous to class).

```
externalizable<%> : interface?
```

The `externalizable<%>` interface includes only the `externalize` and `internalize` methods. See `define-serializable-class*` for more information.

5.9 Object, Class, and Interface Utilities

```
(object? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is an object, `#f` otherwise.

```
(class? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a class, `#f` otherwise.

```
(interface? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is an interface, `#f` otherwise.

```
(generic? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a generic, `#f` otherwise.

```
(object=? a b) → eq?
  a : object?
  b : object?
```

Determines if two objects are the same object, or not; this procedure uses `"eq?"`, but also works properly with contracts.

```
(object->vector object [opaque-v]) → vector?
  object : object?
  opaque-v : any/c = #f
```

Returns a vector representing *object* that shows its inspectable fields, analogous to struct->vector.

```
(class->interface class) → interface?
  class : class?
```

Returns the interface implicitly defined by *class*.

```
(object-interface object) → interface?
  object : object?
```

Returns the interface implicitly defined by the class of *object*.

```
(is-a? v type) → boolean?
  v : any/c
  type : (or/c interface? class?)
```

Returns #t if *v* is an instance of a class *type* or a class that implements an interface *type*, #f otherwise.

```
(subclass? v class) → boolean?
  v : any/c
  class : class?
```

Returns #t if *v* is a class derived from (or equal to) *class*, #f otherwise.

```
(implementation? v interface) → boolean?
  v : any/c
  interface : interface?
```

Returns #t if *v* is a class that implements *interface*, #f otherwise.

```
(interface-extension? v interface) → boolean?
  v : any/c
  interface : interface?
```

Returns #t if *v* is an interface that extends *interface*, #f otherwise.

```
(method-in-interface? sym interface) → boolean?
  sym : symbol?
  interface : interface?
```

Returns `#t` if *interface* (or any of its ancestor interfaces) includes a member with the name *sym*, `#f` otherwise.

```
(interface->method-names interface) → (listof symbol?)
  interface : interface?
```

Returns a list of symbols for the method names in *interface*, including methods inherited from superinterfaces, but not including methods whose names are local (i.e., declared with `define-local-member-names`).

```
(object-method-arity-includes? object
                               sym
                               cnt)   → boolean?
  object : object?
  sym : symbol?
  cnt : nonnegative-exact-integer?
```

Returns `#t` if *object* has a method named *sym* that accepts *cnt* arguments, `#f` otherwise.

```
(field-names object) → (listof symbol?)
  object : object?
```

Returns a list of all of the names of the fields bound in *object*, including fields inherited from superinterfaces, but not including fields whose names are local (i.e., declared with `define-local-member-names`).

```
(object-info object) → (or/c class? false/c) boolean?
  object : any/c
```

Returns two values, analogous to the return values of `struct-info`: K%

- `class`: a class or `#f`; the result is `#f` if the current inspector does not control any class for which the *object* is an instance.
- `skipped?`: `#f` if the first result corresponds to the most specific class of *object*, `#t` otherwise.

```
(class-info class)
```

```
→ symbol?
  nonnegative-exact-integer?
  (listof symbol?)
  (any/c nonnegative-exact-integer? . -> . any/c)
  (any/c nonnegative-exact-integer? any/c . -> . any/c)
  (or/c class? false/c)
  boolean?
 class : class?
```

Returns seven values, analogous to the return values of `struct-type-info`:

- `name`: the class's name as a symbol;
- `field-cnt`: the number of fields (public and private) defined by the class;
- `field-name-list`: a list of symbols corresponding to the class's public fields; this list can be larger than `field-k` because it includes inherited fields;
- `field-accessor`: an accessor procedure for obtaining field values in instances of the class; the accessor takes an instance and a field index between `0` (inclusive) and `field-cnt` (exclusive);
- `field-mutator`: a mutator procedure for modifying field values in instances of the class; the mutator takes an instance, a field index between `0` (inclusive) and `field-cnt` (exclusive), and a new field value;
- `super-class`: a class for the most specific ancestor of the given class that is controlled by the current inspector, or `#f` if no ancestor is controlled by the current inspector;
- `skipped?`: `#f` if the sixth result is the most specific ancestor class, `#t` otherwise.

```
(struct (exn:fail:object exn:fail) ())
```

Raised for `class`-related failures, such as attempting to call a method that is not supplied by an object.

6 Units

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.

```
(require scheme/unit)
```

The bindings documented in this section are provided by the `scheme/unit` and `scheme` libraries, but not `scheme/base`.The `scheme/unit` module name can be used as a language name with `#lang`; see §6.9 "Single-Unit Modules".

6.1 Creating Units

```
(unit
  (import tagged-sig-spec ...)
  (export tagged-sig-spec ...)
  init-depends-decl
  unit-body-expr-or-defn
  ...)

  tagged-sig-spec = sig-spec
                  | (tag id sig-spec)

         sig-spec = sig-id
                  | (prefix id sig-spec)
                  | (rename sig-spec (id id) ...)
                  | (only sig-spec id ...)
                  | (except sig-spec id ...)

init-depends-decl =
                  | (init-depend tagged-sig-id ...)

    tagged-sig-id = sig-id
                  | (tag id sig-id)
```

Produces a unit that encapsulates its `unit-body-expr-or-defn`s. Expressions in the `unit` body can refer to identifiers bound by the `sig-spec`s of the `import` clause, and the body must include one definition for each identifier of a `sig-spec` in the `export` clause. An identifier that is exported cannot be `set!`ed in either the defining unit or in importing units, although the implicit assignment to initialize the variable may be visible as a mutation.

Each import or export `sig-spec` ultimately refers to a `sig-id`, which is an identifier that is bound to a signature by `define-signature`.

In a specific import or export position, the set of identifiers bound or required by a particular `sig-id` can be adjusted in a few ways:

- `(prefix id sig-spec)` as an import binds the same as `sig-spec`, except that each binding is prefixed with `id`. As an export, this form causes definitions using the `id` prefix to satisfy the exports required by `sig-spec`.
- `(rename sig-spec (id id) ...)` as an import binds the same as `sig-spec`, except that the first `id` is used for the binding instead of the second `id` (where `sig-spec` by itself must imply a binding for the second `id`). As an export, this form causes a definition for the first `id` to satisfy the export named by the second `id` in `sig-spec`.
- `(only sig-spec id ...)` as an import binds the same as `sig-spec`, but restricted to just the listed `id`s (where `sig-spec` by itself must imply a binding for each `id`). This form is not allowed for an export.
- `(except sig-spec id ...)` as an import binds the same as `sig-spec`, but excluding all listed `id`s (where `sig-spec` by itself must imply a binding for each `id`). This form is not allowed for an export.

As suggested by the grammar, these adjustments to a signature can be nested arbitrarily.

A unit's declared imports are matched with actual supplied imports by signature. That is, the order in which imports are suppplied to a unit when linking is irrelevant; all that matters is the signature implemented by each supplied import. One actual import must be provided for each declared import. Similarly, when a unit implements multiple signatures, the order of the export signatures does not matter.

To support multiple imports or exports for the same signature, an import or export can be tagged using the form `(tag id sig-spec)`. When an import declaration of a unit is tagged, then one actual import must be given the same tag (with the same signature) when the unit is linked. Similarly, when an export declaration is tagged for a unit, then references to that particular export must explicitly use the tag.

A unit is prohibited syntactically from importing two signatures that are not distinct, unless they have different tags; two signatures are *distinct* only if when they share no ancestor through `extends`. The same syntactic constraint applies to exported signatures. In addition, a unit is prohibited syntactically from importing the same identifier twice (after renaming and other transformations on a *sig-spec*), exporting the same identifier twice (again, after renaming), or exporting an identifier that is imported.

When units are linked, the bodies of the linked units are executed in an order that is specified at the linking site. An optional `(init-depend tagged-sig-id ...)` declaration constrains the allowed orders of linking by specifying that the current unit must be initialized after the unit that supplies the corresponding import. Each *tagged-sig-id* in an `init-depend` declaration must have a corresponding import in the `import` clause.

```
(define-signature id extension-decl
  (sig-elem ...))

extension-decl =
               | extends sig-id

      sig-elem = id
               | (define-syntaxes (id ...) expr)
               | (define-values (value-id ...) expr)
               | (open sig-spec)
               | (sig-form-id . datum)
```

Binds an identifier to a signature that specifies a group of bindings for import or export:

- Each *id* in a signature declaration means that a unit implementing the signature must supply a variable definition for the *id*. That is, *id* is available for use in units importing the signature, and *id* must be defined by units exporting the signature.
- Each `define-syntaxes` form in a signature declaration introduces a macro to that is available for use in any unit that imports the signature. Free variables in the definition's *expr* refer to other identifiers in the signature first, or the context of the `define-signature` form if the signature does not include the identifier.
- Each `define-values` form in a signature declaration introduces code that effectively prefixes every unit that imports the signature. Free variables in the definition's *expr* are treated the same as for `define-syntaxes`.
- Each `(open sig-spec)` adds to the signature everything specified by *sig-spec*.
- Each `(sig-form-id . datum)` extends the signature in a way that is defined by *sig-form-id*, which must be bound by `define-signature-form`. One such binding is for `struct`.

When a `define-signature` form includes a `extends` clause, then the define signature automatically includes everything in the extended signature. Furthermore, any implementation of the new signature can be used as an implementation of the extended signature.

`(open sig-spec)`

Allowed only in a *sig-elem*; see `define-signature`.

`(only sig-spec id ...)`

Allowed only in a *sig-spec*; see `unit`.

`(except sig-spec id ...)`

Allowed only in a *sig-spec*; see `unit`.

`(rename sig-spec (id id) ...)`

Allowed only in a *sig-spec*; see `unit`.

`(prefix id sig-spec)`

Allowed only in a *sig-spec*; see `unit`.

`(import tagged-sig-spec ...)`

Allowed only in certain forms; see, for example, `unit`.

`(export tagged-sig-spec ...)`

Allowed only in certain forms; see, for example, `unit`.

`(link linkage-decl ...)`

Allowed only in certain forms; see, for example, `compound-unit`.

`(tag id sig-spec)`
`(tag id sig-id)`

Allowed only in certain forms; see, for example, `unit`.

`(init-depend tagged-sig-id ...)`

Allowed only in a init-depend-decl; see unit.

```
extends
```

This form is allowed only within define-signature.

6.2 Invoking Units

```
(invoke-unit unit-expr)
(invoke-unit unit-expr (import tagged-sig-spec ...))
```

Invokes the unit produced by *unit-expr*. For each of the unit's imports, the invoke-unit expression must contain a *tagged-sig-spec* in the import clause; see unit for the grammar of *tagged-sig-spec*. If the unit has no imports, the import clause can be omitted.

When no *tagged-sig-spec*s are provided, *unit-expr* must produce a unit that expect no imports. To invoke the unit, all bindings are first initialized to the #<undefined> value. Next, the unit's body definitions and expressions are evaluated in order; in the case of a definition, evaluation sets the value of the corresponding variable(s). Finally, the result of the last expression in the unit is the result of the invoke-unit expression.

Each supplied *tagged-sig-spec* takes bindings from the surrounding context and turns them into imports for the invoked unit. The unit need not declare an imports for evey provided *tagged-sig-spec*, but one *tagged-sig-spec* must be provided for each declared import of the unit. For each variable identifier in each provided *tagged-sig-spec*, the value of the identifier's binding in the surrounding context is used for the corresponding import in the invoked unit.

```
(define-values/invoke-unit unit-expr
  (import tagged-sig-spec ...)
  (export tagged-sig-spec ...))
```

Like invoke-unit, but the values of the unit's exports are copied to new bindings.

The unit produced by *unit-expr* is linked and invoked as for invoke-unit. In addition, the export clause is treated as a kind of import into the local definition context. That is, for every binding that would be available in a unit that used the export clauses's *tagged-sig-spec* as an import, a definition is generated for the context of the define-values/invoke-unit form.

6.3 Linking Units and Creating Compound Units

```
(compound-unit
  (import link-binding ...)
  (export tagged-link-id ...)
  (link linkage-decl ...))

  link-binding = (link-id : tagged-sig-id)

tagged-link-id = (tag id link-id)
               | link-id

  linkage-decl = ((link-binding ...) unit-expr tagged-link-id)
```

Links several units into one new compound unit without immediately invoking any of the linked units. The *unit-expr*s in the link clause determine the units to be linked in creating the compound unit. The *unit-expr*s are evaluated when the compound-unit form is evaluated.

The import clause determines the imports of the compound unit. Outside the compound unit, these imports behave as for a plain unit; inside the compound unit, they are propagated to some of the linked units. The export clause determines the exports of the compound unit. Again, outside the compound unit, these exports are trested the same as for a plain unit; inside the compound unit, they are drawn from the exports of the linked units. Finally, the left-hand and right-hand parts of each declaration in the link clause specify how the compound unit's imports and exports are propagated to the linked units.

Individual elements of an imported or exported signature are not available within the compound unit. Instead, imports and exports are connected at the level of whole signatures. Each specific import or export (i.e., an instance of some signature, possibly tagged) is given a *link-id* name. Specifically, a *link-id* is bound by the import clause or the left-hand part of an declaration in the link clause. A bound *link-id* is referenced in the right-hand part of a declaration in the link clause or by the export clause.

The left-hand side of a link declaration gives names to each expected export of the unit produced by the corresponding *unit-expr*. The actual unit may export additional signatures, and it may export an extension of a specific signature instead of just the specified one. If the unit does not export one of the specified signatures (with the specified tag, if any), the exn:fail:contract exception is raised when the compound-unit form is evaluated.

The right-hand side of a link declaration specifies the imports to be supplied to the unit produced by the corresponding *unit-expr*. The actual unit may import fewer signatures, and it may import a signature that is extended by the specified one. If the unit imports a signature (with a particular tag) that is not included in the supplied imports, the exn:fail:contract exception is raised when the compound-unit form is evaluated. Each *link-id* supplied as an import must be bound either in the `import` clause or in some declaration within the `link` clause.

The order of declarations in the `link` clause determines the order of invocation of the linked units. When the compound unit is invoked, the unit produced by the first *`unit-expr`* is invoked first, then the second, and so on. If the order specified in the `link` clause is inconsistent with `init-depend` declarations of the actual units, then the `exn:fail:contract` exception is raised when the `compound-unit` form is evaluated.

6.4 Inferred Linking

```
(define-unit unit-id
  (import tagged-sig-spec ...)
  (export tagged-sig-spec ...)
  init-depends-decl
  unit-body-expr-or-defn
  ...)
```

Binds *`unit-id`* to both a unit and static information about the unit.

Evaluating a reference to an *`unit-id`* bound by `define-unit` produces a unit, just like evaluating an `id` bound by `(define id (unit ...))`. In addition, however, *`unit-id`* can be used in `compound-unit/infer`. See `unit` for information on *`tagged-sig-spec`*, *`init-depends-decl`*, and *`unit-body-expr-or-defn`*.

```
(compound-unit/infer
  (import tagged-infer-link-import ...)
  (export tagged-infer-link-export ...)
  (link infer-linkage-decl ...))

tagged-infer-link-import = tagged-sig-id
                         | (link-id : tagged-sig-id)

tagged-infer-link-export = (tag id infer-link-export)
                         | infer-link-export

       infer-link-export = link-id
                         | sig-id

      infer-linkage-decl = ((link-binding ...) unit-id
                                               tagged-link-id)
                         | unit-id
```

Like compound-unit. Syntactically, the difference between compound-unit and compound-unit/infer is that the unit-expr for a linked unit is replaced with a *unit-id*, where a *unit-id* is bound by define-unit (or one of the other unit-binding forms that we introduce later in this section). Furthermore, an import can name just a *sig-id* without locally binding a *link-id*, and an export can be based on a *sig-id* instead of a *link-id*, and a declaration in the *link* clause can be simply a *unit-id* with no specified exports or imports.

The compound-unit/infer form expands to compound-unit by adding sig-ids as needed to the import clause, by replacing *sig-id*s in the export clause by *link-id*s, and by completing the declarations of the *link* clause. This completion is based on static information associated with each *unit-id*. Links and exports can be inferred when all signatures exported by the linked units are distinct from each other and from all imported signatures, and when all imported signatures are distinct. Two signatures are *distinct* only if when they share no ancestor through extends.

The long form of a *link* declaration can be used to resolve ambiguity by giving names to some of a unit's exports and supplying specific bindings for some of a unit's imports. The long form need not name all of a unit's exports or supply all of a unit's imports if the remaining parts can be inferred.

Like compound-unit, the compound-unit/infer form produces a (compound) unit without statically binding information about the result unit's imports and exports. That is, compound-unit/infer consumes static information, but it does not generate it. Two additional forms, define-compound-unit and define-compound-unit/infer, generate static information (where the former does not consume static information).

```
(define-compound-unit id
  (import link-binding ...)
  (export tagged-link-id ...)
  (link linkage-decl ...))
```

Like compound-unit, but binds static information about the compound unit like define-unit.

```
(define-compound-unit/infer id
  (import link-binding ...)
  (export tagged-infer-link-export ...)
  (link infer-linkage-decl ...))
```

Like compound-unit/infer, but binds static information about the compound unit like define-unit.

```
(define-unit-binding unit-id
```

```
  unit-expr
  (import tagged-sig-spec ...+)
  (export tagged-sig-spec ...+)
  init-depends-decl)
```

Like `define-unit`, but the unit implementation is determined from an existing unit produced by *`unit-expr`*. The imports and exports of the unit produced by *`unit-expr`* must be consistent with the declared imports and exports, otherwise the `exn:fail:contract` exception is raised when the `define-unit-binding` form is evaluated.

```
(invoke-unit/infer unit-id)
```

Like `invoke-unit`, but uses static information associated with *`unit-id`* to infer which imports must be assembled from the current context.

```
(define-values/invoke-unit/infer unit-id)
```

Like `define-values/invoke-unit`, but uses static information associated with *`unit-id`* to infer which imports must be assembled from the current context and what exports should be bound by the definition.

6.5 Generating A Unit from Context

```
(unit-from-context tagged-sig-spec)
```

Creates a unit that implements an interface using bindings in the enclosing environment. The generated unit is essentially the same as

```
(unit
  (import)
  (export tagged-sig-spec)
  (define id expr) ...)
```

for each `id` that must be defined to satisfy the exports, and each corresponding `expr` produces the value of `id` in the environment of the `unit-from-context` expression. (The unit cannot be written as above, however, since each `id` definition within the unit shadows the binding outside the `unit` form.)

See `unit` for the grammar of *`tagged-sig-spec`*.

```
(define-unit-from-context id tagged-sig-spec)
```

Like `unit-from-context`, in that a unit is constructed from the enclosing environment, and like `define-unit`, in that *id* is bound to static information to be used later with inference.

6.6 Structural Matching

```
(unit/new-import-export
  (import tagged-sig-spec ...)
  (export tagged-sig-spec ...)
  init-depends-decl
  ((tagged-sig-spec ...) unit-expr tagged-sig-spec))
```

Similar to `unit`, except the body of the unit is determined by an existing unit produced by *unit-expr*. The result is a unit that whose implementation is *unit-expr*, but whose imports, exports, and initialization dependencies are as in the `unit/new-import-export` form (instead of as in the unit produced by *unit-expr*).

The final clause of the `unit/new-import-export` form determines the connection between the old and new imports and exports. The connection is similar to the way that `compound-unit` propagates imports and exports; the difference is that the connection between `import` and the right-hand side of the link clause is based on the names of elements in signatures, rather than the names of the signatures. That is, a *tagged-sig-spec* on the right-hand side of the link clause need not apppear as a *tagged-sig-spec* in the `import` clause, but each of the bindings implied by the linking *tagged-sig-spec* must be implied by some *tagged-sig-spec* in the `import` clause. Similarly, each of the bindings implied by an `export` *tagged-sig-spec* must be implied by some left-hand-side *tagged-sig-spec* in the linking clause.

```
(define-unit/new-import-export unit-id
  (import tagged-sig-spec ...)
  (export tagged-sig-spec ...)
  init-depends-decl
  ((tagged-sig-spec ...) unit-expr tagged-sig-spec))
```

Like `unit/new-import-export`, but binds static information to *unit-id* like `define-unit`.

6.7 Extending the Syntax of Signatures

```
(define-signature-form sig-form-id expr)
(define-signature-form (sig-form-id id) body ...+)
```

Binds sig-form-id for use within a define-signature form.

In the first form, the result of expr must be a transformer procedure. In the second form, sig-form-id is bound to a transformer procedure whose argument is id and whose body is the bodys. The result of the transformer must be a list of syntax objects, which are substituted for a use of sig-form-id in a define-signature expansion. (The result is a list so that the transformer can produce multiple declarations; define-signature has no splicing begin form.)}

```
(struct id (field ...) option ...)

 field = id
       | [id #:mutable]

option = #:mutable
       | #:omit-constructor
       | #:omit-define-syntaxes
       | #:omit-define-values
```

For use with define-signature. The expansion of a struct signature form includes all of the identifiers that would be bound by (define-struct id (field ...) option ...), where the extra option #:omit-constructor omits the make-id identifier.

6.8 Unit Utilities

```
(unit? v) → boolean?
  v : any/c
```

Returns #t if v is a unit, #f otherwise.

```
(provide-signature-elements sig-spec ...)
```

Expands to a provide of all identifiers implied by the sig-specs. See unit for the grammar of sig-spec.

6.9 Single-Unit Modules

When scheme/unit is used as a language name with #lang, the module body is treated as a unit body. The body must match the following module-body grammar:

```
module-body = require-decl ...
              (import tagged-sig-expr ...)
              (export tagged-sig-expr ...)
              init-depends-decl
              unit-body-expr-or-defn
              ...

require-decl = (require require-spec ...)
             | (begin require-decl ...)
             | derived-require-form
```

After any number of *require-decl*s, the content of the module is the same as a `unit` body.

The resulting unit is exported as *base*`@`, where *base* is derived from the enclosing module's name (i.e., its symbolic name, or its path without the directory and file suffix). If the module name ends in `-unit`, then *base* corresponds to the module name before `-unit`. Otherwise, the module name serves as *base*.

6.10 Single-Signature Modules

```
#lang scheme/signature
```

The `scheme/signature` language treats a module body as a unit signature.

The body must match the following *module-body* grammar:

```
module-body = (require require-spec ...) ... sig-spec ...
```

See §6.1 "Creating Units" for the grammar of *sig-spec*. Unlike the body of a `scheme/unit` module, a `require` in a `scheme/signature` module must be a literal use of `require`.

The resulting signature is exported as *base*`^`, where *base* is derived from the enclosing module's name (i.e., its symbolic name, or its path without the directory and file suffix). If the module name ends in `-sig`, then *base* corresponds to the module name before `-sig`. Otherwise, the module name serves as *base*.

6.11 Transformer Helpers

```
(require scheme/unit-exptime)
```

The `scheme/unit-exptime` library provides procedures that are intended for use by macro transformers. In particular, the library is typically imported using `for-syntax` into a module that defines macro with `define-syntax`.

```
(unit-static-signatures unit-identifier
                        err-syntax)
 → (list/c (cons/c (or/c symbol? false/c)
                   identifier?))
   (list/c (cons/c (or/c symbol? false/c)
                   identifier?))
  unit-identifier : identifier?
  err-syntax : syntax?
```

If *unit-identifier* is bound to static unit information via `define-unit` (or other such forms), the result is two values. The first value is for the unit's imports, and the second is for the unit's exports. Each result value is a list, where each list element pairs a symbol or `#f` with an identifier. The symbol or `#f` indicates the import's or export's tag (where `#f` indicates no tag), and the identifier indicates the binding of the corresponding signature.

If *unit-identifier* is not bound to static unit information, then the `exn:fail:syntax` exception is raised. In that case, the given *err-syntax* argument is used as the source of the error, where `unit-identifer` is used as the detail source location.

```
(signature-members sig-identifier
                   err-syntax)    → (or/c identifier? false/c)
                                    (listof identifier?)
                                    (listof identifier?)
                                    (listof identifier?)
  sig-identifier : identifier?
  err-syntax : syntax?
```

If *sig-identifier* is bound to static unit information via `define-signature` (or other such forms), the result is four values:

- an identifier or `#f` indicating the signature (of any) that is extended by the *sig-identifier* binding;
- a list of identifiers representing the variables supplied/required by the signature;
- a list of identifiers for variable definitions in the signature (i.e., variable bindings that are provided on import, but not defined by units that implement the signature); and
- a list of identifiers with syntax definitions in the signature.

If *sig-identifier* is not bound to a signature, then the `exn:fail:syntax` exception is raised. In that case, the given *err-syntax* argument is used as the source of the error, where *sig-identifier* is used as the detail source location.

7 Contracts

This chapter is long on detail and short on the motivation and pragmatics of using contracts. See §7 "Contracts" in the Guide for more of the latter and less of the former.

A *contract* controls the flow of values to ensure that the expectations of one party are met by another party. The `provide/contract` form is the primary mechanism for associating a contract with a binding.

```
(require scheme/contract)
```

The bindings documented in this section are provided by the `scheme/contract` and `scheme` libraries, but not `scheme/base`.

7.1 Data-structure Contracts

A *flat contract* can be fully checked immediately for a given value.

```
(flat-contract predicate) → flat-contract?
  predicate : (any/c . -> . any/c)
```

Constructs a flat contract from *`predicate`*. A value satisfies the contract if the predicate returns a true value.

```
(flat-named-contract type-name predicate) → flat-contract?
  type-name : string?
  predicate : (any/c . -> . any/c)
```

Like `flat-contract`, but the first argument must be a string used for error reporting. The string describes the type that the predicate checks for.

```
any/c : flat-contract?
```

A flat contract that accepts any value.

When using this contract as the result portion of a function contract, consider using `any` instead; using `any` leads to better memory performance, but it also allows multiple results.

```
none/c : flat-contract?
```

A flat contract that accepts no values.

```
(or/c contract ...) → contract?
  contract : (or/c contract? (any/c . -> . any/c))
```

Takes any number of predicates and higher-order contracts and returns a contract that accepts any value that any one of the contracts accepts, individually.

If all of the arguments are procedures or flat contracts, the result is a flat contract. If only one of the arguments is a higher-order contract, the result is a contract that just checks the flat contracts and, if they don't pass, applies the higher-order contract.

If there are multiple higher-order contracts, `or/c` uses `contract-first-order-passes?` to distinguish between them. More precisely, when an `or/c` is checked, it first checks all of the flat contracts. If none of them pass, it calls `contract-first-order-passes?` with each of the higher-order contracts. If only one returns true, `or/c` uses that contract. If none of them return true, it signals a contract violation. If more than one returns true, it signals an error indicating that the `or/c` contract is malformed.

The `or/c` result tests any value by applying the contracts in order, from left to right, with the exception that it always moves the non-flat contracts (if any) to the end, checking them last.

```
(and/c contract ...) → contract?
  contract : (or/c contract? (any/c . -> . any/c))
```

Takes any number of contracts and returns a contract that checks that accepts any value that satisfies all of the contracts, simultaneously.

If all of the arguments are procedures or flat contracts, the result is a flat contract.

The contract produced by `and/c` tests any value by applying the contracts in order, from left to right.

```
(not/c flat-contract) → flat-contract?
  flat-contract : (or/c flat-contract? (any/c . -> . any/c))
```

Accepts a flat contracts or a predicate and returns a flat contract that checks the inverse of the argument.

```
(=/c z) → flat-contract?
  z : number?
```

Returns a flat contract that requires the input to be a number and `=` to *z*.

```
(</c n) → flat-contract?
```

```
n : real?
```

Returns a flat contract that requires the input to be a number and < to *n*.

```
(>/c n) → flat-contract?
  n : number?
```

Like </c, but for >.

```
(<=/c n) → flat-contract?
  n : number?
```

Like </c, but for <=.

```
(>=/c n) → flat-contract?
  n : number?
```

Like </c, but for >=.

```
(between/c n m) → flat-contract?
  n : number?
  m : number?
```

Returns a flat contract that requires the input to be a between *n* and *m* or equal to one of them.

```
(real-in n m) → flat-contract?
  n : real?
  m : real?
```

Returns a flat contract that requires the input to be a real number between *n* and *m*, inclusive.

```
(integer-in j k) → flat-contract?
  j : exact-integer?
  k : exact-integer?
```

Returns a flat contract that requires the input to be an exact integer between *j* and *k*, inclusive.

```
natural-number/c : flat-contract?
```

A flat contract that requires the input to be an exact non-negative integer.

```
(string-len/c len) → flat-contract?
  len : nonnegative-exact-integer?
```

Returns a flat contract that recognizes strings that have fewer than `len` characters.

```
false/c : flat-contract?
```

A flat contract that recognizes `#f`.

```
printable/c : flat-contract?
```

A flat contract that recognizes values that can be written out and read back in with `write` and `read`.

```
(one-of/c v ...+) → flat-contract?
  v : any/c
```

Accepts any number of atomic values and returns a flat contract that recognizes those values, using `eqv?` as the comparison predicate. For the purposes of `one-of/c`, atomic values are defined to be: characters, symbols, booleans, null keywords, numbers, void, and undefined.

```
(symbols sym ...+) → flat-contract?
  sym : symbol?
```

Accepts any number of symbols and returns a flat contract that recognizes those symbols.

```
(vectorof c) → flat-contract?
  c : (or/c flat-contract? (any/c . -> . any/c))
```

Accepts a flat contract (or a predicate that is converted to a flat contract via `flat-contract`) and returns a flat contract that checks for vectors whose elements match the original contract.

```
(vector-immutableof c) → contract?
  c : (or/c contract? (any/c . -> . any/c))
```

Like `vectorof`, but the contract needs not be a flat contract. Beware that when this contract is applied to a value, the result is not `eq?` to the input.

```
(vector/c c ...) → flat-contract?
  c : (or/c flat-contract? (any/c . -> . any/c))
```

Accepts any number of flat contracts (or predicates that are converted to flat contracts via `flat-contract`) and returns a flat-contract that recognizes vectors. The number of elements in the vector must match the number of arguments supplied to `vector/c`, and each element of the vector must match the corresponding flat contract.

```
(vector-immutable/c c ...) → contract?
  c : (or/c contract? (any/c . -> . any/c))
```

Like `vector/c`, but the individual contracts need not be flat contracts. Beware that when this contract is applied to a value, the result is not `eq?` to the input.

```
(box/c c) → flat-contract?
  c : (or/c flat-contract? (any/c . -> . any/c))
```

Returns a flat-contract that recognizes boxes. The content of the box must match *c*.

```
(box-immutable/c c) → contract?
  c : (or/c contract? (any/c . -> . any/c))
```

Like `box/c`, but *c* need not be flat contract. Beware that when this contract is applied to a value, the result is not `eq?` to the input.

```
(listof c) → contract?
  c : (or/c contract? (any/c . -> . any/c))
```

Returns a contract that recognizes a list whose every element matches the contract *c*. Beware that when this contract is applied to a value, the result is not necessarily `eq?` to the input.

```
(cons/c car-c cdr-c) → contract?
  car-c : contract?
  cdr-c : contract?
```

Produces a contract the recognizes pairs first and second elements match *car-c* and *cdr-c*, respectively. Beware that when this contract is applied to a value, the result is not necessarily `eq?` to the input.

```
(list/c c ...) → contract?
```

```
c : (or/c contract? (any/c . -> . any/c))
```

Produces a contract for a list. The number of elements in the list must match the number of arguments supplied to `list/c`, and each element of the list must match the corresponding contract. Beware that when this contract is applied to a value, the result is not necessarily `eq?` to the input.

```
(syntax/c c) → flat-contract?
  c : flat-contract?
```

Produces a flat contract that recognizes syntax objects whose `syntax-e` content matches *`c`*.

```
(struct/c struct-id flat-contract-expr ...)
```

Produces a flat contract that recognizes instances of the structure type named by *`struct-id`*, and whose field values match the flat contracts produced by the *`flat-contract-expr`*s.

```
(parameter/c c) → contract?
  c : contract?
```

Produces a contract on parameters whose values must match `contract`.

```
(flat-rec-contract id flat-contract-expr ...)
```

Constructs a recursive flat contract. A `flat-contract-expr` can refer to `id` to refer recursively to the generated contract.

For example, the contract

```
(flat-rec-contract sexp
  (cons/c sexp sexp)
  number?
  symbol?)
```

is a flat contract that checks for (a limited form of) S-expressions. It says that an `sexp` is either two `sexp` combined with `cons`, or a number, or a symbol.

Note that if the contract is applied to a circular value, contract checking will not terminate.}

```
(flat-murec-contract ([id flat-contract-expr ...] ...) body ...+)
```

A generalization of `flat-rec-contracts` for defining several mutually recursive flat contracts simultaneously. Each *id* is visible in the entire `flat-murec-contract` form, and the result of the final *body* is the result of the entire form.

```
any
```

Represents a contract that is always satisfied. In particular, it can accept multiple values. It can only be used in a result position of contracts like `->`. Using `any` elsewhere is a syntax error.

```
(promise/c expr)
```

Constructs a contract on a promise. The contract does not force the promise, but when the promise is forced, the contract checks that the result value meets the contract produced by *expr*.

7.2 Function Contracts

A *function contract* wraps a procedure to delay checks for its arguments and results. There are three primary function contract combinators that have increasing amounts of expressiveness and increasing additional overheads. The first `->` is the cheapest. It generates wrapper functions that can call the original function directly. Contracts built with `->*` require packaging up arguments as lists in the wrapper function and then using either `keyword-apply` or `apply`. Finally, `->d` is the most expensive, because it requires delaying the evaluation of the contract expressions for the domain and range until the function itself is called or returns.

The `case->` contract is a specialized contract, designed to match `case-lambda` and `unconstrained-domain->` allows range checking without requiring that the domain have any particular shape (see below for an exmaple use).

```
(-> dom ... range)

  dom = dom-expr
      | keyword dom-expr

range = range-expr
      | (values range-expr ...)
      | any
```

Produces a contract for a function that accepts a fixed number of arguments and returns either a fixed number of results or completely unspecified results (the latter when `any` is specified).

Each *dom-expr* is a contract on an argument to a function, and each `res-expr` is a contract on a result of the function.

For example,

```
(integer? boolean? . -> . integer?)
```

produces a contract on functions of two arguments. The first argument must be an integer, and the second argument must be a boolean. The function must produce an integer.

Using an `->` between two whitespace-delimited `.`s is the same as putting the `->` right after the enclosing open parenthesis. See §2.4.3 "Lists and Scheme Syntax" or §12.6.5 "Reading Pairs and Lists" for more information.

A domain specification may include a keyword. If so, the function must accept corresponding (mandatory) keyword arguments, and the values for the keyword arguments must match the corresponding contracts. For example:

```
(integer? #:x boolean? . -> . integer?)
```

is a contract on a function that accepts a by-position argument that is an integer and a `#:x` argument is that a boolean.

If `any` is used as the last sub-form for `->`, no contract checking is performed on the result of the function, and tail-recursion is preserved. Note that the function may return multiple values in that case.

If `(values res-expr ...)` is used as the last sub-form of `->`, the function must produce a result for each contract, and each values must match its respective contract.

```
(->* (mandatory-dom ...) (optional-dom ...) rest range)

mandatory-dom = dom-expr
              | keyword dom-expr

 optional-dom = dom-expr
              | keyword dom-expr

         rest =
              | #:rest rest-expr

        range = range-expr
              | (values range-expr ...)
              | any
```

The `->*` contract combinator produces contracts for functions that accept optional arguments (either keyword or positional) and or arbitrarily many arguments. The first clause of a `->*` contract describes the mandatory arguments, and is similar to the argument description of a `->` contract. The second clause describes the optional arguments. The last clause describes the range of the function. It can either be `any` or a sequence of contracts, indicating that the function must return multiple values. If present, the `rest-expr` contract governs the arguments in the rest parameter.

As an example, the contract

```
(->* () (boolean? #:x integer?) #:rest (listof symbol?) (symbol?))
```

matches functions that optionally accept a boolean, an integer keyword argument `#:x` and arbitrarily more symbols, and that return a symbol.

```
(->d (mandatory-dependent-dom ...)
     (optional-dependent-dom ...)
     dependent-rest
     pre-cond
     dep-range)

mandatory-dependent-dom = [id dom-expr]
                        | keyword [id dom-expr]

 optional-dependent-dom = [id dom-expr]
                        | keyword [id dom-expr]

         dependent-rest =
                        | #:rest id rest-expr

               pre-cond =
                        | #:pre-cond boolean-expr

              dep-range = any
                        | [_ range-expr] post-cond
                        | (values [_ range-expr] ...) post-cond
                        | [id range-expr] post-cond
                        | (values [id range-expr] ...) post-cond

              post-cond =
                        | #:post-cond boolean-expr
```

The `->d` is similar in shape to `->*`, with two extensions: names have been added to each argument and result, which allows the contracts to depend on the values of the arguments and results, and pre- and post-condition expressions have been added in order to express contracts that are not naturally tied to a particular argument or result.

The first two subforms of a `->d` contract cover the mandatory and optional arguments. Following that is an optional rest-args contract, and an optional pre-condition. The *dep-range* non-terminal covers the possible post-condition contracts. If it is `any`, then any result (or results) are allowed. Otherwise, the result contract can be a name and a result contract, or a multiple values return and, in either of the last two cases, it may be optionally followed by a post-condition.

Each of the *id*s on an argument (including the rest argument) is visible in all of the subexpressions of `->d`. Each of the *id*s on a result is visible in the subexpressions of the *dep-range*.

If the identifier position of the range contract is `_` (an underscore), then the range contract expressions are evaluated when the function is called (and the underscore is not bound in the range). Otherwise the range expressions are evaluated when the function returns.

```
(case-> (-> dom-expr ... rest range) ...)

 rest =
      | #:rest rest-expr

range = range-expr
      | (values range-expr ...)
      | any
```

This contract form is designed to match `case-lambda`. Each argument to `case->` is a contract that governs a clause in the `case-lambda`. If the `#:rest` keyword is present, the corresponding clause must accept an arbitrary number of arguments. The *range* specification is just like that for `->` and `->*`.

```
(unconstrained-domain-> res-expr ...)
```

Constructs a contract that accepts a function, but makes no constraint on the function's domain. The *res-expr*s determine the number of results and the contract for each result.

Generally, this contract must be combined with another contract to ensure that the domain is actually known to be able to safely call the function itself.

For example, the contract

```
(provide/contract
 [f (->d ([size natural-number/c]
          [proc (and/c (unconstrained-domain-> number?)
                       (lambda (p)
                         (procedure-arity-includes? p size)))])
         ()
         number?)])
```

says that the function `f` accepts a natural number and a function. The domain of the function that `f` accepts must include a case for `size` arguments, meaning that `f` can safely supply `size` arguments to its input.

For example, the following is a definition of `f` that cannot be blamed using the above contract:

```
(define (f i g)
  (apply g (build-list i add1)))
```

7.3 Lazy Data-structure Contracts

```
(define-contract-struct id (field-id ...))
```

Like `define-struct`, but with two differences: it does not define field mutators, and it does define two contract constructors: *`id`*`/c` and *`id`*`/dc`. The first is a procedure that accepts as many arguments as there are fields and returns a contract for struct values whose fields match the arguments. The second is a syntactic form that also produces contracts on the structs, but the contracts on later fields may depend on the values of earlier fields.

The generated contract combinators are *lazy*: they only verify the contract holds for the portion of some data structure that is actually inspected. More precisely, a lazy data structure contract is not checked until a selector extracts a field of a struct.

```
(id/dc field-spec ...)

field-spec = [field-id contract-expr]
           | [field-id (field-id ...) contract-expr]
```

In each *`field-spec`* case, the first *`field-id`* specifies which field the contract applies to; the fields must be specified in the same order as the original `define-contract-struct`. The first case is for when the contract on the field does not depend on the value of any other field. The second case is for when the contract on the field does depend on some other fields, and the parenthesized *`field-id`*s indicate which fields it depends on; these dependencies can only be to earlier fields.

As an example, consider the following module:

```
(module product mzscheme
  (require mzlib/contract)

  (define-contract-struct kons (hd tl))

  ; sorted-list/gt : number -> contract
  ; produces a contract that accepts
```

```
; sorted kons-lists whose elements
; are all greater than num.
(define (sorted-list/gt num)
  (or/c null?
        (kons/dc [hd (>=/c num)]
                 [tl (hd) (sorted-list/gt hd)])))

; product : kons-list -> number
; computes the product of the values
; in the list. if the list contains
; zero, it avoids traversing the rest
; of the list.
(define (product l)
  (cond
    [(null? l) 1]
    [else
     (if (zero? (kons-hd l))
         0
         (* (kons-hd l)
            (product (kons-tl l))))]))

(provide kons? make-kons kons-hd kons-tl)
(provide/contract [product (-> (sorted-list/gt -inf.0) number?)])
```

The module provides a single function, `product` whose contract indicates that it accepts sorted lists of numbers and produces numbers. Using an ordinary flat contract for sorted lists, the product function cannot avoid traversing having its entire argument be traversed, since the contract checker will traverse it before the function is called. As written above, however, when the product function aborts the traversal of the list, the contract checking also stops, since the `kons/dc` contract constructor generates a lazy contract.

7.4 Attaching Contracts to Values

```
(provide/contract p/c-item ...)

p/c-item = (struct id ((id contract-expr) ...))
         | (struct (id identifier) ((id contract-expr) ...))
         | (rename orig-id id contract-expr)
         | (id contract-expr)
```

Can only appear at the top-level of a `module`. As with `provide`, each *`id`* is provided from the module. In addition, clients of the module must live up to the contract specified by *`contract-expr`* for each export.

The `provide/contract` form treats modules as units of blame. The module that defines the provided variable is expected to meet the positive (co-variant) positions of the contract. Each module that imports the provided variable must obey the negative (contra-variant) positions of the contract.

Only uses of the contracted variable outside the module are checked. Inside the module, no contract checking occurs.

The `rename` form of a `provide/contract` exports the first variable (the internal name) with the name specified by the second variable (the external name).

The `struct` form of a `provide/contract` clause provides a structure definition, and each field has a contract that dictates the contents of the fields. The struct definition must come before the provide clause in the module's body. If the struct has a parent, the second `struct` form (above) must be used, with the first name referring to the struct itself and the second name referring to the parent struct. Unlike `define-struct`, however, all of the fields (and their contracts) must be listed. The contract on the fields that the sub-struct shares with its parent are only used in the contract for the sub-struct's maker, and the selector or mutators for the super-struct are not provided.

```
(define/contract id contract-expr init-value-expr)
```

Attaches the contract *`contract-expr`* to *`init-value-expr`* and binds that to *`id`*.

The `define/contract` form treats individual definitions as units of blame. The definition itself is responsible for positive (co-variant) positions of the contract and each reference to *`id`* (including those in the initial value expression) must meet the negative positions of the contract.

Error messages with `define/contract` are not as clear as those provided by `provide/contract`, because `define/contract` cannot detect the name of the definition where the reference to the defined variable occurs. Instead, it uses the source location of the reference to the variable as the name of that definition.

```
(contract contract-expr to-protect-expr
          positive-blame-expr negative-blame-expr)
(contract contract-expr to-protect-expr
          positive-blame-expr negative-blame-expr
          contract-source-expr)
```

The primitive mechanism for attaching a contract to a value. The purpose of `contract` is as a target for the expansion of some higher-level contract specifying form.

The `contract` expression adds the contract specified by *`contract-expr`* to the value produced by *`to-protect-expr`*. The result of a `contract` expression is the result of the *`to-`*

protect-expr expression, but with the contract specified by *contract-expr* enforced on *to-protect-expr*.

The values of *positive-blame-expr* and *negative-blame-expr* must be symbols indicating how to assign blame for positive and negative positions of the contract specified by *contract-expr*.

If specified, *contract-source-expr*, indicates where the contract was assumed. Its value must be a syntax object specifying the source location of the location where the contract was assumed. If the syntax object wraps a symbol, the symbol is used as the name of the primitive whose contract was assumed. If absent, it defaults to the source location of the `contract` expression.

7.5 Building New Contract Combinators

Contracts are represented internally as functions that accept information about the contract (who is to blame, source locations, etc) and produce projections (in the spirit of Dana Scott) that enforce the contract. A projection is a function that accepts an arbitrary value, and returns a value that satisfies the corresponding contract. For example, a projection that accepts only integers corresponds to the contract `(flat-contract integer?)`, and can be written like this:

```
(define int-proj
  (lambda (x)
    (if (integer? x)
        x
        (signal-contract-violation))))
```

As a second example, a projection that accepts unary functions on integers looks like this:

```
(define int->int-proj
  (lambda (f)
    (if (and (procedure? f)
             (procedure-arity-includes? f 1))
        (lambda (x)
          (int-proj (f (int-proj x))))
        (signal-contract-violation))))
```

Although these projections have the right error behavior, they are not quite ready for use as contracts, because they do not accomodate blame, and do not provide good error messages. In order to accomodate these, contracts do not just use simple projections, but use functions that accept the names of two parties that are the candidates for blame, as well as a record of the source location where the contract was established and the name of the contract. They can then, in turn, pass that information to `raise-contract-error` to signal a good error message (see below for details on its behavior).

Here is the first of those two projections, rewritten for use in the contract system:

```
(define (int-proj pos neg src-info name)
  (lambda (x)
    (if (integer? x)
        x
        (raise-contract-error
         val
         src-info
         pos
         name
         "expected <integer>, given: ~e"
         val))))
```

The first two new arguments specify who is to be blamed for positive and negative contract violations, respectively. Contracts, in this system, are always established between two parties. One party provides some value according to the contract, and the other consumes the value, also according to the contract. The first is called the "positive" person and the second the "negative". So, in the case of just the integer contract, the only thing that can go wrong is that the value provided is not an integer. Thus, only the positive argument can ever accrue blame (and thus only `pos` is passed to `raise-contract-error`).

Compare that to the projection for our function contract:

```
(define (int->int-proj pos neg src-info name)
  (let ([dom (int-proj neg pos src-info name)]
        [rng (int-proj pos neg src-info name)])
    (lambda (f)
      (if (and (procedure? f)
               (procedure-arity-includes? f 1))
          (lambda (x)
            (rng (f (dom x))))
          (raise-contract-error
           val
           src-info
           pos
           name
           "expected a procedure of one argument, given: ~e"
           val)))))
```

In this case, the only explicit blame covers the situation where either a non-procedure is supplied to the contract, or where the procedure does not accept one argument. As with the integer projection, the blame here also lies with the producer of the value, which is why `raise-contract-error` gets `pos` and not `neg` as its argument.

The checking for the domain and range are delegated to the `int-proj` function, which is supplied its arguments in the first two line of the `int->int-proj` function. The trick here is that, even though the `int->int-proj` function always blames what it sees as positive we can reverse the order of the `pos` and `neg` arguments so that the positive becomes the negative.

This is not just a cheap trick to get this example to work, however. The reversal of the positive and the negative is a natural consequence of the way functions behave. That is, imagine the flow of values in a program between two modules. First, one module defines a function, and then that module is required by another. So, far the function itself has to go from the original, providing module to the requiring module. Now, imagine that the providing module invokes the function, suppying it an argument. At this point, the flow of values reverses. The argument is travelling back from the requiring module to the providing module! And finally, when the function produces a result, that result flows back in the original direction. Accordingly, the contract on the domain reverses the positive and the negative, just like the flow of values reverses.

We can use this insight to generalize the function contracts and build a function that accepts any two contracts and returns a contract for functions between them.

```
(define (make-simple-function-contract dom-proj range-proj)
  (lambda (pos neg src-info name)
    (let ([dom (dom-proj neg pos src-info name)]
          [rng (range-proj pos neg src-info name)])
      (lambda (f)
        (if (and (procedure? f)
                 (procedure-arity-includes? f 1))
            (lambda (x)
              (rng (f (dom x))))
            (raise-contract-error
             val
             src-info
             pos
             name
             "expected a procedure of one argument, given: ~e"
             val))))))
```

Projections like the ones described above, but suited to other, new kinds of value you might make, can be used with the contract library primitives below.

```
(make-proj-contract name
                    proj
                    first-order-test) → contract?
  name : any/c
  proj : (symbol? symbol? any/c any/c . -> . any/c)
  first-order-test : (any/c . -> . any/c)
```

The simplest way to build a contract. It can be less efficient than using other contract constructors described below, but it is the right choice for new contract constructors or first-time contract builders.

The first argument is the name of the contract. It can be an arbitrary S-expression. The second is a projection (see above).

The final argument is a predicate that is a conservative, first-order test of a value. It should be a function that accepts one argument and returns a boolean. If it returns `#f`, its argument must be guaranteed to fail the contract, and the contract should detect this right when the projection is invoked. If it returns true, the value may or may not violate the contract, but any violations must not be signaled immediately.

From the example above, the predicate should accept unary functions, but reject all other values.

```
(build-compound-type-name c/s ...) → any
  c/s : any/c
```

Produces an S-expression to be used as a name for a contract. The arguments should be either contracts or symbols. It wraps parenthesis around its arguments and extracts the names from any contracts it is supplied with.

```
(coerce-contract id expr)
```

Evaluates *expr* and, if the result is a contract, just returns it. If the result is a procedure of arity one, it converts that into a contract. If the result is neither, it signals an error, using the first argument in the error message. The message says that a contract or a procedure of arity one was expected.

```
(flat-contract/predicate? val) → boolean?
  val : any/c
```

A predicate that indicates when `coerce-contract` will fail.

```
(raise-contract-error val
                      src-info
                      to-blame
                      contract-name
                      fmt
                      arg ...)       → any
  val : any/c
  src-info : any/c
  to-blame : symbol?
  contract-name : any/c
```

```
fmt : string?
arg : any/c
```

Signals a contract violation. The first argument is the value that failed to satisfy the contract. The second argument is is the *src-info* passed to the projection and the third should be either `pos` or `neg` (typically `pos`, see the beginning of this section) that was passed to the projection. The fourth argument is the *contract-name* that was passed to the projection and the remaining arguments are used with `format` to build an actual error message.

7.6 Contract Utilities

```
(guilty-party exn) → any
  exn : exn?
```

Extracts the name of the guilty party from an exception raised by the contract system.

```
(contract? v) → boolean?
  v : any/c
```

Returns `#t` if its argument is a contract (ie, constructed with one of the combinators described in this section), `#f` otherwise.

```
(flat-contract? v) → boolean?
  v : any/c
```

Returns `#t` when its argument is a contract that has been constructed with `flat-contract` (and thus is essentially just a predicate), `#f` otherwise.

```
(flat-contract-predicate v) → (any/c . -> . any/c)
  v : flat-contract?
```

Extracts the predicate from a flat contract.

```
(contract-first-order-passes? contract v) → boolean?
  contract : contract?
  v : any/c
```

Returns a boolean indicating if the first-order tests of *contract* pass for *v*.

If it returns `#f`, the contract is guaranteed not to hold for that value; if it returns `#t`, the contract may or may not hold. If the contract is a first-order contract, a result of `#t` guarantees that the contract holds.

```
(make-none/c sexp-name) → contract?
  sexp-name : any/c
```

Makes a contract that accepts no values, and reports the name *sexp-name* when signaling a contract violation.

```
(contract-violation->string)
 → (any/c any/c symbol? symbol? any/c string? . -> . string?)
(contract-violation->string proc) → void?
  proc : (any/c any/c symbol? symbol? any/c string? . -> . string?)
```

This is a parameter that is used when constructing a contract violation error. Its value is procedure that accepts six arguments: the value that the contract applies to, a syntax object representing the source location where the contract was established, the names of the two parties to the contract (as symbols) where the first one is the guilty one, an sexpression representing the contract, and a message indicating the kind of violation. The procedure then returns a string that is put into the contract error message. Note that the value is often already included in the message that indicates the violation.

```
(recursive-contract contract-expr)
```

Delays the evaluation of its argument until the contract is checked, making recursive contracts possible.

```
(opt/c contract-expr)
```

This optimizes its argument contract expression by traversing its syntax and, for known contract combinators, fuses them into a single contract combinator that avoids as much allocation overhad as possible. The result is a contract that should behave identically to its argument, except faster (due to the less allocation).

```
(define-opt/c (id id ...) expr)
```

This defines a recursive contract and simultaneously optimizes it. Semantically, it behaves just as if the `-opt/c` were not present, defining a function on contracts (except that the body expression must return a contract). But, it also optimizes that contract definition, avoiding extra allocation, much like `opt/c` does.

For example,

```
(define-contract-struct bt (val left right))

(define-opt/c (bst-between/c lo hi)
  (or/c null?
        (bt/c [val (real-in lo hi)]
              [left (val) (bst-between/c lo val)]
              [right (val) (bst-between/c val hi)])))

(define bst/c (bst-between/c -inf.0 +inf.0))
```

defines the `bst/c` contract that checks the binary search tree invariant. Removing the `-opt/c` also makes a binary search tree contract, but one that is (approximately) 20 times slower.

8 Pattern Matching

The `match` form and related forms support general pattern matching on Scheme values. See also §3.7 "Regular Expressions" for information on regular-expression matching on strings, bytes, and streams.

```
(require scheme/match)
```

The bindings documented in this section are provided by the `scheme/match` and `scheme` libraries, but not `scheme/base`.

```
(match val-expr clause ...)

clause = [pat expr ...+]
       | [pat (=> id) expr ...+]
```

Finds the first *pat* that matches the result of *val-expr*, and evaluates the corresponding *expr*s with bindings introduced by *pat* (if any). The last *expr* in the matching clause is evaluated in tail position with respect to the `match` expression.

The *clause*s are tried in order to find a match. If no *clause* matches, then the `exn:fail` exception is raised.

An optional `(=> id)` between a *pat* and the *expr*s is bound to a *failure procedure* of zero arguments. If this procedure is invoked, it escapes back to the pattern matching expression, and resumes the matching process as if the pattern had failed to match. The *expr*s must not mutate the object being matched before calling the failure procedure, otherwise the behavior of matching is unpredictable.

The grammar of *pat* is as follows, where non-italicized identifiers are recognized symbolically (i.e., not by binding).

pat	::=	*id*	match anything, bind identifier
	|	_	match anything
	|	*literal*	match literal
	|	`(quote datum)`	match `equal?` value
	|	`(list lvp ...)`	match sequence of *lvp*s
	|	`(list-rest lvp ... pat)`	match *lvp*s consed onto a *pat*
	|	`(list-no-order pat ...)`	match *pat*s in any order
	|	`(list-no-order pat ... lvp)`	match *pat*s in any order
	|	`(vector lvp ...)`	match vector of *pat*s
	|	`(hash-table (pat pat) ...)`	match hash table
	|	`(hash-table (pat pat) ...+ ooo)`	match hash table
	|	`(cons pat pat)`	match pair of *pat*s
	|	`(mcons pat pat)`	match mutable pair of *pat*s

```
             | (box pat)                         match boxed pat
             | (struct struct-id (pat ...))      match struct-id instance
             | (regexp rx-expr)                  match string
             | (regexp rx-expr pat)              match string, result with pat
             | (pregexp px-expr)                 match string
             | (pregexp px-expr pat)             match string, result with pat
             | (and pat ...)                     match when all pats match
             | (or pat ...)                      match when any pat match
             | (not pat ...)                     match when no pat matches
             | (app expr pat)                    match (expr value) to pat
             | (? expr pat ...)                  match if (expr value) and pats
             | (quasiquote qp)                   match a quasipattern
             | derived-pattern                   match using extension
literal    ::= #t                                match true
             | #f                                match false
             | string                            match equal? string
             | bytes                             match equal? byte string
             | number                            match equal? number
             | char                              match equal? character
             | keyword                           match equal? keyword
             | regexp                            match equal? regexp literal
             | pregexp                           match equal? pregexp literal
lvp        ::= pat ooo                           greedily match pat instances
             | pat                               match pat
qp         ::= literal                           match literal
             | id                                match symbol
             | (qp ...)                          match sequences of qps
             | (qp ... . qp)                     match qps ending qp
             | (qp ... ooo)                      match qps ending repeated qp
             | #(qp ...)                         match vector of qps
             | #&qp                              match boxed qp
             | ,pat                              match pat
             | ,@(list lvp ...)                  match lvps, spliced
             | ,@(list-rest lvp ... pat)         match lvps plus pat, spliced
             | ,@'qp                             match list-matching qp, spliced
ooo        ::= ...                               zero or more; ... is literal
             | ___                               zero or more
             | ..k                               k or more
             | __k                               k or more
```

In more detail, patterns match as follows:

- *id*, excluding the reserved names _, ..., ___, ..k, and __k for non-negative integers *k* — matches anything, and binds *id* to the matching values. If an *id* is used multiple times within a pattern, the corresponding matches must be the same according to (match-equality-test), except that instances of an *id* in different or and not sub-patterns are independent.

Examples:

```
> (match '(1 2 3)
    [(list a b a) (list a b)]
    [(list a b c) (list c b a)])
'(3 2 1)
> (match '(1 '(x y z) 1)
    [(list a b a) (list a b)]
    [(list a b c) (list c b a)])
'(1 '(x y z))
```

- _ — matches anything, without binding any identifiers.

Examples:

```
> (match '(1 2 3)
    [(list _ _ a) a])
3
```

- #t, #f, *string*, *bytes*, *number*, *char*, or (quote *datum*) — matches an equal? constant.

Examples:

```
> (match "yes"
    ["no" #f]
    ["yes" #t])
#t
```

- (list *lvp* ...) — matches a list of elements. In the case of (list *pat* ...), the pattern matches a list with as many element as *pat*s, and each element must match the corresponding *pat*. In the more general case, each *lvp* corresponds to a "spliced" list of greedy matches.

  For spliced lists, ... and ___ are synonyms for zero or more matches. The ..*k* and __*k* forms are also synonyms, specifying *k* or more matches. Pattern variables that precede these splicing operators are bound to lists of matching forms.

Examples:

```
> (match '(1 2 3)
    [(list a b c) (list c b a)])
'(3 2 1)
> (match '(1 2 3)
    [(list 1 a ...) a])
'(2 3)
> (match '(1 2 3)
    [(list 1 a ..3) a]
    [_ 'else])
```

```
else
> (match '(1 2 3 4)
    [(list 1 a ..3) a]
    [_ 'else])
(2 3 4)
> (match '(1 2 3 4 5)
    [(list 1 a ..3 5) a]
    [_ 'else])
(2 3 4)
> (match '(1 (2) (2) (2) 5)
    [(list 1 (list a) ..3 5) a]
    [_ 'else])
(2 2 2)
```

- `(list-rest lvp ... pat)` — similar to a `list` pattern, but the final *`pat`* matches the "rest" of the list after the last *`lvp`*. In fact, the matched value can be a non-list chain of pairs (i.e., an "improper list") if *`pat`* matches non-list values.

Examples:

```
> (match '(1 2 3 . 4)
    [(list-rest a b c d) d])
4
> (match '(1 2 3 . 4)
    [(list-rest a ... d) (list a d)])
((1 2 3) 4)
```

- `(list-no-order pat ...)` — similar to a `list` pattern, but the elements to match each *`pat`* can appear in the list in any order.

Examples:

```
> (match '(1 2 3)
    [(list-no-order 3 2 x) x])
1
```

- `(list-no-order pat ... lvp)` — generalizes `list-no-order` to allow a pattern that matches multiple list elements that are interspersed in any order with matches for the other patterns.

Examples:

```
> (match '(1 2 3 4 5 6)
    [(list-no-order 6 2 y ...) y])
(1 3 4 5)
```

- `(vector lvp ...)` — like a `list` pattern, but matching a vector.

Examples:

```
> (match #(1 (2) (2) (2) 5)
    [(vector 1 (list a) ..3 5) a])
```

```
(2 2 2)
```

- `(hash-table (pat pat) ...)` — similar to `list-no-order`, but matching against hash table's key–value pairs.

Examples:

```
> (match #hash(("a" . 1) ("b" . 2))
    [(hash-table ("b" b) ("a" a)) (list b a)])
(2 1)
```

- `(hash-table (pat pat) ...+ ooo)` — Generalizes `hash-table` to support a final repeating pattern.

Examples:

```
> (match #hash(("a" . 1) ("b" . 2))
    [(hash-table (key val) ...) key])
("a" "b")
```

- `(cons pat1 pat2)` — matches a pair value.

Examples:

```
> (match (cons 1 2)
    [(cons a b) (+ a b)])
3
```

- `(mcons pat1 pat2)` — matches a mutable pair value.

Examples:

```
> (match (mcons 1 2)
    [(cons a b) 'immutable]
    [(mcons a b) 'mutable])
mutable
```

- `(box pat)` — matches a boxed value.

Examples:

```
> (match #&1
    [(box a) a])
1
```

- `(struct struct-id (pat ...))` — matches an instance of a structure type names *struct-id*, where each field in the instance matches the corresponding *pat*.

  Usually, *struct-id* is defined with `define-struct`. More generally, *struct-id* must be bound to expansion-time information for a structure type (see §4.6 "Structure Type Transformer Binding"), where the information includes at least a predicate binding and field accessor bindings corresponding to the number of field *pat*s. In particular, a module import or a `unit` import with a signature containing a `struct` declaration can provide the structure type information.

Examples:

```
(define-struct tree (val left right))

> (match (make-tree 0 (make-tree 1 #f #f) #f)
    [(struct tree (a (struct tree (b  _ _)) _)) (list a b)])
(0 1)
```

- `(struct struct-id _)` — matches any instance of *struct-id*, without regard to contents of the fields of the instance.
- `(regexp rx-expr)` — matches a string that matches the regexp pattern produced by *rx-expr*; see §3.7 "Regular Expressions" for more information about regexps.

Examples:

```
> (match "apple"
    [(regexp #rx"p+") 'yes]
    [_ 'no])
yes
> (match "banana"
    [(regexp #rx"p+") 'yes]
    [_ 'no])
no
```

- `(regexp rx-expr pat)` — extends the `regexp` form to further constrain the match where the result of `regexp-match` is matched against *pat*.

Examples:

```
> (match "apple"
    [(regexp #rx"p+(.)" (list _ "l")) 'yes]
    [_ 'no])
yes
> (match "append"
    [(regexp #rx"p+(.)" (list _ "l")) 'yes]
    [_ 'no])
no
```

- `(pregexp rx-expr)` or `(regexp rx-expr pat)` — like the `regexp` patterns, but if *rx-expr* produces a string, it is converted to a pattern using `pregexp` instead of `regexp`.
- `(and pat ...)` — matches if all of the *pat*s match. This pattern is often used as `(and id pat)` to bind *id* to the entire value that matches *pat*.

Examples:

```
> (match '(1 (2 3) 4)
   [(list _ (and a (list _ ...)) _) a])
(2 3)
```

- `(or pat ...)` — matches if any of the *pat*s match. **Beware**: the result expression can be duplicated once for each *pat*! Identifiers in *pat* are bound only in the corresponding copy of the result expression; in a module context, if the result expression refers to a binding, then that all *pat*s must include the binding.

Examples:

```
> (match '(1 2)
   [(or (list a 1) (list a 2)) a])
1
```

- `(not pat ...)` — matches when none of the *pat*s match, and binds no identifiers.

Examples:

```
> (match '(1 2 3)
   [(list (not 4) ...) 'yes]
   [_ 'no])
yes
> (match '(1 4 3)
   [(list (not 4) ...) 'yes]
   [_ 'no])
no
```

- `(app expr pat)` — applies *expr* to the value to be matched; the result of the application is matched again *pat*.

Examples:

```
> (match '(1 2)
   [(app length 2) 'yes])
yes
```

- `(? expr pat ...)` — applies *expr* to the value to be matched, and checks whether the result is a true value; the additional *pat*s must also match (i.e., `?` combines a predicate application and an `and` pattern).

Examples:

```
> (match '(1 3 5)
   [(list (? odd?) ...) 'yes])
yes
```

- `(quasiquote qp)` — introduces a quasipattern, in which identifiers match symbols. Like the `quasiquote` expression form, `unquote` and `unquote-splicing` escape back to normal patterns.

Examples:

```
> (match '(1 2 3)
    [`(1 ,a ,(? odd? b)) (list a b)])
(2 3)
```

- *derived-pattern* — matches a pattern defined by a macro extension via `define-match-expander`.

8.1 Combined Matching Forms

```
(match-lambda clause ...)
```

Equivalent to `(lambda (id) (match id clause ...))`.

```
(match-lambda* clause ...)
```

Equivalent to `(lambda lst (match lst clause ...))`.

```
(match-let ([pat expr] ...) body ...+)
```

Generalizes `let` to support pattern bindings. Each *expr* is matched against its corresponding *pat* (the match must succeed), and the bindings that *pat* introduces are visible in the *body*s.

Examples:

```
> (match-let ([(list a b) '(1 2)]
              [(vector x ...) #(1 2 3 4)])
    (list b a x))
(2 1 (1 2 3 4))
```

```
(match-let* ([pat expr] ...) body ...+)
```

Like `match-let`, but generalizes `let*`, so that the bindings of each *pat* are available in each subsequent *expr*.

Examples:

```
> (match-let* ([(list a b) '(#(1 2 3 4) 2)]
               [(vector x ...) a])
    x)
(1 2 3 4)
```

```
(match-letrec ([pat expr] ...) body ...+)
```

Like `match-let`, but generalizes `letrec`.

```
(match-define pat expr)
```

Defines the names bound by *pat* to the values produced by matching against the result of *expr*.

Examples:

```
> (match-define (list a b) '(1 2))
> b
2
```

8.2 Extending `match`

```
(define-match-expander id proc-expr)
(define-match-expander id proc-expr proc-expr)
```

Binds *id* to a pattern transformer.

The first *proc-expr* subexpression must evaluate to a transformer that produces a *pat* for `match`. Whenever *id* appears as the beginning of a pattern, this transformer is given, at expansion time, a syntax object corresponding to the entire pattern (including *id*). The pattern is the replaced with the result of the transformer.

A transformer produced by a second *proc-expr* subexpression is used when *id* is used in an expression context. Using the second *proc-expr*, *id* can be given meaning both inside and outside patterns.

```
(match-equality-test) → (any/c any/c . -> . any)
(match-equality-test comp-proc) → void?
  comp-proc : (any/c any/c . -> . any)
```

A parameter that determines the comparison procedure used to check whether multiple uses of an identifier match the “same” value. The default is `equal?`.

9 Control Flow

9.1 Multiple Values

See §1.1.3 "Multiple Return Values" for general information about multiple result values. In addition to `call-with-values` (described in this section), the `let-values`, `let*-values`, `letrec-values`, and `define-values` forms (among others) create continuations that receive multiple values.

```
(values v ...) → any
  v : any/c
```

Returns the given *v*s. That is, `values` returns as provided arguments.

Examples:

```
> (values 1)
1
> (values 1 2 3)
1
2
3
> (values)
```

```
(call-with-values generator receiver) → any
  generator : (-> any)
  receiver : procedure?
```

Calls *generator*, and passes the values that *generator* produces as arguments to *receiver*. Thus, `call-with-values` creates a continuation that accepts any number of values that *receiver* can accept. The *receiver* procedure is called in tail position with respect to the `call-with-values` call.

Examples:

```
> (call-with-values (lambda () (values 1 2)) +)
3
> (call-with-values (lambda () 1) (lambda (x y) (+ x y)))
#<procedure>: expects 2 arguments, given 1: 1
```

9.2 Exceptions

See §1.1.15 "Exceptions" for information on the PLT Scheme exception model. It is based on a proposal by Friedman, Haynes, and Dybvig [Friedman95].

Whenever a primitive error occurs in PLT Scheme, 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`).

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

9.2.1 Raising Exceptions

```
(raise v [barrier?]) → any
  v : any/c
  barrier? : any/c = #t
```

Raises an exception, where *v* represents the exception being raised. The *v* argument can be anything; it is passed to the current *exception handler*.

If *barrier?* is true, then the call to the exception handler is protected by a continuation barrier, so that multiple returns/escapes are impossible. All exceptions raised by `scheme` functions effectively use `raise` with a `#t` value for *barrier?*.

Breaks are disabled from the time the exception is raised until the exception handler obtains control, and the handler itself is `parameterize-breaked` to disable breaks initially; see §9.6 "Breaks" for more information on breaks.

```
(error sym) → any
  sym : symbol?
(error msg v ...) → any
  msg : string?
  v : any/c
(error src format v ...) → any
  src : symbol?
  format : string?
  v : any/c
```

Raises the exception `exn:fail`, which contains an error string. The different forms produce the error string in different ways:

- `(error sym)` creates a message string by concatenating `"error: "` with the string form of *sym*.

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

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

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

```
(raise-user-error sym) → any
  sym : symbol?
(raise-user-error msg v ...) → any
  msg : string?
  v : any/c
(raise-user-error src format v ...) → any
  src : symbol?
  format : string?
  v : any/c
```

Like `error`, but constructs an exception with `make-exn:fail:user` instead of `make-exn:fail`. The default error display handler does not show a "stack trace" for `exn:fail:user` exceptions (see §9.5 "Continuation Marks"), so `raise-user-error` should be used for errors that are intended for end users.

```
(raise-type-error name expected v) → any
  name : symbol?
  expected : string?
  v : any/c
(raise-type-error name expected bad-pos v) → any
  name : symbol?
  expected : string?
  bad-pos : nonnegative-exact-integer?
  v : any/c
```

Creates an `exn:fail:contract` value and `raise`s it as an exception. The *name* argument is used as the source procedure's name in the error message. The *expected* argument is used as a description of the expected type.

In the first form, *v* is the value received by the procedure that does not have the expected type.

In the second form, the bad argument is indicated by an index *bad-pos* (counting from 0), and all of the original arguments *v* are provided (in order). The resulting error message names the bad argument and also lists the other arguments. If *bad-pos* is not less than the number of *v*s, the `exn:fail:contract` exception is raised.

```
(raise-mismatch-error name message v) → any
  name : symbol?
  message : string?
  v : any/c
```

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

```
(raise-arity-error name arity-v [arg-v ...]) → any
  name : (or/c symbol? procedure?)
  arity-v : (or/c exact-nonnegative-integer?
                  arity-at-least?
                  (listof
                   (or/c exact-nonnegative-integer?
                         arity-at-least?)))
  arg-v : any/c = #f
```

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

```
(raise-syntax-error  name
                     message
                    [expr
                     sub-expr]) → any
  name : (or/c symbol? false/c)
  message : string?
  expr : any/c = #f
  sub-expr : any/c = #f
```

Creates an `exn:fail:syntax` value and `raise`s it as an exception. Macros use this procedure to report syntax errors.

The *name* argument is usually #f when *expr* is provided; it is described in more detail below. The *message* is used as the main body of the error message.

The optional *expr* argument is the erroneous source syntax object or S-expression. The optional *sub-expr* argument is a syntax object or S-expression within *expr* that more precisely locates the error. If *sub-expr* is provided, it is used (in syntax form) for the exprs field of the generated exception record, else the *expr* is used if provided, otherwise the exprs field is the empty list. Source location information in the error-message text is similarly extracted from *sub-expr* or *expr*, when at least one is a syntax object.

The form name used in the generated error message is determined through a combination of the *name*, *expr*, and *sub-expr* arguments:

- When *name* is #f, and when *expr* is either an identifier or a syntax pair containing an identifier as its first element, then the form name from the error message is the identifier's symbol.
- When *name* is #f and when *expr* is not an identifier or a syntax pair containing and identifier as its first element, then the form name in the error message is "?".
- symbol: When *name* is a symbol, then the symbol is used as the form name in the generated error message.

See also error-print-source-location.

9.2.2 Handling Exceptions

```
(call-with-exception-handler f thunk) → any
  f : (any/c . -> . any)
  thunk : (-> any)
```

Installs *f* as the exception handler for the dynamic extent of the call to *thunk*. If an exception is raised during the evaluation of *thunk* (in an extension of the current continuation that does not have its own exception handler), then *f* is applied to the raised value in the continuation of the raise call (but normally extended with a continuation barrier; see §1.1.12 "Prompts, Delimited Continuations, and Barriers" and raise).

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

```
(uncaught-exception-handler) → (any/c . -> . any)
(uncaught-exception-handler f) → void?
  f : (any/c . -> . any)
```

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

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

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

```
(with-handlers ((pred-expr handler-expr))
  body ...+)
```

Evaluates each *`pred-expr`* and and *`handler-expr`* in the order that they are specified, and then evaluates the *`body`*s with a new exception handler during the its dynamic extent.

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

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

Before any predicate or handler procedure is invoked, the continuation of the entire `with-handlers` expression is restored, but also `parameterize-break`ed to disable breaks.

Thus, breaks are disabled by default during the predicate and handler procedures (see §9.6 "Breaks"), and the exception handler is the one from the continuation of the `with-handlers` expression.

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

```
(with-handlers* ((pred-expr handler-expr))
  body ...+)
```

Like `with-handlers`, but if a *`handler-expr`* procedure is called, breaks are not explicitly disabled, and the call is in tail position with respect to the `with-handlers*` form.

9.2.3 Configuring Default Handling

```
(error-escape-handler) → (-> any)
(error-escape-handler proc) → void?
  proc : (-> any)
```

A parameter for the *error escape handler*, which takes no arguments and escapes from the dynamic context of an exception. The default error escape handler escapes using `(abort-current-continuation (default-continuation-prompt-tag) void)`.

The error escape handler is normally called directly by an exception handler, in a parameterization that sets the error display handler and error escape handler to the default handlers, and it is normally `parameterize-breaked` to disable breaks. To escape from a run-time error in a different context, use `raise` or `error`.

Due to a continuation barrier around exception-handling calls, an error escape handler cannot invoke a full continuation that was created prior to the exception, but it can abort to a prompt (see `call-with-continuation-prompt`) or invoke an escape continuation (see `call-with-escape-continuation`).

```
(error-display-handler) → (string? any/c . -> . any)
(error-display-handler proc) → void?
  proc : (string? any/c . -> . any)
```

A parameter for the *error display handler*, which is called by the default exception handler with an error message and the exception value. More generally, the handler's first argument is a string to print as an error message, and the second is a value representing a raised exception.

The default error display handler `display`s its first argument to the current error port (determined by the `current-error-port` parameter) and extracts a stack trace (see `continuation-mark-set->context`) to display from the second argument if it is an `exn` value but not an `exn:fail:user` value.

The default error display handler in DrScheme also uses the second argument to highlight source locations.

To report a run-time error, use `raise` or procedures like `error`, instead of calling the error display procedure directly.

```
(error-print-width) → (and exact-integer? (>=/c 3))
(error-print-width width) → void?
  width : (and exact-integer? (>=/c 3))
```

A parameter whose value is used as the maximum number of characters used to print a Scheme value that is embedded in a primitive error message.

```
(error-print-context-length) → nonnegative-exact-integer?
(error-print-context-length cnt) → void?
  cnt : nonnegative-exact-integer?
```

A parameter whose value is used by the default error display handler as the maximum number of lines of context (or "stack trace") to print; a single "..." line is printed if more lines are available after the first *cnt* lines. A `0` value for *cnt* disables context printing entirely.

```
(error-value->string-handler)
 → (any/c nonnegative-exact-integer?
          . -> .
          string?)
(error-value->string-handler proc) → void?
  proc : (any/c nonnegative-exact-integer?
                . -> .
                string?)
```

A parameter that determines the *error value conversion handler*, which is used to print a Scheme value that is embedded in a primitive error message.

The integer argument to the handler specifies the maximum number of characters that should be used to represent the value in the resulting string. The default error value conversion handler `print`s the value into a string (using the current global port print handler; see `global-port-print-handler`). If the printed form is too long, the printed form is truncated and the last three characters of the return string are set to "...".

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

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

```
(error-print-source-location) → boolean?
(error-print-source-location include?) → void?
  include? : any/c
```

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

9.2.4 Built-in Exception Types

```
(struct exn (message continuation-marks))
        #:transparent)
  message : string?
  continuation-marks : continuation-mark-set?
```

The base structure type for exceptions. The `message` field contains an error message, and the `continuation-marks` field contains the value produced by `(current-continuation-marks)` immediately before the exception was raised.

```
(struct (exn:fail exn) ())
        #:transparent)
```

Raised for exceptions that represent errors, as opposed to `exn:break`.

```
(struct (exn:fail:contract exn:fail) ())
        #:transparent)
```

Raised for errors from the inappropriate run-time use of a function or syntactic form.

```
(struct (exn:fail:contract:arity exn:fail:contract) ())
        #:transparent)
```

Raised when a procedure is applied to the wrong number of arguments.

```
(struct (exn:fail:contract:divide-by-zero exn:fail:contract) ())
        #:transparent)
```

Raised for division by exact zero.

```
(struct (exn:fail:contract:continuation exn:fail:contract) ())
        #:transparent)
```

Raised when a continuation is applied where the jump would cross a continuation barrier.

```
(struct (exn:fail:contract:variable exn:fail:contract) (id))
        #:transparent)
  id : symbol?
```

Raised for a reference to a not-yet-defined top-level variable or module-level variable.

```
(struct (exn:fail:syntax exn:fail) (exprs))
        #:transparent)
  exprs : (listof syntax?)
```

Raised for a syntax error that is not a `read` error. The `exprs` indicate the relevant source expressions, least-specific to most-specific.

```
(struct (exn:fail:read exn:fail) (srclocs))
        #:transparent)
  srclocs : (listof srcloc?)
```

Raised for a `read` error. The `srclocs` indicate the relevant source expressions.

```
(struct (exn:fail:read:eof exn:fail:read) ())
        #:transparent)
```

Raised for a `read` error, specifically when the error is due to an unexpected end-of-file.

```
(struct (exn:fail:read:non-char exn:fail:read) ()
        #:transparent)
```

Raised for a `read` error, specifically when the error is due to an unexpected non-character (i.e., “special”) element in the input stream.

```
(struct (exn:fail:filesystem exn:fail) ()
        #:transparent)
```

Raised for an error related to the filesystem (such as a file not found).

```
(struct (exn:fail:filesystem:exists exn:fail:filesystem) ()
        #:transparent)
```

Raised for an error when attempting to create a file that exists already.

```
(struct (exn:fail:filesystem:version exn:fail:filesystem) ()
        #:transparent)
```

Raised for a version-mismatch error when loading an extension.

```
(struct (exn:fail:network exn:fail) ()
        #:transparent)
```

Raised for TCP and UDP errors.

```
(struct (exn:fail:out-of-memory exn:fail) ()
        #:transparent)
```

Raised for an error due to insufficient memory, in cases where sufficient memory is at least available for raising the exception.

```
(struct (exn:fail:unsupported exn:fail) ()
        #:transparent)
```

Raised for an error due to an unsupported feature on the current platform or configuration.

```
(struct (exn:fail:user exn:fail) ()
        #:transparent)
```

Raised for errors that are intended to be seen by end-users. In particular, the default error printer does not show the program context when printing the error message.

```
(struct (exn:break exn) (continuation))
        #:transparent)
  continuation : continuation?
```

Raised asynchronously (when enabled) in response to a break request. The `continuation` field can be used by a handler to resume the interrupted computation.

```
prop:exn:srclocs : struct-type-property?
```

A property that identifiers structure types that provide a list of `srcloc` values. The property is normally attached to structure types used to represent exception information.

The property value must be a procedure that accepts a single value—the structure type instance from which to extract source locations—and returns a list of `srcloc`s.

```
(exn:srclocs? v) → boolean?
  v : any/c
```

Returns `#t` if `v` has the `prop:exn:srclocs` property, `#f` otherwise.

```
(exn:srclocs-accessor v) → (exn:srclocs?. ->    listof srcloc)
  v : exn:srclocs?
```

Returns the `srcloc`-getting procedure associated with `v`.

```
(struct srcloc (source line column position span))
        #:transparent)
  source : any/c
  line : (or/c positive-exact-integer? false/c)
  column : (or/c nonnegative-exact-integer? false/c)
  position : (or/c positive-exact-integer? false/c)
  span : (or/c nonnegative-exact-integer? false/c)
```

The fields of an `srcloc` instance are as follows:

- `source` — An arbitrary value identifying the source, often a path (see §14.1 “Paths”).
- `line` — The line number (counts from 1) or `#f` (unknown).
- `column` — The column number (counts from 0) or `#f` (unknown).

- `position` — The starting position (counts from 1) or `#f` (unknown).
- `span` — The number of covered positions (counts from 0) or `#f` (unknown).

9.3 Delayed Evaluation

```
(require scheme/promise)
```

The bindings documented in this section are provided by the `scheme/promise` and `scheme` libraries, but not `scheme/base`.

A *promise* encapsulates an expression to be evaluated on demand via `force`. After a promise has been `force`d, every later `force` of the promise produces the same result.

```
(promise? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a promise, `#f` otherwise.

```
(delay expr)
```

Creates a promise that, when `force`d, evaluates *expr* to produce its value.

```
(lazy expr)
```

Like `delay`, except that if *expr* produces a promise, then the promise is `force`d to obtain a value.

```
(force v) → any
  v : any/c
```

If *v* is a promise, then the promise is forced to obtain a value. If the promise has not been forced before, then the result is recorded in the promise so that future `force`s on the promise produce the same value (or values). If forcing the promise raises an exception, then the exception is similarly recorded so that forcing the promise will raise the same exception every time.

If *v* is `force`d again before the original call to `force` returns, then the `exn:fail` exception is raised.

If *v* is not a promise, then it is returned as the result.

9.4 Continuations

See §1.1.1 “Sub-expression Evaluation and Continuations” and §1.1.12 “Prompts, Delimited Continuations, and Barriers” for general information about continuations. PLT Scheme’s support for prompts and composable continuations most closely resembles Dorai Sitaram’s `%` and `fcontrol` operator [Sitaram93].

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

- applying an exception handler, an error escape handler, or an error display handler (see §9.2 “Exceptions”);
- applying a macro transformer (see §11.4 “Syntax Transformers”), evaluating a compile-time expression, or applying a module name resolver (see §13.4.1 “Resolving Module Names”);
- applying a custom-port procedure (see §12.1.9 “Custom Ports”), an event guard procedure (see §10.2.1 “Events”), or a parameter guard procedure (see §10.3.2 “Parameters”);
- applying a security-guard procedure (see §13.5 “Security Guards”);
- applying a will procedure (see §15.3 “Wills and Executors”); or
- evaluating or loading code from the stand-alone MzScheme command line (see §16.1 “Starting MzScheme or MrEd”).

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

```
(call-with-continuation-prompt  thunk
                               [prompt-tag
                                handler])   → any
  thunk : (-> any)
  prompt-tag : continuation-prompt-tag?
             = (default-continuation-prompt-tag)
  handler : (or/c procedure? false/c) = #f
```

Calls *thunk* with the current continuation extended by a prompt. The prompt is tagged by *prompt-tag*, which must be a result from either `default-continuation-prompt-tag` (the default) or `make-continuation-prompt-tag`. The result of *thunk* is the result of the `call-with-continuation-prompt` call.

The *handler* argument specifies a handler procedure to be called in tail position with respect to the `call-with-continuation-prompt` call when the installed prompt is the target of a `abort-current-continuation` call with *prompt-tag*; the remaining arguments of `abort-current-continuation` are supplied to the handler procedure. If *handler* is `#f`, the default handler accepts a single `abort-thunk` argument and calls `(call-with-continuation-prompt abort-thunk prompt-tag #f)`; that is, the default handler reinstalls the prompt and continues with a given thunk.

```
(abort-current-continuation prompt-tag
                            v ...+)     → any
  prompt-tag : any/c
  v : any/c
```

Resets the current continuation to that of the nearest prompt tagged by *prompt-tag* in the current continuation; if no such prompt exists, the `exn:fail:contract:continuation` exception is raised. The *v*s are delivered as arguments to the target prompt's handler procedure.

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

```
(make-continuation-prompt-tag) → continuation-prompt-tag?
(make-continuation-prompt-tag sym) → continuation-prompt-tag?
  sym : symbol?
```

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

```
(default-continuation-prompt-tag) → continuation-prompt-tag?
```

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

```
(call-with-current-continuation proc
                                [prompt-tag]) → any
  proc : (continuation? . -> . any)
```

```
  prompt-tag : continuation-prompt-tag?
             = (default-continuation-prompt-tag)
```

Captures the current continuation up to the nearest prompt tagged by *prompt-tag*; if no such prompt exists, the `exn:fail:contract:continuation` exception is raised. The truncated continuation includes only continuation marks and `dynamic-wind` frames installed since the prompt.

The capture continuation is delivered to *proc*, which is called in tail position with respect to the `call-with-current-continuation` call.

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

The arguments supplied to an applied procedure become the result values for the restored continuation. In particular, if multiple arguments are supplied, then the continuation receives multiple results.

If, at application time, a continuation barrier appears between the current continuation and the prompt tagged with *prompt-tag*, and if the same barrier is not part of the captured continuation, then the `exn:fail:contract:continuation` exception is raised.

A continuation can be invoked from the thread (see §10.1 “Threads”) other than the one where it was captured.

```
(call/cc proc [prompt-tag]) → any
  proc : (continuation? . -> . any)
  prompt-tag : continuation-prompt-tag?
             = (default-continuation-prompt-tag)
```

The `call/cc` binding is an alias for `call-with-current-continuation`.

```
(call-with-composable-continuation  proc
                                   [prompt-tag]) → any
  proc : (continuation? . -> . any)
  prompt-tag : continuation-prompt-tag?
             = (default-continuation-prompt-tag)
```

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

```
(call-with-escape-continuation  proc
                               [prompt-tag]) → any
  proc : (continuation? . -> . any)
  prompt-tag : continuation-prompt-tag?
             = (default-continuation-prompt-tag)
```

Like `call-with-current-continuation`, but *proc* is not called in tail position, and the continuation procedure supplied to *proc* can only be called during the dynamic extent of the `call-with-escape-continuation` call. A continuation barrier, however, never prevents the application of the continuation.

Due to the limited applicability of its continuation, `call-with-escape-continuation` can be implemented more efficiently than `call-with-current-continuation`.

A continuation obtained from `call-with-escape-continuation` is actually a kind of prompt. Escape continuations are provided mainly for backward compatibility, since they pre-date general prompts in MzScheme, and because `call/ec` is often an easy replacement for `call/cc` to improve performance.

```
(call/ec proc [prompt-tag]) → any
  proc : (continuation? . -> . any)
  prompt-tag : continuation-prompt-tag?
             = (default-continuation-prompt-tag)
```

The `call/ec` binding is an alias for `call-with-escape-continuation`.

```
(let/cc k body ...+)
```

Equivalent to `(call/cc (lambda (k) body ...))`.

```
(let/ec k body ...+)
```

Equivalent to `(call/ec (lambda (k) body ...))`.

```
(call-with-continuation-barrier thunk) → any
  thunk : (-> any)
```

Applies *thunk* with a barrier between the application and the current continuation. The results of *thunk* are the results of the `call-with-continuation-barrier` call.

```
(continuation-prompt-available? prompt-tag
                                [cont])      → any
  prompt-tag : continuation-prompt-tag?
  cont : continuation? = (call/cc values)
```

Returns `#t` if *cont*, which must be a continuation, includes a prompt tagged by *prompt-tag*, `#f` otherwise.

```
(continuation? v) → boolean?
  v : any/c
```

Return `#t` if *v* is a continuation as produced by `call-with-current-continuation`, `call-with-composable-continuation`, or `call-with-escape-continuation`, `#f` otherwise.

```
(continuation-prompt-tag? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a continuation prompt tag as produced by `default-continuation-prompt-tag` or `make-continuation-prompt-tag`.

```
(dynamic-wind pre-thunk
              value-thunk
              post-thunk) → any
  pre-thunk : (-> any)
  value-thunk : (-> any)
  post-thunk : (-> any)
```

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

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

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

- jumps to a more deeply nested *pre-thunk*, if any, or jumps to the destination continuation; then
- continues with the context of the *pre-thunk*'s dynamic-wind call.

Normally, the second part of this continuation is never reached, due to a jump in the first part. However, the second part is relevant because it enables jumps to escape continuations that are contained in the context of the dynamic-wind call. Furthermore, it means that the continuation marks (see §9.5 "Continuation Marks") and parameterization (see §10.3.2 "Parameters") for *pre-thunk* correspond to those of the dynamic-wind call that installed *pre-thunk*. The *pre-thunk* call, however, is parameterize-breaked to disable breaks (see also §9.6 "Breaks").

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

In both cases, the target for a jump is recomputed after each *pre-thunk* or *post-thunk* completes. When a prompt-delimited continuation (see §1.1.12 "Prompts, Delimited Continuations, and Barriers") is captured in a *post-thunk*, it might be delimited and instantiated in such a way that the target of a jump turns out to be different when the continuation is applied than when the continuation was captured. There may even be no appropriate target, if a relevant prompt or escape continuation is not in the continuation after the restore; in that case, the first step in a *pre-thunk* or *post-thunk*'s continuation can raise an exception.

Examples:

```
> (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 ")))
in pre out in post out
> (let/ec k0
    (let/ec k1
```

```
       (dynamic-wind
        void
        (lambda () (k0 'cancel))
        (lambda () (k1 'cancel-canceled)))))
cancel-canceled
> (let* ([x (make-parameter 0)]
         [l null]
         [add (lambda (a b)
                (set! l (append l (list (cons a b)))))])
    (let ([k (parameterize ([x 5])
               (dynamic-wind
                   (lambda () (add 1 (x)))
                   (lambda () (parameterize ([x 6])
                                (let ([k+e (let/cc k (cons k void))])
                                  (add 2 (x))
                                  ((cdr k+e))
                                  (car k+e))))
                   (lambda () (add 3 (x)))))])
      (parameterize ([x 7])
        (let/cc esc
          (k (cons void esc)))))
    l)
((1 . 5) (2 . 6) (3 . 5) (1 . 5) (2 . 6) (3 . 5))
```

9.4.1 Classical Control Operators

```
(require scheme/control)
```

The bindings documented in this section are provided by the `scheme/control` library, not `scheme/base` or `scheme`.

The `scheme/control` library provides various control operators from the research literature on higher-order control operators, plus a few extra convenience forms. These control operators are implemented in terms of `call-with-continuation-prompt`, `call-with-composable-continuations`, etc., and they generally work sensibly together. Many are redundant; for example, `reset` and `shift` are aliases.

```
(abort v ...) → any
  v : any/c
```

Returns the *v*s to a prompt using the default continuation prompt tag and the default abort handler.

That is, `(abort v ...)` is equivalent to

```
(abort-current-continuation
 (default-continuation-prompt-tag)
 (lambda () (values v ...)))
```

```
(% expr)
(% expr handler-expr)
(fcontrol v) → any
  v : any/c
```

Sitaram's operators [Sitaram93].

The essential reduction rules are:

```
(% val proc) => val
(% E[(fcontrol val)] proc) => (proc val (lambda (x) E[x]))
  ; where E has no %
```

When *handler-expr* is omitted, `%` is the same as `prompt`.

```
(prompt expr ...+)
(control id expr ...+)
```

Among the earliest operators for higher-order control [Felleisen88, Sitaram90].

The essential reduction rules are:

```
(prompt val) => val
(prompt E[(control k expr)]) => (prompt ((lambda (k) expr)
                                         (lambda (v) E[v])))
  ; where E has no prompt
```

```
(prompt-at prompt-tag-expr expr ...+)
(control-at prompt-tag-expr id expr ...+)
```

Like `prompt` and `control`, but using specific prompt tags:

```
(prompt-at tag val) => val
(prompt-at tag E[(control-at tag k expr)]) => (prompt-at tag
                                               ((lambda (k) expr)
                                                (lambda (v) E[v])))
  ; where E has no prompt-at for tag
```

```
(reset expr ...+)
(shift id expr ...+)
```

Danvy and Filinski's operators [Danvy90].

The essential reduction rules are:

```
(reset val) => val
(reset E[(shift k expr)]) => (reset ((lambda (k) expr)
                                     (lambda (v) (reset E[v]))))
  ; where E has no reset
```

The `reset` and `prompt` forms are interchangeable.

```
(reset-at prompt-tag-expr expr ...+)
(shift-at prompt-tag-expr identifer expr ...+)
```

Like `reset` and `shift`, but using the specified prompt tags.

```
(prompt0 expr ...+)
(reset0 expr ...+)
(control0 id expr ...+)
(shift0 id expr ...+)
```

Generalizations of `prompt`, etc. [Shan04].

The essential reduction rules are:

```
(prompt0 val) => val
(prompt0 E[(control0 k expr)]) => ((lambda (k) expr)
                                   (lambda (v) E[v]))
(reset0 val) => val
(reset0 E[(shift0 k expr)]) => ((lambda (k) expr)
                                (lambda (v) (reset0 E[v])))
```

The `reset0` and `prompt0` forms are interchangable. Furthermore, the following reductions apply:

```
(prompt E[(control0 k expr)]) => (prompt ((lambda (k) expr)
                                          (lambda (v) E[v])))
(reset E[(shift0 k expr)]) => (reset ((lambda (k) expr)
                                      (lambda (v) (reset0 E[v]))))
(prompt0 E[(control k expr)]) => (prompt0 ((lambda (k) expr)
                                           (lambda (v) E[v])))
(reset0 E[(shift k expr)]) => (reset0 ((lambda (k) expr)
                                       (lambda (v) (reset E[v]))))
```

That is, both the `prompt`/`reset` and `control`/`shift` sites must agree for `0`-like behavior, otherwise the non-0 behavior applies.

```
(prompt0-at prompt-tag-expr expr ...+)
(reset0-at prompt-tag-expr expr ...+)
(control0-at prompt-tag-expr id expr ...+)
(shift0-at prompt-tag-expr id expr ...+)
```

Variants of `prompt0`, etc. that accept a prompt tag.

```
(spawn proc) → any
  proc : ((any/c . -> . any) . -> . any)
```

The operators of Hieb and Dybvig [Hieb90].

The essential reduction rules are:

```
(prompt-at tag obj) => obj
(spawn proc) => (prompt tag (proc (lambda (x) (abort tag x))))
(prompt-at tag E[(abort tag proc)])
  => (proc (lambda (x) (prompt-at tag E[x])))
  ; where E has no prompt-at for tag
```

```
(splitter proc) → any
  proc : (((-> any) . -> . any)
          ((continuation? . -> . any) . -> . any)
          . -> . any)
```

The operator of Queinnec and Serpette [Queinnec91].

The essential reduction rules are:

```
(splitter proc) => (prompt-at tag
                     (proc (lambda (thunk)
                             (abort tag thunk))
                           (lambda (proc)
                             (control0-at tag k (proc k)))))
(prompt-at tag E[(abort tag thunk)]) => (thunk)
  ; where E has no prompt-at for tag
(prompt-at tag E[(control0-at tag k expr)]) => ((lambda (k) expr)
                                                 (lambda (x) E[x]))
  ; where E has no prompt-at for tag
```

```
(new-prompt) → any
```

```
(set prompt-expr expr ...+)
(cupto prompt-expr id expr ...+)
```

The operators of Gunter et al. [Gunter95].

In this library, `new-prompt` is an alias for `make-continuation-prompt-tag`, `set` is an alias for `prompt0-at`, and `cupto` is an alias for `control0-at`.

9.5 Continuation Marks

See §1.1.11 "Continuation Frames and Marks" and §1.1.12 "Prompts, Delimited Continuations, and Barriers" for general information about continuation marks.

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

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

The `with-continuation-mark` form installs a mark on the first frame of the current continuation (see §2.17 "Continuation Marks: `with-continuation-mark`"). Procedures such as `current-continuation-marks` allow inspection of marks.

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

When a continuation procedure returned by `call-with-current-continuation` or `call-with-composable-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` or `call-with-composable-continuation` was invoked.

```
(continuation-marks cont [prompt-tag]) → continuation-mark-set?
  cont : continuation?
  prompt-tag : prompt-tag? = (default-continuation-prompt-tag)
```

Returns an opaque value containing the set of continuation marks for all keys in the continuation *cont* up to the prompt tagged by *prompt-tag*. If *cont* is an escape continuation (see §1.1.12 "Prompts, Delimited Continuations, and Barriers"), then the current continuation must extend *cont*, or the `exn:fail:contract` exception is raised. If *cont* was not captured with respect to *prompt-tag* and does not include a prompt for *prompt-tag*, the `exn:fail:contract` exception is raised.

```
(current-continuation-marks [prompt-tag])
 → continuation-mark-set?
  prompt-tag : prompt-tag? = (default-continuation-prompt-tag)
```

Returns an opaque value containing the set of continuation marks for all keys in the current continuation up to *prompt-tag*. In other words, it produces the same value as

```
(call-with-current-continuation
  (lambda (k)
    (continuation-marks k prompt-tag))
  prompt-tag)
```

```
(continuation-mark-set->list  mark-set
                              key-v
                             [prompt-tag]) → list?
  mark-set : continuation-mark-set?
  key-v : any/c
  prompt-tag : prompt-tag? = (default-continuation-prompt-tag)
```

Returns a newly-created list containing the marks for *key-v* in *mark-set*, which is a set of marks returned by `current-continuation-marks`. The result list is truncated at the first point, if any, where continuation frames were originally separated by a prompt tagged with *prompt-tag*..

```
(continuation-mark-set->list*  mark-set
                               key-v
                              [none-v
                               prompt-tag]) → (listof vector?)
  mark-set : continuation-mark-set?
  key-v : any/c
  none-v : any/c = #f
  prompt-tag : prompt-tag? = (default-continuation-prompt-tag)
```

Returns a newly-created list containing vectors of marks in *mark-set* for the keys in `key-list`, up to *prompt-tag*. The length of each vector in the result list is the same as the length of `key-list`, and a value in a particular vector position is the value for the corresponding key in `key-list`. Values for multiple keys appear in a single vector only when the marks are for the same continuation frame in *mark-set*. The *none-v* argument is used for vector elements to indicate the lack of a value.

```
(continuation-mark-set-first mark-set
                             key-v
                            [prompt-tag]) → any
  mark-set : (or/c continuation-mark-set? false/c)
  key-v : any/c
  prompt-tag : prompt-tag? = (default-continuation-prompt-tag)
```

Returns the first element of the list that would be returned by `(continuation-mark-set->list (or mark-set (current-continuation-marks prompt-tag)) key-v prompt-tag)`, or `#f` if the result would be the empty list. Typically, this result can be computed more quickly using `continuation-mark-set-first`.

```
(continuation-mark-set? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a mark set created by `continuation-marks` or `current-continuation-marks`, `#f` otherwise.

```
(continuation-mark-set->context mark-set) → list?
  mark-set : continuation-mark-set?
```

Returns a list representing a "stack trace" for *mark-set*'s continuation. The list contains pairs, where the `car` of each pair contains either `#f` or a symbol for a procedure name, and the `cdr` of each pair contains either `#f` or a `srcloc` value for the procedure's source location (see §12.1.4 "Counting Positions, Lines, and Columns"); the `car` and `cdr` are never both `#f`.

The stack-trace list is the result of `continuation-mark-set->list` with *mark-set* and Scheme's private key for procedure-call marks. A stack trace is extracted from an exception and displayed by the default error display handler (see `current-error-display-handler`) for exceptions other than `exn:fail:user` (see `raise-user-error` in §9.2.1 "Raising Exceptions").

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 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))))
((mark2 mark1))
> (let loop ([n 1000])
    (if (zero? n)
        (extract-current-continuation-marks 'key)
        (with-continuation-mark 'key n
          (loop (sub1 n)))))
(1)
```

9.6 Breaks

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

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

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

Before calling a `with-handlers` predicate or handler, an exception handler, an error display handler, an error escape handler, an error value conversion handler, or a `pre-thunk` or `post-thunk` for a `dynamic-wind`, the call is `parameterize-break`ed to disable breaks. Furthermore, breaks are disabled during the transitions among handlers related to exceptions, during the transitions between `pre-thunk`s and `post-thunk`s for `dynamic-wind`, and during other transitions for a continuation jump. For example, if breaks are disabled when a continuation is invoked, and if breaks are also disabled in the target continuation, then breaks will remain disabled until from the time of the invocation until the target continuation executes unless a relevant `dynamic-wind` `pre-thunk` or `post-thunk` explicitly enables breaks.

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

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

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

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

In general, it is impossible using only `semaphore-wait` to implement the guarantee that either the semaphore is decremented or an exception is raised, but not both. Scheme therefore supplies `semaphore-wait/enable-break` (see §10.2.3 "Semaphores"), which does permit the implementation of such an exclusive guarantee:

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

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

To allow similar implementation patterns over blocking port operations, MzScheme provides `read-bytes-avail!/enable-break`, `write-bytes-avail/enable-break`, and other procedures.

```
(break-enabled) → boolean?
(break-enabled on?) → void?
  on? : any/c
```

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

```
(parameterize-break boolean-expr body ...+)
```

Evaluates `boolean-expr` to determine whether breaks are initially enabled in while evaluating the `body`s in sequence. The result of the `parameter-break` expression is the result of the last `expr`.

Like `parameterize` (see §10.3.2 "Parameters"), a fresh thread cell (see §10.3.1 "Thread Cells") is allocated to hold the break-enabled state of the continuation, and calls to `break-enabled` within the continuation access or modify the new cell. Unlike parameters, the break setting is not inherited by new threads.

```
(current-break-parameterization) → break-parameterization?
```

Analogous to `(current-parameterization)` (see §10.3.2 "Parameters"); it returns a break-parameterization (effectively a thread cell) that holds the current continuation's break-enable state.

```
(call-with-break-parameterization break-param
                                  thunk)      → any
  break-param : break-parameterization?
  thunk : (-> any)
```

Analogous to `(call-with-parameterization parameterization thunk)` (see §10.3.2 "Parameters"), calls `thunk` in a continuation whose break-enabled state is in `break-param`. The `thunk` is *not* called in tail position with respect to the `call-with-break-parameterization` call.

9.7 Exiting

```
(exit [v]) → any
  v : any/c = #t
```

Passes *v* to the current exit handler. If the exit handler does not escape or terminate the thread, `#<void>` is returned.

```
(exit-handler) → (any/c . -> . any)
```

```
(exit-handler proc) → void?
  proc : (any/c . -> . any)
```

A parameter that determines the current *exit handler*. The exit handler is called by `exit`.

The default exit handler in the `mzscheme` executable takes any argument and shuts down the OS-level Scheme process. The argument is used as the OS-level exit code if it is an exact integer between `1` and `255` (which normally means “failure”); otherwise, the exit code is `0`, (which normally means “success”).

10 Concurrency

PLT Scheme supports multiple threads of control within a program. Threads run concurrently, in the sense that one thread can preempt another without its cooperation, but threads currently all run on the same processor (i.e., the same underlying OS process and thread).

10.1 Threads

See §1.1.13 “Threads” for basic information on the PLT Scheme thread model.

When a thread is created, it is placed into the management of the current custodian and added to the current thread group (see §13.7 “Thread Groups”). A thread can have any number of custodian managers added through `thread-resume`.

A thread that has not terminated can be garbage collected (see §1.1.7 “Garbage Collection”) if it is unreachable and suspended, or if it is unreachable and blocked on a set of unreachable events through `semaphore-wait` or `semaphore-wait/enable-break`, `channel-put` or `channel-get`, `sync` or `sync/enable-break`, or `thread-wait`.

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

All constant-time procedures and operations provided by MzScheme are thread-safe because they are *atomic*. For example, `set!` assigns to a variable as an atomic action with respect to all threads, so that no thread can see a “half-assigned” variable. Similarly, `vector-set!` assigns to a vector atomically. The `hash-set!` procedure is not atomic, but the table is protected by a lock; see §3.13 “Hash Tables” for more information. Port operations are generally not atomic, but they are thread-safe in the sense that a byte consumed by one thread from an input port will not be returned also to another thread, and procedures like `port-commit-peeked` and `write-bytes-avail` offer specific concurrency guarantees.

10.1.1 Creating Threads

```
(thread thunk) → thread?
  thunk : (-> any)
```

Calls *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.

```
(thread? v) → thread?
  v : any/c
```

Returns `#t` if *v* is a thread descriptor, `#f` otherwise.

```
(current-thread) → thread?
```

Returns the thread descriptor for the currently executing thread.

```
(thread/suspend-to-kill thunk) → thread
  thunk : (-> any)
```

Like `thread`, except that "killing" the thread through `kill-thread` or `custodian-shutdown-all` merely suspends the thread instead of terminating it.

```
(call-in-nested-thread thunk [cust]) → any
  thunk : (->any)
  cust : custodian? = (current-custodian)
```

Creates a nested thread managed by *cust* to execute *thunk*. (The nested thread's current custodian is inherited from the creating thread, independent of the *cust* argument.) The current thread blocks until *thunk* returns, and the result of the `call-in-nested-thread` call is the result returned by *thunk*.

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

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

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

10.1.2 Suspending, Resuming, and Killing Threads

```
(thread-suspend thd) → void?
  thd : thread?
```

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

```
(thread-resume thd [benefactor]) → void?
  thd : thread?
  benefactor : (or/c thread? custodian? false/c) = #f
```

Resumes the execution of *thd* if it is suspended and has at least one custodian (possibly added through *benefactor*, as described below). If the thread has terminated, or if the thread is already running and *benefactor* is not supplied, or if the thread has no custodian and *benefactor* is not supplied, then `thread-resume` has no effect. Otherwise, if *benefactor* is supplied, it triggers up to three additional actions:

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

```
(kill-thread thd) → void?
  thd : thread?
```

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

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

```
(break-thread thd) → void?
  thd : thread?
```

Registers a break with the specified thread. If breaking is disabled in `thd`, the break will be ignored until breaks are re-enabled (see §9.6 "Breaks").

```
(sleep [secs]) → void?
  secs : nonnegative-number? = 0
```

Causes the current thread to sleep until at least `secs` seconds have passed after it starts sleeping. A zero value for `secs` simply acts as a hint to allow other threads to execute. The value of `secs` can be non-integral to request a sleep duration to any precision; the precision of the actual sleep time is unspecified.

```
(thread-running? thd) → any
  thd : thread?
```

Returns `#t` if `thd` has not terminated and is not suspended, `#f` otherwise.

```
(thread-dead? thd) → any
  thd : thread?
```

Returns `#t` if `thd` has terminated, `#f` otherwise.

10.1.3 Synchronizing Thread State

```
(thread-wait thd) → void?
  thd : thread?
```

Blocks execution of the current thread until `thd` has terminated. Note that `(thread-wait (current-thread))` deadlocks the current thread, but a break can end the deadlock (if breaking is enabled; see §9.6 "Breaks").

```
(thread-dead-evt thd) → evt?
  thd : thread?
```

Returns a synchronizable event (see §10.2.1 "Events") that is ready if and only if `thd` has terminated. Unlike using `thd` directly, however, a reference to the event does not prevent `thd` from being garbage collected (see §1.1.7 "Garbage Collection").

```
(thread-resume-evt thd) → evt?
  thd : thread?
```

Returns a synchronizable event (see §10.2.1 "Events") that becomes ready when `thd` is running. (If `thd` has terminated, the event never becomes ready.) If `thd` runs and is then suspended after a call to `thread-resume-evt`, the result event remains ready; after each suspend of `thd` a fresh event is generated to be returned by `thread-resume-evt`. The result of the event is `thd`, but if `thd` is never resumed, then reference to the event does not prevent `thd` from being garbage collected (see §1.1.7 "Garbage Collection").

```
(thread-suspend-evt thd) → evt?
  thd : thread?
```

Returns a synchronizable event (see §10.2.1 "Events") that becomes ready when `thd` is suspended. (If `thd` has terminated, the event will never unblock.) If `thd` is suspended and then resumes after a call to `thread-suspend-evt`, the result event remains ready; after each resume of `thd` created a fresh event to be returned by `thread-suspend-evt`.

10.1.4 Thread Mailboxes

Each thread has a *mailbox* through which it can receive arbitrary message. In other words, each thread has a built-in asynchronous channel.

See also §10.2.4 "Buffered Asynchronous Channels".

```
(thread-send thd v [fail-thunk]) → any
  thd : thread?
  v : any/c
  fail-thunk : (or/c (-> any) false/c)
             = (lambda () (raise-mismatch-error ....))
```

Queues `v` as a message to `thd` without blocking. If the message is queued, the result is `#<void>`. If `thd` stops running—as in `thread-running?`—before the message is queued, then `fail-thunk` is called (through a tail call) if is a procedure to produce the result, or `#f` is returned if `fail-thunk` is `#f`.

```
(thread-receive) → any/c
```

Receives and dequeues a message queued for the current thread, if any. If no message is available, `thread-receive` blocks until one is available.

```
(thread-try-receive) → any/c
```

Receives and dequeues a message queued for the current thread, if any, or returns `#f` immediately if no message is available.

```
(thread-receive-evt) → evt?
```

Returns a constant synchronizable event (see §10.2.1 "Events") that becomes ready when the synchronizing thread has a message to receive. The event result is itself.

```
(thread-rewind-receive lst) → void?
  lst : list?
```

Pushes the elements of `lst` back onto the front of the current thread's queue. The elements are pushed one by one, so that the first available message is the last element of `lst`.

10.2 Synchronization

Scheme's synchronization toolbox spans three layers:

- synchronizable events — a general framework for synchronization;
- channels — a primitive that can be used, in principle, to build most other kinds of synchronizable events (except the ones that compose events); and
- semaphores — a simple and especially cheap primitive for synchronization.

10.2.1 Events

A *synchronizable event* (or just *event* for short) works with the `sync` procedure to coordinate synchronization among threads. Certain kinds of objects double as events, including ports and threads. Other kinds of objects exist only for their use as events.

At an point in time, an event is either *ready* for synchronization, or it is not; depending on the kind of event and how it is used by other threads, an event can switch from not ready to ready (or back), at any time. If a thread synchronizes on an event when it is ready, then the event produces a particular *synchronization result*.

Synchronizing an event may affect the state of the event. For example, when synchronizing a semaphore, then the semaphore's internal count is decremented, just as with `semaphore-wait`. For most kinds of events, however (such as a port), synchronizing does not modify the event's state.

The following act as events in stand-alone MzScheme. An extension or embedding application can extend the set of primitive events — in particular, an eventspace in MrEd is an event — and new structure types can generate events (see `prop:evt`).

- `semaphore` — a semaphore is ready when `semaphore-wait` would not block. The synchronization result of semaphore is semaphore itself.
- `semaphore-peek` — a semaphore-peek event returned by `semaphore-peek-evt` applied to `semaphore` is ready exactly when `semaphore` is ready. The synchronization result of semaphore-peek is semaphore-peek itself.
- `channel` — a channel returned by `make-channel` is ready when `channel-get` would not block. The channel's result as an event is the same as the `channel-get` result.
- `channel-put` — an event returned by `channel-put-evt` applied to `channel` is ready when `channel-put` would not block on `channel`. The synchronization result of channel-put is channel-put itself.
- `input-port` — an input port is ready as an event when `read-byte` would not block. The synchronization result of input-port is input-port itself.
- `output-port` — an output port is ready when `write-bytes-avail` would not block or when the port contains buffered characters and `write-bytes-avail*` can flush part of the buffer (although `write-bytes-avail` might block). The synchronization result of output-port is output-port itself.
- `progress` — an event produced by `port-progress-evt` applied to `input-port` is ready after any subsequent read from `input-port`. The synchronization result of progress is progress itself.
- `tcp-listener` — a TCP listener is ready when `tcp-accept` would not block. The synchronization result of listener is listener itself.
- `thd` — a thread is ready when `thread-wait` would not block. The synchronization result of thread is thread itself.
- `thread-dead` — an event returned by `thread-dead-evt` applied to `thd` is ready when `thd` has terminated. The synchronization result of thread-dead is thread-dead itself.
- `thread-resume` — an event returned by `thread-resume-evt` applied to `thd` is ready when `thd` subsequently resumes execution (if it was not already running). The event's result is `thd`.
- `thread-suspend` — an event returned by `thread-suspend-evt` applied to `thd` is ready when `thd` subsequently suspends execution (if it was not already suspended). The event's result is `thd`.

- `alarm` — an event returned by `alarm-evt` is ready after a particular date and time. The synchronization result of alarm is alarm itself.
- `subprocess` — a subprocess is ready when `subprocess-wait` would not block. The synchronization result of subprocess is subprocess itself.
- `will-executor` — a will executor is ready when `will-execute` would not block. The synchronization result of will-executor is will-executor itself.
- `udp` — an event returned by `udp-send-evt` or `udp-receive!-evt` is ready when a send or receive on the original socket would block, respectively. The synchronization result of udp is udp itself.
- `choice` — an event returned by `choice-evt` is ready when one or more of the `evt`s supplied to `chocie-evt` are ready. If the choice event is chosen, one of its ready `evt`s is chosen pseudo-randomly, and the result is the chosen `evt`'s result.
- `wrap` — an event returned by `wrap-evt` applied to `evt` and `proc` is ready when `evt` is ready. The event's result is obtained by a call to `proc` (with breaks disabled) on the result of `evt`.
- `handle` — an event returned by `handle-evt` applied to `evt` and `proc` is ready when `evt` is ready. The event's result is obtained by a tail call to `proc` on the result of `evt`.
- `guard` — an event returned by `guard-evt` applied to `thunk` generates a new event every time that `guard` is used with `sync` (or whenever it is part of a choice event used with `sync`, etc.); the generated event is the result of calling `thunk` when the synchronization begins; if `thunk` returns a non-event, then `thunk`'s result is replaced with an event that is ready and whose result is `guard`.
- `nack-guard` — an event returned by `nack-guard-evt` applied to `proc` generates a new event every time that `nack-guard` is used with `sync` (or whenever it is part of a choice event used with `sync`, etc.); the generated event is the result of calling `proc` with a NACK ("negative acknowledgment") event when the synchronization begins; if `proc` returns a non-event, then `proc`'s result is replaced with an event that is ready and whose result is `nack-guard`.

  If the event from `proc` is not ultimately chosen as the unblocked event, then the NACK event supplied to `proc` becomes ready with a `#<void>` value. This NACK event becomes ready when the event is abandoned because some other event is chosen, because the synchronizing thread is dead, or because control escaped from the call to `sync` (even if `nack-guard`'s `proc` has not yet returned a value). If the event returned by `proc` is chosen, then the NACK event never becomes ready.
- `poll-guard` — an event returned by `poll-guard-evt` applied to `proc` generates a new event every time that `poll-guard` is used with `sync` (or whenever it is part of a choice event used with `sync`, etc.); the generated event is the result of calling `proc` with a boolean: `#t` if the event will be used for a poll, `#f` for a blocking synchronization.

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

- `struct` — a structure whose type has the `prop:evt` property identifies/generates an event through the property.
- `always-evt` — a constant event that is always ready. The synchronization result of `always-evt` is `always-evt` itself.
- `never-evt` — a constant event that is never ready.
- `idle` — an event produced by `system-idle-evt` is ready when, if this event were replaced by `never-evt`, no thread in the system would be available to run. In other words, all threads must be suspended or blocked on events with timeouts that have not yet expired. The event's result is `#<void>`.

```
(evt? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a synchronizable event, `#f` otherwise.

```
(sync evt ...+) → any
  evt : evt?
```

Blocks as long as none of the synchronizable events *evt*s are ready, as defined above.

When at least one *evt* is ready, its synchronization result (often *evt* itself) is returned. If multiple *evt*s are ready, one of the *evt*s is chosen pseudo-randomly for the result; the `current-evt-pseudo-random-generator` parameter sets the random-number generator that controls this choice.

```
(sync/timeout timeout-secs evt ...+) → any
  timeout-secs : (or/c nonnegative-number? false/c)
  evt : evt?
```

Like `sync`, but returns `#f` if *timeout-secs* is not `#f` and if *timeout-secs* seconds pass without a successful synchronization.

If *timeout-secs* is `0`, each *evt* is checked at least once, so a *timeout-secs* value of `0` can be used for polling.

See also `alarm-evt` for an alternative timeout mechanism.

```
(sync/enable-break evt ...+) → any
  evt : evt?
```

Like `sync`, but breaking is enabled (see §9.6 "Breaks") while waiting on the *evt*s. If breaking is disabled when `sync/enable-break` is called, then either all *evt*s remain unchosen or the `exn:break` exception is raised, but not both.

```
(sync/timeout/enable-break timeout-secs
                           evt ...+)    → any
  timeout-secs : (or/c nonnegative-number? false/c)
  evt : evt?
```

Like `sync/enable-break`, but with a timeout in seconds (or `#f`), as for `sync/timeout`.

```
(choice-evt evt ...) → evt?
  evt : evt?
```

Creates and returns a single event that combines the *evt*s. Supplying the result to `sync` is the same as supplying each *evt* to the same call.

```
(wrap-evt evt wrap) → evt?
  evt : (and/c evt? (not/c handle-evt?))
  wrap : (any/c . -> . any)
```

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

```
(handle-evt evt handle) → evt?
  evt : (and/c evt? (not/c handle-evt?))
  handle : (any/c . -> . any)
```

Like `wrap`, except that *handle* is called in tail position with respect to the synchronization request, and without breaks explicitly disabled.

```
(guard-evt generator) → evt?
  generator : (-> evt?)
```

Creates a value that behaves as an event, but that is actually an event generator. For details, see the overview.

```
(nack-guard-evt generator) → evt?
  generator : (evt? . -> . evt?)
```

Creates a value that behaves as an event, but that is actually an event generator; the generator procedure receives an event that becomes ready with a `#<void>` value if the generated event was not ultimately chosen. For details, see the overview.

```
(poll-guard-evt generator) → evt?
  generator : (boolean? . -> . evt?)
```

Creates a value that behaves as an event, but that is actually an event generator; the generator procedure receives a boolean indicating whether the event is used for polling. For details, see the overview.

```
always-evt : evt?
```

A constant event that is always ready, with itself as its result.

```
never-evt : evt?
```

A constant event that is never ready.

```
(system-idle-evt) → evt?
```

Returns an event that is ready when the system is otherwise idle; see the overview for more information. The result of the `system-idle-evt` procedure is always the same event.

```
(alarm-evt msecs) → evt
  msecs : nonnegative-number?
```

Returns a synchronizable event that is not ready when `(current-inexact-milliseconds)` would return a value that is less than *msecs*, and it is ready when `(current-inexact-milliseconds)` would return a value that is more than *msecs*.

```
(handle-evt? evt) → boolean?
  evt : evt?
```

Returns `#t` if *evt* was created by `handle-evt` or by `choice-evt` applied to another event for which `handle-evt?` produces `#t`. Such events are illegal as an argument to `handle-evt` or `wrap-evt`, because they cannot be wrapped further. For any other event, `handle-evt?`

produces #f, and the event is a legal argument to handle-evt or wrap-evt for further wrapping.

```
prop:evt : struct-type-property?
```

A structure type property that identifies structure types whose instances can serve as synchronizable events. The property value can be any of the following:

- An event evt: In this case, using the structure as an event is equivalent to using evt.
- A procedure proc of one argument: In this case, the structure is similar to an event generated by guard-evt, except that the would-be guard procedure proc receives the structure as an argument, instead of no arguments.
- An exact, non-negative integer between 0 (inclusive) and the number of non-automatic fields in the structure type (exclusive, not counting supertype fields): The integer identifies a field in the structure, and the field must be designated as immutable. If the field contains an object or an event-generating procedure of one argument, the event or procedure is used as above. Otherwise, the structure acts as an event that is never ready.

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

Examples:

```
> (define-struct wt (base val)
                 #:property prop:evt (struct-field-index base))
> (define sema (make-semaphore))
> (sync/timeout 0 (make-wt sema #f))
#f
> (semaphore-post sema)
> (sync/timeout 0 (make-wt sema #f))
#<semaphore>
> (semaphore-post sema)
> (sync/timeout 0 (make-wt (lambda (self) (wt-val self)) sema))
#<semaphore>
> (semaphore-post sema)
> (define my-wt (make-wt (lambda (self) (wrap-evt
                                            (wt-val self)
                                            (lambda (x) self)))
                         sema))
> (sync/timeout 0 my-wt)
```

```
#<wt>
> (sync/timeout 0 my-wt)
#f
```

```
(current-evt-pseudo-random-generator)
 → pseudo-random-generator?
(current-evt-pseudo-random-generator generator) → void?
  generator : pseudo-random-generator?
```

A parameter that determines the pseudo-random number generator used by `sync` for events created by `choice-evt`.

10.2.2 Channels

A *channel* both synchronizes a pair of threads and passes a value from one to the other. Channels are synchronous; both the sender and the receiver must block until the (atomic) transaction is complete. Multiple senders and receivers can access a channel at once, but a single sender and receiver is selected for each transaction.

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

For buffered asynchronous channels, see §10.2.4 "Buffered Asynchronous Channels".

```
(make-channel) → channel?
```

Creates and returns a new channel. The channel can be used with `channel-get`, with `channel-try-get`, or as a synchronizable event (see §10.2.1 "Events") to receive a value through the channel. The channel can be used with `channel-put` or through the result of `channel-put-evt` to send a value through the channel.

```
(channel? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a channel created by `make-channel`, `#f` otherwise.

```
(channel-get ch) → any
  ch : channel?
```

Blocks until a sender is ready to provide a value through *ch*. The result is the sent value.

```
(channel-try-get ch) → any
  ch : channel?
```

Receives and returns a value from `ch` if a sender is immediately ready, otherwise returns `#f`.

```
(channel-put ch v) → void?
  ch : channel?
  v : any/c
```

Blocks until a receiver is ready to accept the value `v` through `ch`.

```
(channel-put-evt ch v) → evt?
  ch : channel?
  v : any/c
```

Returns a fresh synchronizable event for use with `sync`. The event is ready when `(channel-put ch v)` would not block, and the event's synchronization result is the event itself.

10.2.3 Semaphores

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

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

In addition to its use with semaphore-specific procedures, semaphores can be used as events; see §10.2.1 "Events".

```
(make-semaphore [init]) → semaphore?
  init : nonnegative-exact-integer? = 0
```

Creates and returns a new semaphore with the counter initially set to `init`. If `init-k` is larger than a semaphore's maximum internal counter value, the `exn:fail` exception is raised.

```
(semaphore? v) → boolean?
```

```
v : any/c
```

Returns `#t` if *v* is a semaphore created by `make-semaphore`, `#f` otherwise.

```
(semaphore-post sema) → void?
  sema : semaphore?
```

Increments the semaphore's internal counter and returns `#<void>`. If the semaphore's internal counter has already reached its maximum value, the `exn:fail` exception is raised.

```
(semaphore-wait sema) → void?
  sema : semaphore?
```

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) → boolean?
  sema : semaphore?
```

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) → void?
  sema : semaphore?
```

Like `semaphore-wait`, but breaking is enabled (see §9.6 "Breaks") while waiting on *sema*. If breaking is disabled when `semaphore-wait/enable-break` is called, then either the semaphore's counter is decremented or the `exn:break` exception is raised, but not both.

```
(semaphore-peek-evt sema) → evt?
  sema : semaphore?
```

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

```
(call-with-semaphore  sema
                      proc
                     [try-fail-thunk]
                      arg ...)        → any
```

```
sema : semaphore?
proc : procedure?
try-fail-thunk : (or/c (-> any) false/c) = #f
arg : any/c
```

Waits on *sema* using `semaphore-wait`, calls *proc* with all *arg*s, and then posts to *sema*. A continuation barrier blocks full continuation jumps into or out of *proc* (see §1.1.12 "Prompts, Delimited Continuations, and Barriers"), but escape jumps are allowed, and *sema* is posted on escape. If *try-fail-thunk* is provided and is not `#f`, then `semaphore-try-wait?` is called on *sema* instead of `semaphore-wait`, and *try-fail-thunk* is called if the wait fails.

```
(call-with-semaphore/enable-break  sema
                                   proc
                                  [try-fail-thunk]
                                   arg ...)         → any
  sema : semaphore?
  proc : procedure?
  try-fail-thunk : (or/c (-> any) false/c) = #f
  arg : any/c
```

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

10.2.4 Buffered Asynchronous Channels

```
(require scheme/async-channel)
```

The bindings documented in this section are provided by the `scheme/async-channel` library, not `scheme/base` or `scheme`.

See also §10.1.4 "Thread Mailboxes".

```
(async-channel? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is an asynchronous channel, `#f` otherwise.

```
(make-async-channel [limit]) → async-channel?
  limit : (or/c exact-positive-integer? false/c) = #f
```

Returns an asynchronous channel with a buffer limit of *limit* items. A get operation blocks when the channel is empty, and a put operation blocks when the channel has *limit* items already. If `limit` is `#f`, the channel buffer has no limit (so a put never blocks).

The asynchronous channel value can be used directly with `sync`. The channel blocks until `async-channel-get` would return a value, and the unblock result is the received value.

```
(async-channel-get ach) → any/c
  ach : async-channel?
```

Blocks until at least one value is available in `ach`, and then returns the first of the values that were put into `async-channel`.

```
(async-channel-try-get ach) → any/c
  ach : async-channel?
```

If at least one value is immediately available in `ach`, returns the first of the values that were put into `ach`. If `async-channel` is empty, the result is `#f`.

```
(async-channel-put ach v) → void?
  ach : async-channel?
  v : any/c
```

Puts `v` into `ach`, blocking if `ach`'s buffer is full until space is available.

```
(async-channel-put-evt async-channel v) → evt?
  async-channel : channel?
  v : any/c
```

Returns a synchronizable event that is blocked while `(async-channel-put ach v)` would block. The unblock result is the event itself. See also `sync`.

10.3 Thread-Local Storage

Thread cells provides primitive support for thread-local storage. Parameters combine thread cells and continuation marks to support thread-specific, continuation-specific binding.

10.3.1 Thread Cells

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

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

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

```
(make-thread-cell v [preserved?]) → thread-cell?
  v : any/c
  preserved? : any/c = #f
```

Creates and returns a new thread cell. Initially, *v* is the cell's value for all threads. If *preserved?* is true, then the cell's initial value for a newly created threads is the creating thread's value for the cell, otherwise the cell's value is initially *v* in all future threads.

```
(thread-cell? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a thread cell created by `make-thread-cell`, `#f` otherwise.

```
(thread-cell-ref cell) → any
  cell : thread-cell?
```

Returns the current value of *cell* for the current thread.

```
(thread-cell-set! cell v) → any
  cell : thread-cell?
  v : any/c
```

Sets the value in *cell* to *v* for the current thread.

Examples:

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

```
(cancer)
> (thread-cell-set! cnp '(nerve nerve))
> (thread-cell-set! cp '(cancer cancer))
> (thread-cell-ref cnp)
(nerve nerve)
> (thread-cell-ref cp)
(cancer cancer)
> (define ch (make-channel))
> (thread (lambda ()
            (channel-put ch (thread-cell-ref cnp))
            (channel-put ch (thread-cell-ref cp))
            (channel-get ch)
            (channel-put ch (thread-cell-ref cp))))
#<thread>
> (channel-get ch)
(nerve)
> (channel-get ch)
(cancer cancer)
> (thread-cell-set! cp '(cancer cancer cancer))
> (thread-cell-ref cp)
(cancer cancer cancer)
> (channel-put ch 'ok)
> (channel-get ch)
(cancer cancer)
```

```
(current-preserved-thread-cell-values) → any
(current-preserved-thread-cell-values thread-cell-vals) → void?
  thread-cell-vals : any/c
```

When called with no arguments, this procedure produces a *thread-cell-vals* that represents the current values (in the current thread) for all preserved thread cells.

When called with a *thread-cell-vals* generated by a previous call to `current-preserved-thread-cell-values`, the values of all preserved thread cells (in the current thread) are set to the values captured in *thread-cell-vals*; if a preserved thread cell was created after *thread-cell-vals* was generated, then the thread cell's value for the current thread reverts to its initial value.

10.3.2 Parameters

See §1.1.14 "Parameters" for basic information on the parameter model. Parameters correspond to *preserved thread fluids* in Scsh [Gasbichler02].

To parameterize code in a thread- and continuation-friendly manner, use `parameterize`.

The `parameterize` form introduces a fresh thread cell for the dynamic extent of its body expressions.

When a new thread is created, the parameterization for the new thread's initial continuation is the parameterization of the creator thread. Since each parameter's thread cell is preserved, the new thread "inherits" the parameter values of its creating thread. When a continuation is moved from one thread to another, settings introduced with `parameterize` effectively move with the continuation.

In contrast, direct assignment to a parameter (by calling the parameter procedure with a value) changes the value in a thread cell, and therefore changes the setting only for the current thread. Consequently, as far as the memory manager is concerned, the value originally associated with a parameter through `parameterize` remains reachable as long the continuation is reachable, even if the parameter is mutated.

```
(make-parameter v [guard]) → parameter?
  v : any/c
  guard : (or/c (any/c . -> . any) false/c) = #f
```

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

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

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

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

If a continuation is captured during the evaluation of `parameterize`, invoking the continuation effectively re-introduces the parameterization, since a parameterization is associated to a continuation via a continuation mark (see §9.5 "Continuation Marks") using a private key.

Examples:

```
> (parameterize ([exit-handler (lambda (x) 'no-exit)])
    (exit))
> (define p1 (make-parameter 1))
> (define p2 (make-parameter 2))
> (parameterize ([p1 3]
                 [p2 (p1)])
    (cons (p1) (p2)))
(3 . 1)
> (let ([k (let/cc out
             (parameterize ([p1 2])
               (p1 3)
               (cons (let/cc k
                       (out k))
                     (p1))))])
    (if (procedure? k)
        (k (p1))
        k))
(1 . 3)
> (define ch (make-channel))
> (parameterize ([p1 0])
    (thread (lambda ()
              (channel-put ch (cons (p1) (p2))))))
#<thread>
> (channel-get ch)
(0 . 2)
> (define k-ch (make-channel))
> (define (send-k)
    (parameterize ([p1 0])
      (thread (lambda ()
                (let/ec esc
                  (channel-put ch
                               ((let/cc k
                                  (channel-put k-ch k)
                                  (esc)))))))))
> (send-k)
#<thread>
> (thread (lambda () ((channel-get k-ch)
                      (let ([v (p1)])
                        (lambda () v)))))
#<thread>
> (channel-get ch)
1
> (send-k)
#<thread>
> (thread (lambda () ((channel-get k-ch) p1)))
```

```
#<thread>
> (channel-get ch)
0
```

```
(parameterize* ((parameter-expr value-expr) ...)
  body ...+)
```

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

```
(make-derived-parameter v guard wrap) → parameter?
  v : any/c
  guard : (any/c . -> . any)
  wrap : (any/c . -> . any)
```

Returns a parameter procedure that sets or retrieves the same value as `parameter`, but with:

- *`guard`* applied when setting the parameter (before any guard associated with `parameter`), and
- *`wrap`* applied when obtaining the parameter's value.

```
(parameter? v) → boolean?
  v : any/c
```

Returns `#t` if *`v`* is a parameter procedure, `#f` otherwise.

```
(parameter-procedure=? a b) → boolean?
  a : parameter?
  b : parameter?
```

Returns `#t` if the parameter procedures *`a`* and *`b`* always modify the same parameter with the same guards, `#f` otherwise.

```
(current-parameterization) → parameterization?
```

Returns the current continuation's parameterization.

```
(call-with-parameterization parameterization
                            thunk)             → any
```

```
parameterization : parameterization?
thunk : (-> any)
```

Calls *thunk* (via a tail call) with *parameterization* as the current parameterization.

```
(parameterization? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a parameterization returned by `current-parameterization`, `#f` otherwise.

11 Macros

See §1.2 "Syntax Model" for general information on how programs are parsed. In particular, the subsection §1.2.3.2 "Expansion Steps" describes how parsing triggers macros, and §1.2.3.5 "Transformer Bindings" describes how macro transformers are called.

11.1 Pattern-Based Syntax Matching

```
(syntax-case stx-expr (literal-id ...)
  clause ...)

      clause = [pattern result-expr]
             | [pattern fender-expr result-expr]

     pattern = _
             | id
             | (pattern ...)
             | (pattern ...+ . pattern)
             | (pattern ... pattern ellipses pattern ...)
             | (pattern ... pattern ellipses pattern ... . pattern)
             | #(pattern ...)
             | #(pattern ... pattern ellipses pattern ...)
             | #s(key-datum pattern ...)
             | #s(key-datum pattern ... pattern ellipses pattern ...)
             | (ellipses stat-pattern)
             | const

stat-pattern = id
             | (stat-pattern ...)
             | (stat-pattern ...+ . stat-pattern)
             | #(stat-pattern ...)
             | const

    ellipses = ...
```

Finds the first `pattern` that matches the syntax object produced by `stx-expr`, and for which the corresponding `fender-expr` (if any) produces a true value; the result is from the corresponding `result-expr`, which is in tail position for the `syntax-case` form. If no `clause` matches, then the `exn:fail:syntax` exception is raised.

A syntax object matches a `pattern` as follows:

`_`

A `_` pattern (i.e., an identifier with the same binding as `_`) matches any syntax object.

`id`

An `id` matches any syntax object when it is not bound to `...` or `_` and does not have the same binding as any `literal-id`. The `id` is further bound as *pattern variable* for the corresponding `fender-expr` (if any) and `result-expr`. A pattern-variable binding is a transformer binding; the pattern variable can be reference only through forms like `syntax`. The binding's value is the syntax object that matched the pattern with a *depth marker* of 0.

An `id` that has the same binding as a `literal-id` matches a syntax object that is an identifier with the same binding in the sense of `free-identifier=?`. The match does not introduce any pattern variables.

`(pattern ...)`

A `(pattern ...)` pattern matches a syntax object whose datum form (i.e., without lexical information) is a list with as many elements as sub-`pattern`s in the pattern, and where each syntax object that corresponding to an element of the list matches the corresponding sub-`pattern`.

Any pattern variables bound by the sub-`pattern`s are bound by the complete pattern; the bindings must all be distinct.

`(pattern ...+ . pattern)`

The last `pattern` must not be a `(pattern ...)`, `(pattern ...+ . pattern)`, `(pattern ... pattern ellipses pattern ...)`, or `(pattern ... pattern ellipses pattern ... . pattern)` form.

Like the previous kind of pattern, but matches syntax objects that are not necessarily lists; for *n* sub-`pattern`s before the last sub-`pattern`, the syntax object's datum must be a pair such that *n*-1 `cdr`s produce pairs. The last sub-`pattern` is matched against the syntax object corresponding to the *n*th `cdr` (or the `datum->syntax` coercion of the datum using the nearest enclosing syntax object's lexical context and source location).

`(pattern ... pattern ellipses pattern ...)`

Like the `(pattern ...)` kind of pattern, but matching a syntax object with any number (zero or more) elements that match the sub-`pattern` followed by `ellipses` in the corresponding position relative to other sub-`pattern`s.

For each pattern variable bound by the sub-`pattern` followed by `ellipses`, the larger pattern binds the same pattern variable to a list of values, one for each element of the syntax object matched to the sub-`pattern`, with an incremented depth marker. (The sub-`pattern` itself may contain `ellipses`, leading to a pattern variables bound to lists of lists of syntax objects with a depth marker of 2, and so on.)

`(pattern ... pattern ellipses pattern ... . pattern)`

Like the previous kind of pattern, but with a final sub-`pattern` as for `(pattern ...+ . pattern)`. The final `pattern` never matches a syntax object whose datum is a pair.

`#(pattern ...)`

Like a `(pattern ...)` pattern, but matching a vector syntax object whose elements match the corresponding sub-`pattern`s.

`#(pattern ... pattern ellipses pattern ...)`

Like a `(pattern ... pattern ellipses pattern ...)` pattern, but matching a vector syntax object whose elements match the corresponding sub-`pattern`s.

`#s(key-datum pattern ...)`

Like a `(pattern ...)` pattern, but matching a prefab structure syntax object whose fields match the corresponding sub-`pattern`s. The `key-datum` must correspond to a valid first argument to `make-prefab-struct`.

`#s(key-datum pattern ... pattern ellipses pattern ...)`

Like a `(pattern ... pattern ellipses pattern ...)` pattern, but matching a prefab structure syntax object whose elements match the corresponding sub-`pattern`s.

```
(ellipses stat-pattern)
```

Matches the same as *stat-pattern*, which is like a *pattern*, but identifiers with the binding `...` are treated the same as other *id*s.

```
const
```

A *const* is any datum that does not match one of the preceeding forms; a syntax object matches a *const* pattern when its datum is `equal?` to the `quote`d *const*.

```
(syntax-case* stx-expr (literal-id ...) id-compare-expr
  clause ...)
```

Like `syntax-case`, but *id-compare-expr* must produce a procedure that accepts two arguments. A *literal-id* in a *pattern* matches an identifier for which the procedure returns true when given the identifier to match (as the first argument) and the identifier in the *pattern* (as the second argument).

In other words, `syntax-case` is like `syntax-case*` with an *id-compare-expr* that produces `free-identifier=?`.

```
(with-syntax ([pattern stx-expr] ...)
  body ...+)
```

Similar to `syntax-case`, in that it matches a *pattern* to a syntax object. Unlike `syntax-case`, all *pattern*s are matched, each to the result of a corresponding *stx-expr*, and the pattern variables from all matches (which must be distinct) are bound with a single *body* sequence. The result of the `with-syntax` form is the result of the last *body*, which is in tail position with respect to the `with-syntax` form.

If any *pattern* fails to match the corresponding *stx-expr*, the `exn:fail:syntax` exception is raised.

A `with-syntax` form is roughly equivalent to the following `syntax-case` form:

```
(syntax-case (list stx-expr ...) ()
  [(pattern ...) (let () body ...+)])
```

However, if any individual *stx-expr* produces a non-syntax object, then it is converted to one using `datum->syntax` and the lexical context and source location of the individual *stx-expr*.

```
(syntax template)

     template = id
              | (template-elem ...)
              | (template-elem ...+ . template)
              | #(template-elem ...)
              | #s(key-datum template-elem ...)
              | (ellipses stat-template)
              | const

template-elem = template ellipses ...

stat-template = id
              | (stat-template ...)
              | (stat-template ... . stat-template)
              | #(stat-template ...)
              | #s(key-datum stat-template ...)
              | const

     ellipses = ...
```

Constructs a syntax object based on a `template`,which can inlude pattern variables bound by `syntax-case` or `with-syntax`.

Template forms produce a syntax object as follows:

`id`

If `id` is bound as a pattern variable, then `id` as a template produces the pattern variable's match result. Unless the `id` is a sub-`template` that is replicated by `ellipses` in a larger `template`, the pattern variable's value must be a syntax object with a depth marker of 0 (as opposed to a list of matches).

More generally, if the pattern variable's value has a depth marker *n*, then it can only appear within a template where it is replicated by at least *n* `ellipses`es. In that case, the template will be replicated enough times to use each match result at least once.

If `id` is not bound as a pattern variable, then `id` as a template produces `(quote-syntax id)`.

`(template-elem ...)`

Produces a syntax object whose datum is a list, and where the elements of the list correspond to syntax objects producesd by the `template-elem`s.

A *template-elem* is a sub-*template* replicated by any number of *ellipses*es:

- If the sub-*template* is replicated by no *ellipses*es, then it generates a single syntax object to incorporate into the result syntax object.
- If the sub-*template* is replicated by one *ellipses*, then it generates a sequence of syntax objects that is "inlined" into the resulting syntax object.

  The number of generated elements depends the values of pattern variables referenced within the sub-*template*. There must be at least one pattern variable whose value is has a depth marker less than the number of *ellipses*es after the pattern variable within the sub-*template*.

  If a pattern variable is replicated by more *ellipses*es in a *template* than the depth marker of its binding, then the pattern variable's result is determined normally for inner *ellipses*es (up to the binding's depth marker), and then the result is replicated as necessary to satisfy outer *ellipses*es.
- For each *ellipses* after the first one, the preceding element (with earlier replicating *ellipses*s) is conceptually wrapped with parentheses for generating output, and then the wrapping parentheses are removed in the resulting syntax object.

`(template-elem ... . template)`

Like the previous form, but the result is not necessarily a list; instead, the place of the empty list in resulting syntax object's datum is taken by the syntax object produced by *template*.

`#(template-elem ...)`

Like the `(template-elem ...)` form, but producing a syntax object whose datum is a vector instead of a list.

`#s(key-datum template-elem ...)`

Like the `(template-elem ...)` form, but producing a syntax object whose datum is a prefab structure instead of a list. The *key-datum* must correspond to a valid first argument of `make-prefab-struct`.

`(ellipses stat-template)`

Produces the same result as *stat-template*, which is like a *template*, but `...` is treated like a *id* (with no pattern binding).

`const`

A `const` template is any form that does not match the preceding cases, and it produces the result `(quote-syntac const)`.

A `(syntax template)` form is normally abbreviated as `#'template`; see also §12.6.7 "Reading Quotes". If `template` contains no pattern variables, then `#'template` is equivalent to `(quote-syntax template)`.

```
(quasisyntax template)
```

Like `syntax`, but `(unsyntax expr)` and `(unsyntax-splicing expr)` escape to an expression within the `template`.

The `expr` must produce a syntax object (or syntax list) to be substituted in place of the `unsyntax` or `unsyntax-splicing` form within the quasiquoting template, just like `unquote` and `unquote-splicing` within `quasiquote`. (If the escaped expression does not generate a syntax object, it is converted to one in the same was as for the right-hand sides of `with-syntax`.) Nested `quasisyntax`es introduce quasiquoting layers in the same way as nested `quasiquote`s.

Also analogous `quasiquote`, the reader converts `` #` `` to `quasisyntax`, `#,` to `unsyntax`, and `#,@` to `unsyntax-splicing`. See also §12.6.7 "Reading Quotes".

```
(unsyntax expr)
```

Illegal as an expression form. The `unsyntax` form is for use only with a `quasisyntax` template.

```
(unsyntax-splicing expr)
```

Illegal as an expression form. The `unsyntax-splicing` form is for use only with a `quasisyntax` template.

```
(syntax/loc stx-expr template)
```

Like `syntax`, except that the immediate resulting syntax object takes its source-location information from the result of `stx-expr` (which must produce a syntax object), unless the `template` is just a pattern variable.

```
(quasisyntax/loc stx-expr template)
```

Like `quasisyntax`, but with source-location assignment like `syntax/loc`.

```
(syntax-rules (literal-id ...)
  [(id . pattern) template] ...)
```

Equivalent to

```
(lambda (stx)
  (syntax-case stx (literal-id ...)
    [(generated-id . pattern) #'template]  ...))
```

where each *`generated-id`* binds no identifier in the corresponding *`template`*.

```
(syntax-id-rules (literal-id ...)
  [pattern template] ...)
```

Equivalent to

```
(lambda (stx)
  (make-set!-transformer
   (syntax-case stx (literal-id ...)
     [pattern #'template]  ...)))
```

```
(define-syntax-rule (id . pattern) template)
```

Equivalent to

```
(define-syntax id
  (syntax-rules ()
   [(id . pattern) template]))
```

`...`

The `...` transformer binding prohibits `...` from being used as an expression. This binding useful only in syntax patterns and templates, where it indicates repetitions of a pattern or template. See `syntax-case` and `syntax`.

`_`

The `_` transformer binding prohibits `_` from being used as an expression. This binding useful only in syntax patterns, where it indicates a pattern that matches any syntax object. See `syntax-case`.

11.2 Syntax Object Content

```
(syntax? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a syntax object, `#f` otherwise. See also §1.2.2 "Syntax Objects".

```
(syntax-source stx) → any
  stx : syntax?
```

Returns the source for the syntax object *stx*, or `#f` if none is known. The source is represented by an arbitrary value (e.g., one passed to `read-syntax`), but it is typically a file path string. Source-location information is dropped for a syntax object that is marshaled as part of compiled code; see also `current-compile`.

```
(syntax-line stx) → (or/c positive-exact-integer?
                          false/c)
  stx : syntax?
```

Returns the line number (positive exact integer) for the start of the syntax object in its source, or `#f` if the line number or source is unknown. The result is `#f` if and only if `(syntax-column stx)` produces `#f`. See also §12.1.4 "Counting Positions, Lines, and Columns", and see `syntax-source` for information about marshaling compiled syntax objects.

```
(syntax-column stx) → (or/c exact-nonnegative-integer?
                            false/c)
  stx : syntax?
```

Returns the column number (non-negative exact integer) for the start of the syntax object in its source, or `#f` if the source column is unknown. The result is `#f` if and only if `(syntax-line stx)` produces `#f`. See also §12.1.4 "Counting Positions, Lines, and Columns", and see `syntax-source` for information about marshaling compiled syntax objects.

```
(syntax-position stx) → (or/c positive-exact-integer?
                              false/c)
  stx : syntax?
```

Returns the character position (positive exact integer) for the start of the syntax object in its source, or `#f` if the source position is unknown. See also §12.1.4 "Counting Positions, Lines, and Columns", and see `syntax-source` for information about marshaling compiled syntax objects.

```
(syntax-span stx) → (or/c exact-nonnegative-integer?
                          false/c)
  stx : syntax?
```

Returns the span (non-negative exact integer) in characters of the syntax object in its source, or `#f` if the span is unknown. See also `syntax-source` for information about marshaling compiled syntax objects.

```
(syntax-original? stx) → boolean?
  stx : syntax?
```

Returns `#t` if *stx* has the property that `read-syntax` and `read-honu-syntax` attach to the syntax objects that they generate (see §11.6 "Syntax Object Properties"), and if *stx*'s lexical information does not indicate that the object was introduced by a syntax transformer (see §1.2.2 "Syntax Objects"). The result is `#f` otherwise. This predicate can be used to distinguish syntax objects in an expanded expression that were directly present in the original expression, as opposed to syntax objects inserted by macros.

```
(syntax-source-module stx) → (or/c module-path-index? symbol?)
  stx : syntax?
```

Returns a module path index or symbol (see §13.4.2 "Compiled Modules and References") for the module whose source contains *stx*, or `#f` if *stx* has no source module.

```
(syntax-e stx) → any
  stx : syntax?
```

Unwraps the immediate datum structure from a syntax object, leaving nested syntax structure (if any) in place. The result of `(syntax-e (scheme stx))` is one of the following:

- a symbol
- a syntax pair (described below)
- the empty list
- an immutable vector containing syntax objects
- an immutable box containing syntax objects
- some other kind of datum—usually a number, boolean, or string

A *syntax pair* is a pair containing a syntax object as its first element, and either the empty list, a syntax pair, or a syntax object as its second element.

A syntax object that is the result of `read-syntax` reflects the use of delimited `.` in the input by creating a syntax object for every pair of parentheses in the source, and by creating a pair-valued syntax object *only* for parentheses in the source. See §12.6.5 "Reading Pairs and Lists" for more information.

```
(syntax->list stx) → (or/c list? false/c)
  stx : syntax?
```

Returns a list of syntax objects or `#f`. The result is a list of syntax objects when `(syntax->datum stx)` would produce a list. In other words, syntax pairs in `(syntax-e stx)` are flattened.

```
(syntax->datum stx) → any
  stx : syntax?
```

Returns a datum by stripping the lexical information, source-location information, properties, and certificates from *stx*. Inside of pairs, (immutable) vectors, and (immutable) boxes, syntax objects are recursively stripped.

The stripping operation does not mutate *stx*; it creates new pairs, vectors, and boxes as needed to strip lexical and source-location information recursively.

```
(datum->syntax ctxt v srcloc [prop cert]) → syntax?
  ctxt : (or/c syntax? false/c)
  v : any/c
  srcloc : (or/c syntax? false/c
                 (list/c any/c
                         (or/c exact-positive-integer? false/c)
                         (or/c exact-nonnegative-integer? false/c)
                         (or/c exact-nonnegative-integer? false/c)
                         (or/c exact-positive-integer? false/c))
                 (vector/c any/c
                          (or/c exact-positive-integer? false/c)
                          (or/c exact-nonnegative-integer? false/c)
                          (or/c exact-nonnegative-integer? false/c)
                          (or/c exact-positive-integer? false/c)))
  prop : (or/c syntax? false/c) = #f
  cert : (or/c syntax? false/c) = #f
```

Converts the datum *v* to a syntax object. If *v* is a pair, vector, or box, then the contents are recursively converted; mutable vectors and boxes are essentially replaced by immutable vectors and boxes. Syntax objects already in *v* are preserved as-is in the result. For any kind of value other than a pair, vector, box, or syntax object, conversion means wrapping the value with lexical information, source-location information, properties, and certificates.

Converted objects in *v* are given the lexical context information of *ctxt* and the source-location information of *srcloc*. If *v* is not already a syntax object, then the resulting immediate syntax object is given the properties (see §11.6 “Syntax Object Properties”) of *prop* and the inactive certificates (see §11.7 “Syntax Certificates”) of *cert*; if *v* is a pair, vector, or box, recursively converted values are not given properties or certificates.

Any of *ctxt*, *srcloc*, *prop*, or *cert* can be `#f`, in which case the resulting syntax has no lexical context, source information, new properties, and/or certificates.

If *srcloc* is not `#f` or a syntax object, it must be a list or vector of five elements:

```
(list source-name line column position span)
```
or
```
(vector source-name line column position span)
```

where `source-name-v` is an arbitrary value for the source name; `line` is an integer for the source line, or `#f`; `column` is an integer for the source column, or `#f`; `position` is an integer for the source position, or `#f`; and `span` is an integer for the source span, or `#f`. The `line` and `column` values must both be numbers or both be `#f`, otherwise the `exn:fail:contract` exception is raised.

Graph structure is not preserved by the conversion of *v* to a syntax object. Instead, *v* is essentially unfolded into a tree. If *v* has a cycle through pairs, vectors, and boxes, then the `exn:fail:contract` exception is raised.

```
(identifier? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a syntax object and `(syntax-e stx)` produces a symbol.

```
(generate-temporaries stx-pair) → (listof identifier?)
  stx-pair : (or syntax? list?)
```

Returns a list of identifiers that are distinct from all other identifiers. The list contains as many identifiers as *stx-pair* contains elements. The *stx-pair* argument must be a syntax pair that can be flattened into a list. The elements of *stx-pair* can be anything, but string, symbol, keyword (possibly wrapped as syntax), and identifier elements will be embedded in the corresponding generated name, which is useful for debugging purposes. The generated identifiers are built with interned symbols (not `gensym`s), so the limitations described with `current-compile` do not apply.

11.3 Syntax Object Bindings

```
(bound-identifier=? a-id b-id [phase-level]) → boolean?
```

```
  a-id : syntax?
  b-id : syntax?
  phase-level : (or/c exact-integer? false/c)
              = (syntax-local-phase-level)
```

Returns `#t` if the identifier `a-id` would bind `b-id` (or vice-versa) if the identifiers were substituted in a suitable expression context at the phase level indicated by `phase-level`, `#f` otherwise. A `#f` value for `phase-level` corresponds to the label phase level.

```
(free-identifier=? a-id b-id [phase-level]) → boolean?
  a-id : syntax?
  b-id : syntax?
  phase-level : (or/c exact-integer? false/c)
              = (syntax-local-phase-level)
```

Returns `#t` if `a-id` and `b-id` access the same lexical, module, or top-level binding at the phase level indicated by `phase-level`. A `#f` value for `phase-level` corresponds to the label phase level.

"Same module binding" means that the identifiers refer to the same original definition site, not necessarily the `require` or `provide` site. Due to renaming in `require` and `provide`, the identifiers may return distinct results with `syntax-e`.

```
(free-transformer-identifier=? a-id b-id) → boolean?
  a-id : syntax?
  b-id : syntax?
```

Same as `(free-identifier=? a-id b-id (add1 (syntax-local-phase-level)))`.

```
(free-template-identifier=? a-id b-id) → boolean?
  a-id : syntax?
  b-id : syntax?
```

Same as `(free-identifier=? a-id b-id (sub1 (syntax-local-phase-level)))`.

```
(free-label-identifier=? a-id b-id) → boolean?
  a-id : syntax?
  b-id : syntax?
```

Same as `(free-identifier=? a-id b-id #f)`.

```
(check-duplicate-identifier ids) → (or/c identifier? false/c)
  ids : (listof identifier?)
```

Compares each identifier in *ids* with every other identifier in the list with `bound-identifier=?`. If any comparison returns `#t`, one of the duplicate identifiers is returned (the first one in *ids* that is a duplicate), otherwise the result is `#f`.

```
(identifier-binding id-stx [phase-level])
 → (or/c (one-of 'lexical #f)
         (listof module-path-index?
                 symbol?
                 module-path-index?
                 symbol?
                 (one-of/c 0 1)
                 (or/c exact-integer? false/c)
                 (or/c exact-integer? false/c)))
  id-stx : syntax?
  phase-level : (or/c exact-integer? false/c)
              = (syntax-local-phase-level)
```

Returns one of three kinds of values, depending on the binding of *id-stx* at the phase level indicated by *phase-level* (where a `#f` value for *phase-level* corresponds to the label phase level):

- The result is `'lexical` if *id-stx* has a local binding. If `'lexical` is produced for any *phase-level* value, then it is produced for all *phase-level* values.
- The result is a list of six items when *id-stx* has a module binding: `(list source-mod source-id nominal-source-mod nominal-source-id source-phase import-phase nominal-export-phase)`.
  - *source-mod* is a module path index (see §13.4.2 “Compiled Modules and References”) that indicates the defining module.
  - *source-id* is a symbol for the identifier’s name at its definition site in the source module. This can be different from the local name returned by `syntax->datum` for several reasons: the identifier is renamed on import, it is renamed on export, or it is implicitly renamed because the identifier (or its import) was generated by a macro invocation.
  - *nominal-source-mod* is a module path index (see §13.4.2 “Compiled Modules and References”) that indicates the module `required` into the context of *id-stx* to provide its binding. It can be different from *source-mod* due to a re-export in *nominal-source-mod* of some imported identifier. If the same binding is imported in multiple ways, an arbitrary representative is chosen.

- *nominal-source-id* is a symbol for the identifier's name as exported by *nominal-source-mod*. It can be different from *source-id* due to a renaming provide, even if *source-mod* and *nominal-source-mod* are the same.
- *source-phase* is 1 if the source definition is for-syntax, 0 otherwise.
- *import-phase* is 0 if the binding import of *nominal-source-mode* is a plain require, 1 if it is from a for-syntax import, etc.
- *nominal-export-phase* is the phase level of the export from *nominal-source-mod*.

- The result is #f if *id-stx* has a top-level binding.

```
(identifier-transformer-binding id-stx)
 → (or/c (one-of 'lexical #f)
         (listof module-path-index?
                 symbol?
                 module-path-index?
                 symbol?
                 (one-of/c 0 1)
                 (or/c exact-integer? false/c)
                 (or/c exact-integer? false/c)))
  id-stx : syntax?
```

Same as (identifier-binding *id-stx* (add1 (syntax-local-phase-level))).

```
(identifier-template-binding id-stx)
 → (or/c (one-of 'lexical #f)
         (listof module-path-index?
                 symbol?
                 module-path-index?
                 symbol?
                 (one-of/c 0 1)
                 (or/c exact-integer? false/c)
                 (or/c exact-integer? false/c)))
  id-stx : syntax?
```

Same as (identifier-binding *id-stx* (sub1 (syntax-local-phase-level))).

```
(identifier-label-binding id-stx)
```

```
→ (or/c (one-of 'lexical #f)
        (listof module-path-index?
                symbol?
                module-path-index?
                symbol?
                (one-of/c 0 1)
                (or/c exact-integer? false/c)
                (or/c exact-integer? false/c)))
 id-stx : syntax?
```

Same as `(identifier-binding id-stx #f)`.

11.4 Syntax Transformers

```
(make-set!-transformer proc) → set!-transformer?
  proc : (syntax? . -> . syntax?)
```

Creates a syntax transformer that cooperates with `set!`. If the result of `make-set!-transformer` is bound to `identifier` as a transformer binding, then *proc* is applied as a transformer when `identifier` is used in an expression position, or when it is used as the target of a `set!` assignment as `(set! identifier expr)`. When the identifier appears as a `set!` target, the entire `set!` expression is provided to the transformer.

Examples:

```
> (let ([x 1]
        [y 2])
    (let-syntax ([x (make-set!-transformer
                      (lambda (stx)
                        (syntax-case stx (set!)
                          ; Redirect mutation of x to y
                          [(set! id v) #'(set! y v)]
                          ; Normal use of x really gets x
                          [id (identifier? #'id)  #'x])))])
      (begin
        (set! x 3)
        (list x y))))
(1 3)
```

```
(set!-transformer? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a value created by `make-set!-transformer`, `#f` otherwise.

```
(set!-transformer-procedure transformer)
 → (syntax? . -> . syntax?)
  transformer : set!-transformer?
```

Returns the procedure that was passed to `make-set!-transformer` to create *transformer*.

```
(make-rename-transformer id-stx) → rename-transformer?
  id-stx : syntax?
```

Creates a value that, when used as a transformer binding, inserts the identifier *id-stx* in place of whatever identifier binds the transformer, including in non-application positions, and in `set!` expressions. Such a transformer could be written manually, but the one created by `make-rename-transformer` cooperates specially with `syntax-local-value` (see below).

```
(rename-transformer? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a value created by `make-rename-transformer`, `#f` otherwise.

```
(rename-transformer-target transformer) → syntax?
  transformer : rename-transformer?
```

Returns the identifier passed to `make-rename-transformer` to create *transformer*.

```
(local-expand  stx
               context-v
               stop-ids
              [intdef-ctx]) → syntax?
  stx : syntax?
  context-v : (or/c (one-of 'expression 'top-level 'module
                            'module-begin)
                    list?)
  stop-ids : (or/c (listof identifier?) false/c)
  intdef-ctx : (or/c internal-definition-context? = #f
                     false/c)
```

Expands *stx* in the lexical context of the expression currently being expanded. The *context-v* argument is used as the result of `syntax-local-context` for immediate expansions; for a particular internal-definition context, generate a unique value and `cons` it onto the current result of `syntax-local-context` if it is a list.

When an identifier in *stop-ids* is encountered by the expander in a subexpression, expansions stops for the subexpression. If `#%app`, `#%top`, or `#%datum` appears in *stop-ids*, then application, top-level variable reference, and literal data expressions without the respective explicit form are not wrapped with the explicit form. If *stop-ids* is `#f` instead of a list, then *stx* is expanded only as long as the outermost form of *stx* is a macro (i.e., expansion does not proceed to sub-expressions).

The optional *intdef-ctx* argument must be either `#f` or the result of `syntax-local-make-definition-context`. In the latter case, lexical information for internal definitions is added to *stx* before it is expanded. The lexical information is also added to the expansion result (because the expansion might introduce bindings or references to internal-definition bindings).

Expansion of *stx* can use certificates for the expression already being expanded (see §11.7 “Syntax Certificates”) , and inactive certificates associated with *stx* are activated for *stx* (see §11.7 “Syntax Certificates”). Furthermore, if the transformer is defined within a module (i.e., the current expansion was triggered by a use of a module-defined identifier with a transformer binding) or if the current expression is being expanded for the body of a module, then the expansion of *stx* can use any identifier defined by the module.

This procedure must be called during the dynamic extent of a syntax transformer application by the expander, otherwise the `exn:fail:contract` exception is raised.

```
(syntax-local-expand-expression stx) → syntax? syntax?
  stx : syntax?
```

Like `local-expand` given `'expression` and an empty stop list, but with two results: a syntax object for the fully expanded expression, and a syntax object whose content is opaque. The latter can be used in place of the former (perhaps in a larger expression produced by a macro transformer), and when the macro expander encounters the opaque object, it substitutes the fully expanded expression without re-expanding it; the `exn:fail:syntax` exception is raised if the expansion context includes bindings or marks that were not present for the original expansion, in which case re-expansion might produce different results. Consistent use of `syntax-local-expand-expression` and the opaque object thus avoids quadratic expansion times when local expansions are nested.

This procedure must be called during the dynamic extent of a syntax transformer application by the expander, otherwise the `exn:fail:contract` exception is raised.

```
(local-transformer-expand stx
                          context-v
                          stop-ids
                         [intdef-ctx]) → syntax?
  stx : syntax?
```

```
context-v : (or/c (one-of 'expression 'top-level 'module
                          'module-begin)
                  list?)
stop-ids : (or/c (listof identifier?) false/c)
intdef-ctx : (or/c internal-definition-context? = #f
                   false/c)
```

Like `local-expand`, but *stx* is expanded as a transformer expression instead of a run-time expression.

```
(local-expand/capture-lifts  stx
                             context-v
                             stop-ids
                            [intdef-ctx
                             lift-ctx])  → syntax?
  stx : syntax?
  context-v : (or/c (one-of 'expression 'top-level 'module
                            'module-begin)
                    list?)
  stop-ids : (or/c (listof identifier?) false/c)
  intdef-ctx : (or/c internal-definition-context? = #f
                     false/c)
  lift-ctx : any/c = (gensym 'lifts)
```

Like `local-expand`, the result is a syntax object that represents a `begin` expression. Lifted expressions—from calls to `syntax-local-lift-expression` during the expansion of *stx*—appear with their identifiers in `define-values` forms, and the expansion of *stx* is the last expression in the `begin`. The *lift-ctx* value is reported by `syntax-local-lift-context` during local expansion. The lifted expressions are not expanded, but instead left as provided in the `begin` form.

```
(local-transformer-expand/capture-lifts  stx
                                         context-v
                                         stop-ids
                                        [intdef-ctx])  → syntax?
  stx : syntax?
  context-v : (or/c (one-of 'expression 'top-level 'module
                            'module-begin)
                    list?)
  stop-ids : (or/c (listof identifier?) false/c)
  intdef-ctx : (or/c internal-definition-context? = #f
                     false/c)
```

Like `local-expand/capture-lifts`, but *stx* is expanded as a transformer expression instead of a run-time expression. Lifted expressions are reported as `define-values` forms (in the transformer environment).

```
(syntax-local-make-definition-context)
 → internal-definition-context?
```

Creates an opaque internal-definition context value to be used with `local-expand` and other functions. A transformer should create one context for each set of internal definitions to be expanded, and use it when expanding any form whose lexical context should include the definitions. After discovering an internal `define-values` or `define-syntaxes` form, use `syntax-local-bind-syntaxes` to add bindings to the context.

This procedure must be called during the dynamic extent of a syntax transformer application by the expander, otherwise the `exn:fail:contract` exception is raised.

```
(syntax-local-bind-syntaxes id-list
                            expr
                            intdef-ctx) → void?
  id-list : (listof identifier?)
  expr : (or/c syntax? false/c)
  intdef-ctx : internal-definition-context?
```

Binds each identifier in *id-list* within the internal-definition context represented by *intdef-ctx*, where *intdef-ctx* is the result of `syntax-local-make-definition-context`. Supply `#f` for *expr* when the identifiers correspond to `define-values` bindings, and supply a compile-time expression when the identifiers correspond to `define-syntaxes` bindings; the later case, the number of values produces by the expression should match the number of identifiers, otherwise the `exn:fail:contract:arity` exception is raised.

This procedure must be called during the dynamic extent of a syntax transformer application by the expander, otherwise the `exn:fail:contract` exception is raised.

```
(syntax-local-value  id-stx
                    [failure-thunk
                     intdef-ctx])  → any
  id-stx : syntax?
  failure-thunk : (or/c (-> any) false/c) = #f
  intdef-ctx : (or/c internal-definition-context? = #f
                     false/c)
```

Returns the transformer binding value of *id-stx* in either the context asscoiated with *intdef-ctx* (if not `#f`) or the context of the expression being expanded (if `indef-ctx` is `#f`). If *intdef-ctx* is provided, it must be an extension of the context of the expression being expanded.

If *id-stx* is bound to a rename transformer created with `make-rename-transformer`, `syntax-local-value` effectively calls itself with the target of the rename and returns that result, instead of the rename transformer.

If *id-stx* has no transformer binding (via `define-syntax`, `let-syntax`, etc.) in that environment, the result is obtained by applying *failure-thunk* if not `#f`. If *failure-thunk* is `false`, the `exn:fail:contract` exception is raised.

Resolving *id-stx* can use certificates for the expression being transformed (see §11.7 "Syntax Certificates") as well as inactive certificates associated with *id-stx* (see §11.7 "Syntax Certificates"). Furthermore, if the transformer is defined within a module (i.e., the current transformation was triggered by a use of a module-defined identifier) or if the current expression is being expanded for the body of a module, then resolving *id-stx* can access any identifier defined by the module.

This procedure must be called during the dynamic extent of a syntax transformer application by the expander, otherwise the `exn:fail:contract` exception is raised.

```
(syntax-local-lift-expression stx) → identifier?
  stx : syntax?
```

Returns a fresh identifier, and cooperates with the `module`, `letrec-syntaxes+values`, `define-syntaxes`, `begin-for-syntax`, and top-level expanders to bind the generated identifier to the expression *stx*.

A run-time expression within a module is lifted to the module's top level, just before the expression whose expansion requests the lift. Similarly, a run-time expression outside of a module is lifted to a top-level definition. A compile-time expression in a `letrec-syntaxes+values` or `define-syntaxes` binding is lifted to a `let` wrapper around the corresponding right-hand side of the binding. A compile-time expression within `begin-for-syntax` is lifted to a `define-for-syntax` declaration just before the requesting expression.

Other syntactic forms can capture lifts by using `local-expand/capture-lifts` or `local-transformer-expand/capture-lifts`.

This procedure must be called during the dynamic extent of a syntax transformer application by the expander, otherwise the `exn:fail:contract` exception is raised.

```
(syntax-local-lift-context) → any/c
```

Returns a value that represents the target for expressions lifted via `syntax-local-lift-expression`. That is, for different transformer calls for which this procedure returns the same value (as determined by `eq?`), lifted expressions for the two transformer are moved to the same place. Thus, the result is useful for caching lift information to avoid redundant lifts.

This procedure must be called during the dynamic extent of a syntax transformer application by the expander, otherwise the exn:fail:contract exception is raised.

```
(syntax-local-lift-module-end-declaration stx) → void?
  stx : syntax?
```

Cooperates with the module form to insert *stx* as a top-level declaration at the end of the module currently being expanded. If the current expression being transformed is not within a module form, or if it is not a run-time expression, then the exn:fail:contract exception is raised. If the current expression being transformed is not in the module top-level, then *stx* is eventually expanded in an expression context.

This procedure must be called during the dynamic extent of a syntax transformer application by the expander, otherwise the exn:fail:contract exception is raised.

```
(syntax-local-name) → (or/c symbol? false/c)
```

Returns an inferred name for the expression position being transformed, or #f if no such name is available. See §1.2.6 "Inferred Value Names".

This procedure must be called during the dynamic extent of a syntax transformer application by the expander, otherwise the exn:fail:contract exception is raised.

```
(syntax-local-context)
 → (or/c (one-of 'expression 'top-level 'module 'module-begin)
         list?)
```

Returns an indication of the context for expansion that triggered a syntax transformer call. See §1.2.3.3 "Expansion Context" for more information on contexts.

The symbol results indicate that the expression is being expanded for an expression context, a top-level context, a module context, or a module-begin context.

A list result indicates expansion in an internal-definition context. The identity of the lists's first element (i.e., its eq?ness) reflects the identity of the internal-definition context; in particular two transformer expansions receive the same first value if and only if they are invoked for the same internal-definition context. Later values in the list similarly identify internal-definition contexts that are still being expanded, and that required the expansion of nested internal-definition contexts.

This procedure must be called during the dynamic extent of a syntax transformer application by the expander, otherwise the exn:fail:contract exception is raised.

```
(syntax-local-phase-level) → (or/c exact-integer? false/c)
```

During the dynamic extent of a syntax transformer application by the expander, the result is the phase level of the form being expanded. Otherwise, the result is `0`.

```
(syntax-local-module-exports mod-path) → (listof symbol?)
                                          (listof symbol?)
                                          (listof symbol?)
  mod-path : module-path?
```

Returns three lists of symbols that represent the `provided` bindings of the module named by *mod-path*. The first list corresponds to the phase level 0 exports of the module, the second list corresponds to the phase level -1 exports of the module, and the last list corresponds to the label phase level exports of the module.

This procedure must be called during the dynamic extent of a syntax transformer application by the expander, otherwise the `exn:fail:contract` exception is raised.

```
(syntax-local-get-shadower id-stx) → identifier?
  id-stx : identifier?
```

Returns *id-stx* if no binding in the current expansion context shadows *id-stx*, if *id-stx* has no module bindings in its lexical information, and if the current expansion context is not a module context.

If a binding of `inner-identifier` shadows *id-stx*, the result is the same as `(syntax-local-get-shadower (scheme inner-identifier))`, except that it has the location and properties of *id-stx*.

Otherwise, the result is the same as *id-stx* with its module bindings (if any) removed from its lexical information, and the lexical information of the current module context (if any) added.

Thus, the result is an identifier corresponding to the innermost shadowing of *id-stx* in the current context if its shadowed, and a module-contextless version of *id-stx* otherwise.

This procedure must be called during the dynamic extent of a syntax transformer application by the expander, otherwise the `exn:fail:contract` exception is raised.

```
(syntax-local-certifier [active?])
 → ((syntax?) (any/c (or/c procedure? false/c))
     . ->* . syntax?)
  active? : boolean? = #f
```

Returns a procedure that captures any certificates currently available for `syntax-local-value` or `local-expand`. The procedure accepts one to three arguments: *stx* (required), *key* (optional), and *intro* (optional). The procedure's result is a syntax object like `stx`, except that it includes the captured certificates as inactive (see §11.7 "Syntax Certificates") if *active?* is `#f` (the default) or active otherwise. If `key` is supplied and not `#f`, it is associated with each captured certificate for later use through `syntax-recertify`. If *intro* is supplied, and if it is not `#f` (the default), then it must be a procedure created by `make-syntax-introducer`, in which case the certificate applies only to parts of `stx` that are marked as introduced by *intro*.

Supply `#t` for *active?* when the syntax to be certified can be safely used in any context by any party, and where access to the syntax object should not confer any additional access. Supply `#f` for *active?* when the syntax to be certified is not accessible to parties that might abuse the access that the certificate provides, and when the certified syntax eventually appears (via macro expansion) within a larger expression from which it cannot be safely extracted by other parties.

This procedure must be called during the dynamic extent of a syntax transformer application by the expander, otherwise the `exn:fail:contract` exception is raised.

```
(syntax-transforming?) → boolean?
```

Returns `#t` during the dynamic extent of a syntax transformer application by the expander, `#f` otherwise.

```
(syntax-local-introduce stx) → syntax?
  stx : syntax?
```

Produces a syntax object that is like *stx*, except that a syntax mark for the current expansion is added (possibly canceling an existing mark in parts of *stx*). See §1.2.3.5 "Transformer Bindings" for information on syntax marks.

This procedure must be called during the dynamic extent of a syntax transformer application by the expander, otherwise the `exn:fail:contract` exception is raised.

```
(make-syntax-introducer) → (syntax? . -> . syntax?)
```

Produces a procedure that behaves like `syntax-local-introduce`, but using a fresh syntax mark. Multiple applications of the same `make-syntax-introducer` result procedure use the same mark, and different result procedures use distinct marks.

```
(syntax-local-transforming-module-provides?) → boolean?
```

Returns `#t` while a provide transformer is running (see `make-provide-transformer`) or while a `expand` sub-form of `#%provide` is expanded, `#f` otherwise.

```
(syntax-local-module-defined-identifiers)
 → (listof identifier?) (listof identifier?)
```

Can be called only while `syntax-local-transforming-module-provides?` returns `#t`.

It returns two lists of identifiers corresponding to all definitions within the module being expanded. This information is used for implementing `provide` sub-forms like `all-defined-out`.

The first result list corresponds to phase 0 (i.e., normal) definitions, and the second corresponds to phase -1 (i.e., for-syntax) definitions.

```
(syntax-local-module-required-identifiers mod-path
                                          phase-level)
 → (listof (cons/c (or/c exact-integer? false/c)
                   (listof identifier?)))
  mod-path : (or/c module-path? false/c)
  phase-level : (or/c exact-integer? false/c (one-of/c #t))
```

Can be called only while `syntax-local-transforming-module-provides?` returns `#t`.

It returns an association list mapping phase levels to lists of identifiers. Each list of identifiers includes all bindings imported (into the module being expanded) using the module path *`mod-path`*, or all modules if *`mod-path`* is `#f`. The association list includes all identifiers imported with a *`phase-level`* shift, of all shifts if *`phase-level`* is `#t`.

When an identifier is renamed on import, the result association list includes the identifier by its internal name. Use `identifier-binding` to obtain more information about the identifier.

11.4.1 `require` Transformers

```
(require scheme/require-transform)
```

The bindings documented in this section are provided by the `scheme/require-transform` library, not `scheme/base` or `scheme`.

A transformer binding whose value is a structure with the `prop:require-transformer` property implements a derived *`require-spec`* for `require`.

The transformer is called with the syntax object representing its use as a *`require-spec`* within a `require` form, and the result must be two lists: a list of `import`s and a list of `import-source`s.

If the derived form contains a sub-form that is a *require-spec*, then it can call expand-import to transform the sub-*require-spec* to lists of imports and import sources.

See also define-require-syntax, which supports macro-style require transformers.

```
(expand-import stx) → (listof import?) (listof import-source?)
  stx : syntax?
```

Expands the given *require-spec* to lists of imports and import sources. The latter specifies modules to be instantiated or visited, so the modules that it represents should be a superset of the modules represented in the former list (so that a module will be instantiated or visited even if all of imports are eventually filtered from the former list).

```
(make-require-transformer proc) → require-transformer?
  proc : ((syntax?) . ->* . ((listof import?)
                             (listof import-source?)))
```

Creates a *require transformer* (i.e., a structure with the prop:require-transformer property) using the given procedure as the transformer.

```
prop:require-transformer : struct-type-property?
```

A property to identify require transformers. The property value must be a procedure that takes a syntax object and returns import and import-source lists.

```
(require-transformer? v) → boolean?
  v : any/c
```

Returns #t if *v* has the prop:require-transformer property, #f otherwise.

```
(struct import (local-id
                src-sym
                src-mod-path
                mode
                req-mode
                orig-mode
                orig-stx))
  local-id : identifier?
  src-sym : symbol?
  src-mod-path : module-path?
  mode : (or/c exact-integer? false/c)
  req-mode : (or/c exact-integer? false/c)
```

```
  orig-mode : (or/c exact-integer? false/c)
  orig-stx : syntax?
```

A structure representing a single imported identifier:

- `local-id` — the identifier to be bound within the importing module.
- `src-sym` — the external name of the binding as exported from its source module.
- `src-mod-path` — a module path (relative to the importing module) for the source of the imported binding.
- `orig-stx` — a syntax object for the source of the import, used for error reporting.
- `mode` — the phase level of the binding in the importing module.
- `req-mode` — the phase level shift of the import relative to the exporting module.
- `orig-mode` — the phase level of the binding as exported by the exporting module.

```
(struct import-source (mod-path-stx mode))
  mod-path-stx : (and/c syntax?
                        (lambda (x)
                          (module-path? (syntax->datum x))))
  mode : (or/c exact-integer? false/c)
```

A structure representing an imported module, which must be instantiated or visited even if no binding is imported into a module.

- `mod-path-stx` — a module path (relative to the importing module) for the source of the imported binding.
- `mode` — the phase level shift the import.

```
(syntax-local-require-certifier)
 → ((syntax?) (or/c false/c (syntax? . -> . syntax?))
     . ->* . syntax?)
```

Like `syntax-local-certifier`, but to certify syntax objects that correspond to `require` sub-forms, so that `expand-import` can deconstruct the syntax object as necessary to expand it.

11.4.2 `provide` Transformers

```
(require scheme/provide-transform)
```

The bindings documented in this section are provided by the `scheme/provide-transform` library, not `scheme/base` or `scheme`.

A transformer binding whose value is a structure with the `prop:provide-transformer` property implements a derived *`provide-spec`* for `provide`.

The transformer is called with the syntax object representing its use as a *`provide-spec`* within a `provide` form and a list of symbols representing the export modes specified by enclosing *`provide-spec`*s. The result must be a list of `export`s.

If the derived form contains a sub-form that is a *`provide-spec`*, then it can call `expand-export` to transform the sub-*`provide-spec`* to a lists of exports.

See also `define-provide-syntax`, which supports macro-style `provide` transformers.

```
(expand-export stx modes) → (listof export?)
  stx : syntax?
  modes : (listof (or/c exact-integer? false/c))
```

Expands the given *`provide-spec`* to a list of exports. The *`modes`* list controls the expansion of sub-*`provide-specs`*; for example, an identifier refers to a phase level 0 binding unless the *`modes`* list specifies otherwise. Normally, *`modes`* is either empty or contains a single element.

```
(make-provide-transformer proc) → provide-transformer?
  proc : (syntax? (listof (or/c exact-integer? false/c))
          . -> . (listof export?))
```

Creates a *provide transformer* (i.e., a structure with the `prop:provide-transformer` property) using the given procedure as the transformer.

```
prop:provide-transformer : struct-type-property?
```

A property to identify `provide` transformers. The property value must be a procedure that takes a syntax object and mode list and returns an export list.

```
(provide-transformer? v) → boolean?
  v : any/c
```

Returns #t if *v* has the prop:provide-transformer property, #f otherwise.

```
(struct export (local-id out-sym mode protect? orig-stx))
  local-id : identifier?
  out-sym : symbol?
  mode : (or/c exact-integer? false/c)
  protect? : any/c
  orig-stx : syntax?
```

A structure representing a single imported identifier:

- local-id — the identifier that is bound within the exporting module.
- out-sym — the external name of the binding.
- orig-stx — a syntax object for the source of the export, used for error reporting.
- protect? — indicates whether the identifier should be protected (see §13.9 "Code Inspectors").
- mode — the phase level of the binding in the exporting module.

```
(syntax-local-provide-certifier)
 → ((syntax?) (or/c false/c (syntax? . -> . syntax?))
     . ->* . syntax?)
```

Like syntax-local-certifier, but to certify syntax objects that correspond to provide sub-forms, so that expand-export can deconstruct the syntax object as necessary to expand it.

11.5 Syntax Parameters

```
(require scheme/stxparam)
```

The bindings documented in this section are provided by the scheme/stxparam library, not scheme/base or scheme.

```
(define-syntax-parameter id expr)
```

Binds *id* as syntax to a *syntax parameter*. The *expr* is an expression in the transformer environment that serves as the default value for the syntax parameter. The value is typically obtained by a transformer using syntax-parameter-value.

The *id* can be used with `syntax-parameterize` or `syntax-parameter-value` (in a transformer). If *expr* produces a procedure of one argument or a `make-set!-transformer` result, then *id* can be used as a macro. If *expr* produces a `rename-transformer` result, then *id* can be used as a macro that expands to a use of the target identifier, but `syntax-local-value` of *id* does not produce the target's value.

```
(syntax-parameterize ((id expr) ...) body-expr ...+)
```

Each *id* must be bound to a syntax parameter using `define-syntax-parameter`. Each *expr* is an expression in the transformer environment. During the expansion of the *body-expr*s, the value of each *expr* is bound to the corresponding *id*.

If an *expr* produces a procedure of one argument or a `make-set!-transformer` result, then its *id* can be used as a macro during the expansion of the *body-expr*s. If *expr* produces a `rename-transformer` result, then *id* can be used as a macro that expands to a use of the target identifier, but `syntax-local-value` of *id* does not produce the target's value.

11.5.1 Syntax Parameter Inspection

```
(require scheme/stxparam-exptime)
```

```
(syntax-parameter-value id-stx) → any
  id-stx : syntax?
```

This procedure is intended for use in a transformer environment, where *id-stx* is an identifier bound in the normal environment to a syntax parameter. The result is the current value of the syntax parameter, as adjusted by `syntax-parameterize` form.

This binding is provided `for-syntax` by `scheme/stxparam`, since it is normally used in a transformer. It is provided normally by `scheme/stxparam-exptime`.

```
(make-parameter-rename-transformer id-stx) → any
  id-stx : syntax?
```

This procedure is intended for use in a transformer, where *id-stx* is an identifier bound to a syntax parameter. The result is transformer that behaves as *id-stx*, but that cannot be used with `syntax-parameterize` or `syntax-parameter-value`.

Using `make-parameter-rename-transformer` is analogous to defining a procedure that calls a parameter. Such a procedure can be exported to others to allow access to the parameter value, but not to change the parameter value. Similarly, `make-parameter-rename-`

transformer allows a syntax parameter to used as a macro, but not changed.

The result of make-parameter-rename-transformer is not treated specially by syntax-local-value, unlike the result of make-rename-transformer.

This binding is provided for-syntax by scheme/stxparam, since it is normally used in a transformer. It is provided normally by scheme/stxparam-exptime.

11.6 Syntax Object Properties

Every syntax object has an associated *syntax property* list, which can be queried or extended with syntax-property. Properties are not preserved for a syntax-quoted syntax object in a compiled form that is marshaled to a byte string.

In read-syntax, the reader attaches a 'paren-shape property to any pair or vector syntax object generated from parsing a pair [and] or { and }; the property value is #\[in the former case, and #\{ in the latter case. The syntax form copies any 'paren-shape property from the source of a template to corresponding generated syntax.

Both the syntax input to a transformer and the syntax result of a transformer may have associated properties. The two sets of properties are merged by the syntax expander: each property in the original and not present in the result is copied to the result, and the values of properties present in both are combined with cons (result value first, original value second).

Before performing the merge, however, the syntax expander automatically adds a property to the original syntax object using the key 'origin. If the source syntax has no 'origin property, it is set to the empty list. Then, still before the merge, the identifier that triggered the macro expansion (as syntax) is consed onto the 'origin property so far. The 'origin property thus records (in reverse order) the sequence of macro expansions that produced an expanded expression. Usually, the 'origin value is an immutable list of identifiers. However, a transformer might return syntax that has already been expanded, in which case an 'origin list can contain other lists after a merge. The syntax-track-origin procedure implements this tracking.

Besides 'origin tracking for general macro expansion, MzScheme adds properties to expanded syntax (often using syntax-track-origin) to record additional expansion details:

- When a begin form is spliced into a sequence with internal definitions (see §1.2.3.7 “Internal Definitions”), syntax-track-origin is applied to every spliced element from the begin body. The second argument to syntax-track-origin is the begin form, and the third argument is the begin keyword (extracted from the spliced form).
- When an internal define-values or define-syntaxes form is converted into a letrec-values+syntaxes form (see §1.2.3.7 “Internal Definitions”), syntax-track-origin is applied to each generated binding clause. The second argument to

syntax-track-origin is the converted form, and the third argument is the define-values or define-syntaxes keyword form the converted form.

- When a letrec-values+syntaxes expression is fully expanded, syntax bindings disappear, and the result is either a letrec-values form (if the unexpanded form contained non-syntax bindings), or only the body of the letrec-values+syntaxes form (wrapped with begin if the body contained multiple expressions). To record the disappeared syntax bindings, a property is added to the expansion result: an immutable list of identifiers from the disappeared bindings, as a 'disappeared-binding property.

- When a subtyping define-struct form is expanded, the identifier used to reference the base type does not appear in the expansion. Therefore, the define-struct transformer adds the identifier to the expansion result as a 'disappeared-use property.

- When a reference to an unexported or protected identifier from a module is discovered (and the reference is certified; see §11.7 “Syntax Certificates”), the 'protected property is added to the identifier with a #t value.

- When or read-syntax or read-honu-syntax generates a syntax object, it attaches a property to the object (using a private key) to mark the object as originating from a read. The syntax-original? predicate looks for the property to recognize such syntax objects. (See §11.2 “Syntax Object Content” for more information.)

See §11.8.1 “Information on Expanded Modules” for information about properties generated by the expansion of a module declaration. See lambda and §1.2.6 “Inferred Value Names” for information about properties recognized when compiling a procedure. See current-compile for information on properties and byte codes.

```
(syntax-property stx key v) → syntax?
  stx : syntax?
  key : any/c
  v : any/c
(syntax-property stx key) → any
  stx : syntax?
  key : any/c
```

The three-argument form extends *stx* by associating an arbitrary property value *v* with the key *key*; the result is a new syntax object with the association (while *stx* itself is unchanged).

The two-argument form returns an arbitrary property value associated to *stx* with the key *key*, or #f if no value is associated to *stx* for *key*.

```
(syntax-property-symbol-keys stx) → list?
```

```
stx : syntax?
```

Returns a list of all symbols that as keys have associated properties in *stx*. Uninterned symbols (see §3.6 "Symbols") are not included in the result list.

```
(syntax-track-origin new-stx
                     orig-stx
                     id-stx)  → any
  new-stx : syntax?
  orig-stx : syntax?
  id-stx : syntax?
```

Adds properties to *new-stx* in the same way that macro expansion adds properties to a transformer result. In particular, it merges the properties of *orig-stx* into *new-stx*, first adding *id-stx* as an 'origin property, and it returns the property-extended syntax object. Use the syntax-track-origin procedure in a macro transformer that discards syntax (corresponding to *orig-stx* with a keyword *id-stx*) leaving some other syntax in its place (corresponding to *new-stx*).

For example, the expression

```
(or x y)
```

expands to

```
(let ((or-part x)) (if or-part or-part (or y)))
```

which, in turn, expands to

```
(let-values ([(or-part) x]) (if or-part or-part y))
```

The syntax object for the final expression will have an 'origin property whose value is (list (quote-syntax let) (quote-syntax or)).

11.7 Syntax Certificates

A *syntax certificate* combines a syntax mark (see §1.2.3.5 "Transformer Bindings"), a module path index or symbol module name (see §13.4.2 "Compiled Modules and References"), an inspector (see §13.9 "Code Inspectors"), and an arbitrary key object. A certificate is attached as either an *active certificate* or an *inactive certificate*.

The datum->syntax procedure never transfers an active certificate from one syntax object to another. The syntax-recertify procedure can be used to transfer a certificate from one syntax object to another, but only if the certificate's key is provided, or if a sufficiently powerful inspector is provided. Thus, a certificate's inspector serves two roles: it determines the certificate's power to grant access, and also allows the certificate to be moved arbitrarily by anyone with a more powerful inspector.

The expander generates a certificate when it applies a syntax transformer. The syntax mark in the certificate is fresh, the certificate's module reference corresponds to the module that defined the transformer binding, the inspector is the inspector for the module's declaration (see §13.9 "Code Inspectors"), and the key object is hidden. (Applying the result of `syntax-local-certifier` can introduce certificates with other keys.) The certificate's mark is applied to both the input and output of the syntax transformer, so that it identifies every piece of syntax that was introduced by the transformer (see §1.2.3.5 "Transformer Bindings"). The expander attaches this certificate to parts of the transformer's result, depending on the shape and properties of the result:

- If the result has a `'certify-mode` property (see §11.6 "Syntax Object Properties") that is `'opaque`, then the certificate is attached to the immediate syntax object.
- If the result has a `'certify-mode` property that is `'transparent`, then the certificate is also propagated recursively to syntax object that corresponds to elements of the syntax object's datum as a list (or, more precisely, to the `car`s of the datum as reached by any number of `cdr`s). This recursive propagation uses syntax properties and shapes, as for the immediate attachment.
- If the result has a `'certify-mode` property that is `'transparent-binding`, then the certificate is attached to similar to `'transparent`, but further treating the syntax object corresponding to the second list element as having a `'transparent` value for the `'certify-mode` property if it does not already have a `'certify-mode` property value.
- If the result has no `'certify-mode` property value, but its datum is a pair, and if the syntax object corresponding to the `car` of the pair is an identifier bound to `begin`, then the certificate is propagated as if the syntax object had the `'transparent` property value.
- If the result has no `'certify-mode` property value, but its datum is a pair, and if the syntax object corresponding to the `car` of the pair is an identifier bound to `define-values` or `define-syntaxes`, then the certificate is propagated as if the syntax object had the `'transparent-binding` property value.

The the expander attaches a new active certificate to a syntax object, it also removes any inactive certificates attached to any syntax object within the one where the certificate is attached, and it re-attaches the formerly inactive certificates as active certificates along with the new one.

As the expander processes a form, it accumulates active certificates that are attached to enclosing forms as part of the expansion context:

- To check access to an unexported identifier, the expander checks each of the identifier's marks and module bindings; if, for some mark, the identifier's enclosing expressions include a certificate with the mark, the identifier's binding module, and with an inspector that controls the module's invocation (as opposed to the module's declaration; see again §13.9 "Code Inspectors"), then the access is allowed. To check access to a protected identifier, only the certificate's mark and inspector are used (i.e., the module that bound the transformer is irrelevant, as long as it was evaluated with a sufficiently powerful inspector). The certificate key is not used in checking references.

- To check access to a locally bound identifier, the expander checks the marks of the binding and reference identifiers; for every mark that they have in common, if the reference identifier has a certificate for the mark from an enclosing expression, the binding identifier must have a certificate for the mark from an enclosing expression, otherwise the reference is disallowed. (The reference identifier can have additional certificates for marks that are not attached to the binding identifier.) The binding module (if any) and the certificate key are not used for checking a local reference.

- When the expander encounters a `quote-syntax` form, it attaches all accumulated active certificates from the expressions's context to the quoted syntax objects. The certificates are attached as inactive certificates.

```
(syntax-recertify new-stx
                  old-stx
                  inspector
                  key)      → syntax?
  new-stx : syntax?
  old-stx : syntax?
  inspector : inspector?
  key : any/c
```

Copies certain certificates of *old-stx* to *new-stx*: a certificate is copied if its inspector is either *inspector* or controlled by *inspector*, or if the certificate's key is *key*; otherwise the certificate is not copied. The result is a syntax object like *new-stx*, but with the copied certificates. (The *new-stx* object itself is not modified.) Both active certificates and inactive certificates are copied.

11.8 Expanding Top-Level Forms

```
(expand top-level-form) → syntax?
  top-level-form : any/c
```

Expands all non-primitive syntax in *top-level-form*, and returns a syntax object for the expanded form that contains only core forms, matching the grammar specified by §1.2.3.1 "Fully Expanded Programs".

Before *top-level-form* is expanded, its lexical context is enriched with `namespace-syntax-introduce`, just as for `eval`. Use `syntax->datum` to convert the returned syntax object into a printable datum.

```
(expand-syntax stx) → syntax?
  stx : syntax?
```

Like `(expand stx)`, except that the argument must be a syntax object, and its lexical context is not enriched before expansion.

```
(expand-once top-level-form) → syntax?
  top-level-form : any/c
```

Partially expands `form-level-form` and returns a syntax object for the partially-expanded expression. Due to limitations in the expansion mechanism, some context information may be lost. In particular, calling `expand-once` on the result may produce a result that is different from expansion via `expand`.

Before *top-level-form* is expanded, its lexical context is enriched with `namespace-syntax-introduce`, as for `eval`.

```
(expand-syntax-once stx) → syntax?
  stx : syntax?
```

Like `(expand-once stx)`, except that the argument must be a syntax object, and its lexical context is not enriched before expansion.

```
(expand-to-top-form top-level-form) → syntax?
  top-level-form : any/c
```

Partially expands *top-level-form* to reveal the outermost syntactic form. This partial expansion is mainly useful for detecting top-level uses of `begin`. Unlike the result of `expand-once`, expanding the result of `expand-to-top-form` with `expand` produces the same result as using `expand` on the original syntax.

Before `stx-or-sexpr` is expanded, its lexical context is enriched with `namespace-syntax-introduce`, as for `eval`.

```
(expand-syntax-to-top-form stx) → syntax?
  stx : syntax?
```

Like (expand-to-top-form *stx*), except that the argument must be a syntax object, and its lexical context is not enriched before expansion.

11.8.1 Information on Expanded Modules

Information for an expanded `module` declaration is stored in a set of syntax properties (see §11.6 "Syntax Object Properties") attached to the syntax object:

- 'module-direct-requires — a list of module path indexes (or symbols) representing the modules explicitly imported into the module.
- 'module-direct-for-syntax-requires — a list of module path indexes (or symbols) representing the modules explicitly for-syntax imported into the module.
- 'module-direct-for-template-requires — a list of module path indexes (or symbols) representing the modules explicitly for-template imported into the module.
- 'module-variable-provides — a list of provided items, where each item is one of the following:
  - symbol — represents a locally defined variable that is provided with its defined name.
  - (cons *provided-sym* *defined-sym*) — represents a locally defined variable that is provided with renaming; the first symbol is the exported name, and the second symbol is the defined name.
  - (list* module-path-index *provided-sym* *defined-sym*) — represents a re-exported and possibly re-named variable from the specified module; module-path-index is either a module path index or symbol (see §13.4.2 "Compiled Modules and References"), indicating the source module for the binding. The *provided-sym* is the external name for the re-export, and *defined-sym* is the originally defined name in the module specified by module-path-index.
- 'module-syntax-provides — like 'module-variable-provides, but for syntax exports instead of variable exports.
- 'module-indirect-provides — a list of symbols for variables that are defined in the module but not exported; they may be exported indirectly through macro expansions. Definitions of macro-generated identifiers create uninterned symbols in this list.

11.9 File Inclusion

```
(require scheme/include)
```

The bindings documented in this section are provided by the `scheme/include` and `scheme` libraries, but not `scheme/base`.

```
(include path-spec)

include-spec = string
             | (file string)
             | (lib string ...+)
```

Inlines the syntax in the file designated by *path-spec* in place of the `include` expression.

A *path-spec* resembles a subset of the *mod-path* forms for `require`, but it specifies a file whose content need not be a module. That is, *string* refers to a file using a platform-independent relative path, `(file string)` refers to a file using platform-specific notation, and `(lib string ...)` refers to a file within a collection.

If *path-spec* specifies a relative path, the path is resolved relative to the source for the `include` expression, if that source is a complete path string. If the source is not a complete path string, then *path-spec* is resolved relative to `(current-load-relative-directory)` if it is not `#f`, or relative to `(current-directory)` otherwise.

The included syntax is given the lexical context of the `include` expression, while the included syntax's source location refers to its actual source.

```
(include-at/relative-to context source path-spec)
```

Like `include`, except that the lexical context of *context* is used for the included syntax, and a relative *path-spec* is resolved with respect to the source of *source*. The *context* and *source* elements are otherwise discarded by expansion.

```
(include/reader path-spec reader-expr)
```

Like `include`, except that the procedure produced by the expression *reader-expr* is used to read the included file, instead of `read-syntax`.

The *reader-expr* is evaluated at expansion time in the transformer environment. Since it serves as a replacement for `read-syntax`, the expression's value should be a procedure that consumes two inputs—a string representing the source and an input port—and produces a syntax object or `eof`. The procedure will be called repeatedly until it produces `eof`.

The syntax objects returned by the procedure should have source location information, but usually no lexical context; any lexical context in the syntax objects will be ignored.

```
(include-at/relative-to/reader context source path-spec reader-expr)
```

Combines `include-at/relative-to` and `include/reader`.

12 Input and Output

12.1 Ports

Ports produce and consume bytes. When a port is provided to a character-based operation, the port's bytes are decoded; see §12.1.1 "Encodings and Locales".

The global variable `eof` is bound to the end-of-file value, and `eof-object?` returns `#t` only when applied to this value. Reading from a port produces an end-of-file result when the port has no more data, but some ports may also return end-of-file mid-stream. For example, a port connected to a Unix terminal returns an end-of-file when the user types control-D; if the user provides more input, the port returns additional bytes after the end-of-file.

Every port has a name, as reported by `object-name`. The name can be any value, and it is used mostly for error-reporting purposes. The `read-syntax` procedure uses the name of an input port as the default source location for the syntax objects that it produces.

12.1.1 Encodings and Locales

When a port is provided to a character-based operation, such as `read-char` or `read`, the port's bytes are read and interpreted as a UTF-8 encoding of characters. Thus, reading a single character may require reading multiple bytes, and a procedure like `char-ready?` may need to peek several bytes into the stream to determine whether a character is available. In the case of a byte stream that does not correspond to a valid UTF-8 encoding, functions such as `read-char` may need to peek one byte ahead in the stream to discover that the stream is not a valid encoding.

When an input port produces a sequence of bytes that is not a valid UTF-8 encoding in a character-reading context, then bytes that constitute an invalid sequence are converted to the character `#\`. Specifically, bytes 255 and 254 are always converted to `#\`, bytes in the range 192 to 253 produce `#\` when they are not followed by bytes that form a valid UTF-8 encoding, and bytes in the range 128 to 191 are converted to `#\` when they are not part of a valid encoding that was started by a preceding byte in the range 192 to 253. To put it another way, when reading a sequence of bytes as characters, a minimal set of bytes are changed to the encoding of `#\` so that the entire sequence of bytes is a valid UTF-8 encoding.

See §3.4 "Byte Strings" for procedures that facilitate conversions using UTF-8 or other encodings. See also `reencode-input-port` and `reencode-output-port` for obtaining a UTF-8-based port from one that uses a different encoding of characters.

A *locale* captures information about a user's culture-specific interpretation of character sequences. In particular, a locale determines how strings are "alphabetized," how a lowercase character is converted to an uppercase character, and how strings are compared without regard to case. String operations such as `string-ci=?` are *not* sensitive to the current locale, but operations such as `string-locale-ci=?` (see §3.3 "Strings") produce results consistent with the current locale.

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

A Unix user selects a locale by setting environment variables, such as `LC_ALL`. Under Windows and Mac OS X, the operating system provides other mechanisms for setting the locale. Within Scheme, the current locale can be changed by setting the `current-locale` parameter. The locale name within Scheme is a string, and the available locale names depend on the platform and its configuration, but the `""` locale means the current user's default locale; under Windows and Mac OS X, the encoding for `""` is always UTF-8, and locale-sensitive operations use the operating system's native interface. (In particular, setting the `LC_ALL` and `LC_CTYPE` environment variables do not affect the locale `""` under Mac OS X. Use `getenv` and `current-locale` to explicitly install the environment-specified locale, if desired.) Setting the current locale to `#f` makes locale-sensitive operations locale-insensitive, which means using the Unicode mapping for case operations and using UTF-8 for encoding.

```
(current-locale) → (or/c string? false/c)
(current-locale locale) → void?
  locale : (or/c string? false/c)
```

A parameter that determines the current locale for procedures such as `string-locale-ci=?`.

When locale sensitivity is disabled by setting the parameter to `#f`, strings are compared (etc.) in a fully portable manner, which is the same as the standard procedures. Otherwise, strings are interpreted according to a locale setting (in the sense of the C library's `setlocale`). The `""` locale is always a synonym for the current machine's default locale, and it is the default. The `"C"` locale is also always available; setting the locale to `"C"` is the same as disabling locale sensitivity with `#f` only when string operations are restricted to the first 128 characters. Other locale names are platform-specific.

String or character printing with `write` is not affected by the parameter, and neither are symbol case or regular expressions (see §3.7 "Regular Expressions").

12.1.2 Managing Ports

```
(input-port? v) → boolean?
  v : any/c
```

Returns `#t` if `v` is an input port, `#f` otherwise.

```
(output-port? v) → boolean?
  v : any/c
```

Returns `#t` if `v` is an output port, `#f` otherwise.

```
(port? v) → boolean?
  v : any/c
```

Returns `#t` if either `(input-port? v)` or `(output-port? v)` is `#t`, `#f` otherwise.

```
(close-input-port in) → void?
  in : input-port?
```

Closes the input port `in`. For some kinds of ports, closing the port releases lower-level resources, such as a file handle. If the port is already closed, `close-input-port` has no effect.

```
(close-output-port out) → void?
  out : output-port?
```

Closes the output port `out`. For some kinds of ports, closing the port releases lower-level resources, such as a file handle. Also, if the port is buffered, closing may first flush the port before closing it, and this flushing process can block. If the port is already closed, `close-output-port` has no effect.

```
(port-closed? port) → boolean?
  port : port?
```

Returns `#t` if the input or output port `port` is closed, `#f` otherwise.

```
(current-input-port) → input-port?
(current-input-port in) → void?
  in : input-port?
```

A parameter that determines a default input port for many operations, such as `read`.

```
(current-output-port) → output-port?
(current-output-port out) → void?
  out : output-port?
```

A parameter that determines a default output port for many operations, such as `write`.

```
(current-error-port) → output-port?
(current-error-port out) → void?
  out : output-port?
```

A parameter that determines an output port that is typically used for errors and logging. For example, the default error display handler writes to this port.

```
(file-stream-port? port) → boolean?
  port : port?
```

Returns `#t` if the given port is a file-stream port (see §12.1.5 "File Ports"), `#f` otherwise.

```
(terminal-port? port) → boolean?
  port : port?
```

Returns `#t` if the given port is attached to an interactive terminal, `#f` otherwise.

```
eof : eof-object?
```

A value (distinct from all other values) that represents an end-of-file.

```
(eof-object? a) → boolean?
  a : any/c
```

Returns `#t` is `v` is `eof`, `#f` otherwise.

12.1.3 Port Buffers and Positions

Some ports—especially those that read from and write to files—are internally buffered:

- An input port is typically block buffered by default, which means that on any read, the buffer is filled with immediately-available bytes to speed up future reads. Thus, if a file is modified between a pair of reads to the file, the second read can produce stale data. Calling `file-position` to set an input port's file position flushes its buffer.

- And output port is typically block buffered by default, though a terminal output port is line buffered, and the initial error output port is unbuffered. An output buffer is filled with a sequence of written bytes to be committed as a group, either when the buffer is full (in block mode), when a newline is written (in line mode), when the port is closed via `close-output-port`, or when a flush is explicitly requested via a procedure like `flush-output`.

If a port supports buffering, its buffer mode can be changed via `file-stream-buffer-mode` (even if the port is not a file-stream port).

For an input port, peeking always places peeked bytes into the port's buffer, even when the port's buffer mode is `'none`; furthermore, on some platforms, testing the port for input (via `char-ready?` or `sync`) may be implemented with a peek. If an input port's buffer mode is `'none`, then at most one byte is read for `read-bytes-avail!*`, `read-bytes-avail!`, `peek-bytes-avail!*`, or `peek-bytes-avail!`; if any bytes are buffered in the port (e.g., to satisfy a previous peek), the procedures may access multiple buffered bytes, but no further bytes are read.

In addition, the initial current output and error ports are automatically flushed when `read`, `read-line`, `read-bytes`, `read-string`, etc. are performed on the initial standard input port; more precisely, flushing is performed by the default port read handler (see `port-read-handler`).

```
(flush-output [out]) → void?
  out : output-port? = (current-output-port)
```

Forces all buffered data in the given output port to be physically written. Only file-stream ports, TCP ports, and custom ports (see §12.1.9 "Custom Ports") use buffers; when called on a port without a buffer, `flush-output` has no effect.

```
(file-stream-buffer-mode port)
 → (one-of/c 'none 'line 'block #f)
  port : port?
(file-stream-buffer-mode port mode) → void?
  port : port?
  mode : (one-of/c 'none 'line 'block)
```

Gets or sets the buffer mode for *port*, if possible. File-stream ports support setting the buffer mode, TCP ports (see §14.3 "Networking") support setting and getting the buffer mode, and custom ports (see §12.1.9 "Custom Ports") may support getting and setting buffer modes.

If *mode* is provided, it must be one of `'none`, `'line` (output only), or `'block`, and the port's buffering is set accordingly. If the port does not support setting the mode, the `exn:fail` exception is raised.

If *mode* is not provided, the current mode is returned, or `#f` is returned if the mode cannot be determined. If `file-stream-port` is an input port and *mode* is `'line`, the `exn:fail:contract` exception is raised.

```
(file-position port) → nonnegative-exact-integer?
  port : port?
(file-position port pos) → void?
  port : port?
  pos : (or/c nonnegative-exact-integer? eof-object?)
```

Returns or sets the current read/write position of *port*.

Calling `file-position` without a position on a non-file/non-string input port returns the number of bytes that have been read from that port if the position is known (see §12.1.4 "Counting Positions, Lines, and Columns"), otherwise the `exn:fail:filesystem` exception is raised.

For file-stream ports and string ports, the position-setting variants sets the read/write position to *pos* relative to the beginning of the file/string if *pos* is a number, or to the current end of the file/string if *pos* is `eof`. In position-setting mode, `file-position` raises the `exn:fail:contract` exception for port kinds other than file-stream and string ports. Furthermore, not all file-stream ports support setting the position; f `file-position` is called with a position argument on such a file-stream port, the `exn:fail:filesystem` exception is raised.

When `file-position` sets the position *pos* beyond the current size of an output file or string, the file/string is enlarged to size *pos* and the new region is filled with `0` bytes. If *pos* is beyond the end of an input file or string, then reading thereafter returns `eof` without changing the port's position.

When changing the file position for an output port, the port is first flushed if its buffer is not empty. Similarly, setting the position for an input port clears the port's buffer (even if the new position is the same as the old position). However, although input and output ports produced by `open-input-output-file` share the file position, setting the position via one port does not flush the other port's buffer.

12.1.4 Counting Positions, Lines, and Columns

By default, Scheme keeps track of the *position* in a port as the number of bytes that have been read from or written to any port (independent of the read/write position, which is accessed or changed with `file-position`). Optionally, however, Scheme can track the position in terms of characters (after UTF-8 decoding), instead of bytes, and it can track *line locations* and *column locations*; this optional tracking must be specifically enabled for a port via `port-count-lines!` or the `port-count-lines-enabled` parameter. Position, line, and column locations for a port are used by `read-syntax` and `read-honu-syntax`. Position and line locations are numbered from 1; column locations are numbered from 0.

When counting lines, Scheme treats linefeed, return, and return-linefeed combinations as a line terminator and as a single position (on all platforms). Each tab advances the column count to one before the next multiple of 8. When a sequence of bytes in the range 128 to 253 forms a UTF-8 encoding of a character, the position/column is incremented is incremented once for each byte, and then decremented appropriately when a complete encoding sequence is discovered. See also §12.1 "Ports" for more information on UTF-8 decoding for ports.

A position is known for any port as long as its value can be expressed as a fixnum (which is more than enough tracking for realistic applications in, say, syntax-error reporting). If the position for a port exceeds the value of the largest fixnum, then the position for the port becomes unknown, and line and column tacking is disabled. Return-linefeed combinations are treated as a single character position only when line and column counting is enabled.

```
(port-count-lines! port) → void?
  port : port?
```

Turns on line and column counting for a port. Counting can be turned on at any time, though generally it is turned on before any data is read from or written to a port. When a port is created, if the value of the `port-count-lines-enabled` parameter is true, then line counting is automatically enabled for the port. Line counting cannot be disabled for a port after it is enabled.

```
(port-next-location port)
 → (or/c positive-exact-integer? false/c)
   (or/c nonnegative-exact-integer? false/c)
   (or/c positive-exact-integer? false/c)
  port : port?
```

Returns three values: an integer or `#f` for the line number of the next read/written item, an integer or `#f` for the next item's column, and an integer or `#f` for the next item's position. The next column and position normally increases as bytes are read from or written to the port, but if line/character counting is enabled for *port*, the column and position results can decrease after reading or writing a byte that ends a UTF-8 encoding sequence.

```
(port-count-lines-enabled) → boolean?
(port-count-lines-enabled on?) → void?
  on? : any/c
```

A parameter that determines whether line counting is enabled automatically for newly created ports. The default value is #f.

12.1.5 File Ports

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

When an input or output file-stream port is created, it is placed into the management of the current custodian (see §13.6 "Custodians").

```
(open-input-file path [#:mode mode-flag]) → input-port?
  path : path-string?
  mode-flag : (one-of/c 'binary 'text) = 'binary
```

Opens the file specified by *path* for input. The *mode-flag* argument specifies how the file's bytes are translated on input:

- 'binary — bytes are returned from the port exactly as they are read from the file.
- 'text — return and linefeed bytes (10 and 13) as read from the file are filtered by the port in a platform specific manner:
  - Unix and Mac OS X: no filtering occurs.
  - Windows: a return-linefeed combination from a file is returned by the port as a single linefeed; no filtering occurs for return bytes that are not followed by a linefeed, or for a linefeed that is not preceded by a return.

Under Windows, 'text mode works only with regular files; attempting to use 'text with other kinds of files triggers an exn:fail:filesystem exception.

Otherwise, the file specified by *path* need not be a regular file. It might a device that is connected through the filesystem, such as "aux" under Windows or "/dev/null" under Unix. In all cases, the port is buffered by default.

The port produced by open-input-port should be explicitly closed, either though close-input-port or indirectly via custodian-shutdown-all, to release the OS-level file handle. The input port will not closed automatically if it is otherwise available for garbage collection (see §1.1.7 "Garbage Collection"); a will could be associated input port to close it more automatically (see §15.3 "Wills and Executors").

A path value that is the cleansed version of *path* is used as the name of the opened port.

```
(open-output-file path
                  [#:mode mode-flag
                   #:exists exists-flag]) → output-port?
  path : path-string?
  mode-flag : (one-of/c 'binary 'text) = 'binary
  exists-flag : (one-of/c 'error 'append 'update 'can-update
                          'replace 'truncate
                          'must-truncate 'truncate/replace)
              = 'error
```

Opens the file specified by *path* for output. The *mode-flag* argument specifies how bytes written to the port are translated when written to the file:

- `'binary` — bytes are written to the file exactly as written to the port.
- `'text` — under Windows, a linefeed byte (10) written to the port is translated to a return-linefeed combination in the file; no filtering occurs for returns.

Under Windows, `'text` mode works only with regular files; attempting to use `'text` with other kinds of files triggers an `exn:fail:filesystem` exception.

The *exists-flag* argument specifies how to handle/require files that already exist:

- `'error` — raise `exn:fail:filesystem` if the file exists.
- `'replace` — remove the old file, if it exists, and write a new one.
- `'truncate` — remove all old data, if the file exists.
- `'must-truncate` — remove all old data in an existing file; if the file does not exist, the `exn:fail:filesystem` exception is raised.
- `'truncate/replace` — try `'truncate`; if it fails (perhaps due to file permissions), try `'replace`.
- `'update` — open an existing file without truncating it; if the file does not exist, the `exn:fail:filesystem` exception is raised.
- `'can-update` — open an existing file without truncating it, or create the file if it does not exist.
- `'append` — append to the end of the file, whether it already exists or not; under Windows, `'append` is equivalent to `'update`, except that the file is not required to exist, and the file position is immediately set to the end of the file after opening it.

The file specified by *path* need not be a regular file. It might a device that is connected through the filesystem, such as "aux" under Windows or "/dev/null" under Unix. The output port is block-buffered by default, unless the file corresponds to a terminal, in which case is it line buffered bu default.

The port produced by `open-output-port` should be explicitly closed, either though `close-output-port` or indirectly via `custodian-shutdown-all`, to release the OS-level file handle. The output port will not closed automatically if it is otherwise available for garbage collection (see §1.1.7 "Garbage Collection"); a will could be associated input port to close it more automatically (see §15.3 "Wills and Executors").

A path value that is the cleansed version of *path* is used as the name of the opened port.

```
(open-input-output-file  path
                        [#:mode mode-flag
                         #:exists exists-flag])
 → input-port? output-port?
  path : path-string?
  mode-flag : (one-of/c 'binary 'text) = 'binary
  exists-flag : (one-of/c 'error 'append 'update
                          'replace 'truncate 'truncate/replace)
              = 'error
```

Like `open-output-file`, but producing two values: an input port and an output port. The two ports are connected in that they share the underlying file device. This procedure is intended for use with special devices that can be opened by only one process, such as "COM1" in Windows. For regular files, sharing the device can be confusing. For example, using one port does not automatically flush the other port's buffer, and reading or writing in one port moves the file position (if any) for the other port. For regular files, use separate `open-input-file` and `open-output-file` calls to avoid confusion.

```
(call-with-input-file  path
                       proc
                      [#:mode mode-flag]) → any
  path : path-string?
  proc : (input-port? . -> . any)
  mode-flag : (one-of/c 'binary 'text) = 'binary
```

Calls `open-input-port` with the *path* and *mode-flag* arguments, and passes the resulting port to *proc*. The result of *proc* is the result of the `call-with-input-file` call, but the newly opened port is closed when `thunk` return.

```
(call-with-output-file  path
                        proc
                       [#:mode mode-flag
                        #:exists exists-flag]) → any
  path : path-string?
  proc : (output-port? . -> . any)
  mode-flag : (one-of/c 'binary 'text) = 'binary
  exists-flag : (one-of/c 'error 'append 'update
                          'replace 'truncate 'truncate/replace)
              = 'error
```

Analogous to `call-with-input-file`, but passing *path*, *mode-flag* and *exists-flag* to `open-output-file`.

```
(call-with-input-file*  path
                        proc
                       [#:mode mode-flag]) → any
  path : path-string?
  proc : (input-port? . -> . any)
  mode-flag : (one-of/c 'binary 'text) = 'binary
```

Like `call-with-input-file`, but the newly opened port is closed whenever control escapes the the dynamic extent of the `call-with-input-file*` call, whether through *proc*'s return, a continuation application, or a prompt-based abort.

```
(call-with-output-file*  path
                         proc
                        [#:mode mode-flag
                         #:exists exists-flag]) → any
  path : path-string?
  proc : (output-port? . -> . any)
  mode-flag : (one-of/c 'binary 'text) = 'binary
  exists-flag : (one-of/c 'error 'append 'update
                          'replace 'truncate 'truncate/replace)
              = 'error
```

Like `call-with-output-file`, but the newly opened port is closed whenever control escapes the the dynamic extent of the `call-with-output-file*` call, whether through *proc*'s return, a continuation application, or a prompt-based abort.

```
(with-input-from-file  path
                       thunk
                      [#:mode mode-flag]) → any
  path : path-string?
```

```
  thunk : (-> any)
  mode-flag : (one-of/c 'binary 'text) = 'binary
```

Like `call-with-input-file*`, but instead of passing the newly opened port to the given procedure argument, the port is installed as the current input port (see `current-input-port`) using `parameterize` around the call to *thunk*.

```
(with-output-to-file  path
                      thunk
                     [#:mode mode-flag
                      #:exists exists-flag]) → any
  path : path-string?
  thunk : (-> any)
  mode-flag : (one-of/c 'binary 'text) = 'binary
  exists-flag : (one-of/c 'error 'append 'update
                          'replace 'truncate 'truncate/replace)
              = 'error
```

Like `call-with-output-file*`, but instead of passing the newly opened port to the given procedure argument, the port is installed as the current output port (see `current-output-port`) using `parameterize` around the call to *thunk*.

```
(port-file-identity port) → any
  port : file-stream-port?
```

Returns an exact positive integer that represents the identity of the device and file read or written by `file-stream-port`. For two ports whose open times overlap, the result of `port-file-identity` is the same for both ports if and only if the ports access the same device and file. For ports whose open times do not overlap, no guarantee is provided for the port identities (even if the ports actually access the same file)—except as can be inferred through relationships with other ports. If `file-stream-port` is closed, the `exn:fail` exception is raised. Under Windows 95, 98, and Me, if `file-stream-port` is connected to a pipe instead of a file, the `exn:fail:filesystem` exception is raised.

12.1.6 String Ports

String input and output ports do not need to be explicitly closed. The `file-position` procedure works for string ports in position-setting mode.

```
(open-input-bytes bstr [name]) → input-port?
  bstr : bytes?
  name : any/c = 'string
```

Creates an input port that reads characters from `bstr` (see §3.4 "Byte Strings"). Modifying `bstr` afterward does not affect the byte stream produced by the port. The optional `name` argument is used as the name for the returned port.

```
(open-input-string str [name]) → input-port?
  str : string?
  name : any/c = 'string
```

Creates an input port that reads bytes from the UTF-8 encoding (see §12.1.1 "Encodings and Locales") of `str`. The optional `name` argument is used as the name for the returned port.

```
(open-output-bytes [name]) → output-port?
  name : any/c = 'string
```

Creates an output port that accumulates the output into a byte string. The optional `name` argument is used as the name for the returned port.

```
(open-output-string [name]) → output-port?
  name : any/c = 'string
```

The same as `open-output-bytes`.

```
(get-output-bytes  out
                  [reset?
                   start-pos
                   end-pos]) → bytes?
  out : output-port?
  reset? : any/c = #f
  start-pos : nonnegative-exact-integer? = 0
  end-pos : nonnegative-exact-integer? = #f
```

Returns the bytes accumulated in `out` so far in a freshly-allocated byte string (including any bytes written after the port's current position, if any). the `out` port must be a string output port produced by `open-output-bytes` (or `open-output-string`) or a structure whose `prop:output-port` property refers to such an output port (transitively).

If `reset?` is true, then all bytes are removed from the port, and the port's position is reset to `0`; if `reset?` is `#f`, then all bytes remain in the port for further accumulation (so they are returned for later calls to `get-output-bytes` or `get-output-string`), and the port's position is unchanged.

The `start-pos` and `end-pos` arguments specify the range of bytes in the port to return; supplying `start-pos` and `end-pos` is the same as using `subbytes` on the result of `get-`

output-bytes, but supplying them to get-output-bytes can avoid an allocation. The *end-pos* argument can be #f, which corresponds to not passing a second argument to sub-bytes.

```
(get-output-string out) → string?
  out : output-port?
```

Returns (bytes->string/utf-8 (get-output-bytes *out*) #\?).

Examples:

```
> (define i (open-input-string "hello world"))
> (define o (open-output-string))
> (write (read i) o)
> (get-output-string o)
"hello"
```

12.1.7 Pipes

A Scheme *pipe* is internal to Scheme, and not related to OS-level pipes (which are file-stream ports) for communicating between different processes.

```
(make-pipe [limit input-name output-name])
 → input-port? output-port?
  limit : positive-exact-integer? = #f
  input-name : any/c = 'pipe
  output-name : any/c = 'pipe
```

Returns two port values: 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, with no intermediate buffering. The ports do not need to be explicitly closed.

If *limit* is #f, the new pipe holds an unlimited number of unread bytes (i.e., limited only by the available memory). If *limit* is a positive number, then the pipe will hold at most *limit* unread/unpeeked bytes; writing to the pipe's output port thereafter will block until a read or peek from the input port makes more space available. (Peeks effectively extend the port's capacity until the peeked bytes are read.)

The optional *input-name* and *output-name* are used as the names for the returned input and out ports, respectively.

```
(pipe-content-length pipe-port) → exact-nonnegative-integer?
  pipe-port : port?
```

Returns the number of bytes contained in a pipe, where *pipe-port* is either of the pipe's ports produced by make-pipe. The pipe's content length counts all bytes that have been written to the pipe and not yet read (though possibly peeked).

12.1.8 Structures as Ports

`prop:input-port : struct-type-property?`

`prop:output-port : struct-type-property?`

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

Each property value can be either of the following:

- An input port (for prop:input-port) or output port (for prop:input-port): In this case, using the structure as port is equivalent to using the given one.
- An exact, non-negative integer between 0 (inclusive) and number of non-automatic fields in the structure type (exclusive, not counting supertype fields): The integer identifies a field in the structure, and the field must be designated as immutable. If the field contains an input port (for prop:input-port) or output port (for prop:input-port), the port is used. Otherwise, an empty string input port is used for prop:input-port, and a port that discards all data is used for prop:output-port.

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

12.1.9 Custom Ports

The make-input-port and make-output-port procedures create custom ports with arbitrary control procedures (much like implementing a device driver). Custom ports are mainly useful to obtain fine control over the action of committing bytes as read or written.

```
(make-input-port  name
                  read-in
                  peek
                  close
                 [get-progress-evt
                  commit
                  get-location
                  count-lines!
                  init-position
                  buffer-mode])     → input-port?
  name : any/c
  read-in : (bytes?
            . -> . (one-of/c nonnegative-exact-integer?
                             eof-object?
                             procedure?
                             evt?))
  peek : (bytes? nonnegative-exact-integer? (or/c evt? false/c)
                 . -> . (one-of/c nonnegative-exact-integer?
                                  eof-object?
                                  procedure?
                                  evt?
                                  false/c))
  close : (-> any)
  get-progress-evt : (or/c (-> evt?) false/c) = #f
  commit :  (or/c (positive-exact-integer? evt? evt? . -> . any)
                  false/c)
         = #f
  get-location :  (or/c
                   (()
                    . ->* .
                    ((or/c positive-exact-integer? false/c)
                     (or/c nonnegative-exact-integer? false/c)
                     (or/c positive-exact-integer? false/c)))
                   false/c)
                = #f
  count-lines! : (-> any) = void
  init-position : positive-exact-integer? = 1
  buffer-mode :  (or/c (case-> ((one-of/c 'block 'none) . -> . any)
                               (-> (one-of/c 'block 'none #f)))
                       false/c)
               = #f
```

Creates an input port, which is immediately open for reading. If *close* procedure has no side effects, then the port need not be explicitly closed. See also `make-input-port/peek-to-read`.

The arguments implement the port as follows:

- *name* — the name for the input port.
- *read-in* — a procedure that takes a single argument: a mutable byte string to receive read bytes. The procedure's result is one of the following: %

  - the number of bytes read, as an exact, non-negative integer;
  - eof;
  - a procedure of arity four (representing a "special" result, as discussed further below) and optionally of arity zero, but a procedure result is allowed only when *peek* is not #f;
  - a pipe input port that supplies bytes to be used as long as the pipe has content (see pipe-content-length) or until *read-in* or *peek* is called again; or
  - a synchronizable event (see §10.2.1 "Events") other than a pipe input port that becomes ready when the read is complete (roughly): the event's value can one of the above three results or another event like itself; in the last case, a reading process loops with sync until it gets a non-event result.

  The *read-in* procedure must not block indefinitely. If no bytes are immediately available for reading, the *read-in* must return 0 or an event, and preferably an event (to avoid busy waits). The *read-in* should not return 0 (or an event whose value is 0) when data is available in the port, otherwise polling the port will behave incorrectly. An event result from an event can also break polling.

  If the result of a *read-in* call is not one of the above values, the exn:fail:contract exception is raised. If a returned integer is larger than the supplied byte string's length, the exn:fail:contract exception is raised. If *peek* is #f and a procedure for a special result is returned, the exn:fail:contract exception is raised.

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

  A port's reading procedure may be called in multiple threads simultaneously (if the port is accessible in multiple threads), and the port is responsible for its own internal synchronization. Note that improper implementation of such synchronization mechanisms might cause a non-blocking read procedure to block indefinitely.

  If the result is a pipe input port, then previous *get-progress-evt* calls whose event is not yet ready must have been the pipe input port itself. Furthermore, *get-progress-evt* must continue to return the pipe as long as it contains data, or until the *read-in* or peek-in procedure is called again (instead of using the pipe, for whatever reason). If *read-in* or peek-in is called, any previously associated pipe (as returned by a previous call) will have been disassociated from the port, and is not in use by any other thread as a result of the previous association.

If *peek*, *get-progress-evt*, and *commit* are all provided and non-`#f`, then the following is an acceptable implementation of *read-in*:

```
(lambda (bstr)
  (let* ([progress-evt (get-progress-evt)]
         [v (peek bstr 0 progress-evt)])
    (cond
     [(sync/timeout 0 progress-evt) 0] ; try again
     [(evt? v) (wrap-evt v (lambda (x) 0))] ; sync, try again
     [(and (number? v) (zero? v)) 0] ; try again
     [else
      (if (commit (if (number? v) v 1)
                  progress-evt
                  always-evt)
          v      ; got a result
          0)])))  ; try again
```

An implementor may choose not to implement the *peek*, *get-progress-evt*, and *commit* procedures, however, and even an implementor who does supply them may provide a different *read-in* that uses a fast path for non-blocking reads.

- *peek* — either `#f` or a procedure that takes three arguments:
  - a mutable byte string to receive peeked bytes;
  - a non-negative number of bytes (or specials) to skip before peeking; and
  - either `#f` or a progress event produced by *get-progress-evt*.

  The results and conventions for *peek* are mostly the same as for *read-in*. The main difference is in the handling of the progress event, if it is not `#f`. If the given progress event becomes ready, the *peek* must abort any skip attempts and not peek any values. In particular, *peek* must not peek any values if the progress event is initially ready.

  Unlike *read-in*, *peek* should produce `#f` (or an event whose value is `#f`) if no bytes were peeked because the progress event became ready. Like *read-in*, a `0` result indicates that another attempt is likely to succeed, so `0` is inappropriate when the progress event is ready. Also like *read-in*, *peek* must not block indefinitely.

  The skip count provided to *peek* is a number of bytes (or specials) that must remain present in the port—in addition to the peek results—when the peek results are reported. If a progress event is supplied, then the peek is effectively canceled when another process reads data before the given number can be skipped. If a progress event is not supplied and data is read, then the peek must effectively restart with the original skip count.

  The system does not check that multiple peeks return consistent results, or that peeking and reading produce consistent results.

  If *peek* is `#f`, then peeking for the port is implemented automatically in terms of reads, but with several limitations. First, the automatic implementation is not thread-safe. Second, the automatic implementation cannot handle special results (non-byte and non-eof), so *read-in* cannot return a procedure for a special when *peek* is #f. Finally, the automatic peek implementation is incompatible with progress events, so if *peek* is #f, then progress-evt and *commit* must be #f. See also make-input-port/peek-to-read, which implements peeking in terms of *read-in* without these constraints.

- *close* — a procedure of zero arguments that is called to close the port. The port is not considered closed until the closing procedure returns. The port's procedures will never be used again via the port after it is closed. However, the closing procedure can be called simultaneously in multiple threads (if the port is accessible in multiple threads), and it may be called during a call to the other procedures in another thread; in the latter case, any outstanding reads and peeks should be terminated with an error.

- *get-progress-evt* — either #f (the default), or a procedure that takes no arguments and returns an event. The event must become ready only after data is next read from the port or the port is closed. After the event becomes ready, it must remain so. See the description of *read-in* for information about the allowed results of this function when *read-in* returns a pipe input port. See also semaphore-peek-evt, which is sometimes useful for implementing *get-progress-evt*.

  If *get-progress-evt* is #f, then port-provides-progress-evts? applied to the port will produce #f, and the port will not be a valid argument to port-progress-evt.

- *commit* — either #f (the default), or a procedure that takes three arguments:

  - an exact, positive integer k_r;
  - a progress event produced by *get-progress-evt*;
  - an event, *done*, that is either a channel-put event, channel, semaphore, semaphore-peek event, always event, or never event.

  A *commit* corresponds to removing data from the stream that was previously peeked, but only if no other process removed data first. (The removed data does not need to be reported, because it has been peeked already.) More precisely, assuming that k_p bytes, specials, and mid-stream eofs have been previously peeked or skipped at the start of the port's stream, *commit* must satisfy the following constraints:

  - It must return only when the commit is complete or when the given progress event becomes ready.
  - It must commit only if k_p is positive.
  - If it commits, then it must do so with either k_r items or k_p items, whichever is smaller, and only if k_p is positive.
  - It must never choose *done* in a synchronization after the given progress event is ready, or after *done* has been synchronized once.
  - It must not treat any data as read from the port unless *done* is chosen in a synchronization.

- It must not block indefinitely if *done* is ready; it must return soon after the read completes or soon after the given progress event is ready, whichever is first.
- It can report an error by raising an exception, but only if no data is committed. In other words, no data should be lost due to an exception, including a break exception.
- It must return a true value if data is committed, `#f` otherwise. When it returns a value, the given progress event must be ready (perhaps because data was just committed).
- It must raise an exception if no data (including `eof`) has been peeked from the beginning of the port's stream, or if it would have to block indefinitely to wait for the given progress event to become ready.

A call to *commit* is `parameterize-break`ed to disable breaks.

- *get-location* — either `#f` (the default), or a procedure that takes no arguments and returns three values: the line number for the next item in the port's stream (a positive number or `#f`), the column number for the next item in the port's stream (a non-negative number or `#f`), and the position for the next item in the port's stream (a positive number or `#f`). See also §12.1.4 "Counting Positions, Lines, and Columns".

  This procedure is only called if line counting is enabled for the port via `port-count-lines!` (in which case *count-lines!* is called). The `read`, `read-syntax`, `read-honu`, and `read-honu-syntax` procedures assume that reading a non-whitespace character increments the column and position by one.

- *count-lines!* — a procedure of no arguments that is called if and when line counting is enabled for the port. The default procedure is `void`.

- *init-position* — an exact, positive integer that determines the position of the port's first item, used when line counting is *not* enabled for the port. The default is `1`.

- *buffer-mode* — either `#f` (the default) or a procedure that accepts zero or one arguments. If *buffer-mode* is `#f`, then the resulting port does not support a buffer-mode setting. Otherwise, the procedure is called with one symbol argument (`'block` or `'none`) to set the buffer mode, and it is called with zero arguments to get the current buffer mode. In the latter case, the result must be `'block`, `'none`, or `#f` (unknown). See §12.1.3 "Port Buffers and Positions" for more information on buffer modes.

When *read-in* or *peek* (or an event produced by one of these) returns a procedure, and the procedure is used to obtain a non-byte result. (This non-byte result is *not* intended to return a character or `eof`; in particular, `read-char` raises an exception if it encounters a special-result procedure, even if the procedure produces a byte.) A special-result procedure must accept four arguments, and it can optionally accept zero arguments:

- When the special read is triggered by `read-syntax`, `read-honu-syntax`, or `read-syntax/recursive`, the procedure is passed four arguments that represent a source location.

- When the special read is triggered by `read`, `read-honu`, `read-byte-or-special`, `read-char-or-special`, `peek-byte-or-special`, or `peek-char-or-special`, the procedure is passed no arguments if it accepts zero arguments, otherwise it is passed four arguments that are all `#f`.

The special-value procedure can return an arbitrary value, and it will be called zero or one times (not necessarily before further reads or peeks from the port). See §12.9.2 "Reader-Extension Procedures" for more details on the procedure's result.

If *read-in* or *peek* returns a special procedure when called by any reading procedure other than `read`, `read-syntax`, `read-honu`, `read-honu-syntax`, `read-char-or-special`, `peek-char-or-special`, `read-byte-or-special`, or `peek-byte-or-special`, then the `exn:fail:contract` exception is raised.

Examples:

```
; A port with no input...
; Easy: (open-input-bytes #"")
; Hard:
> (define /dev/null-in
    (make-input-port 'null
                     (lambda (s) eof)
                     (lambda (skip s progress-evt) eof)
                     void
                     (lambda () never-evt)
                     (lambda (k progress-evt done-evt)
                       (error "no successful peeks!"))))
> (read-char /dev/null-in)
#<eof>
> (peek-char /dev/null-in)
#<eof>
> (read-byte-or-special /dev/null-in)
#<eof>
> (peek-byte-or-special /dev/null-in 100)
#<eof>
; A port that produces a stream of 1s:
> (define infinite-ones
    (make-input-port
     'ones
     (lambda (s)
       (bytes-set! s 0 (char->integer #\1)) 1)
     #f
     void))
> (read-string 5 infinite-ones)
"11111"
; But we can't peek ahead arbitrarily far, because the
; automatic peek must record the skipped bytes, so
```

```
; we'd run out of memory.
; An infinite stream of 1s with a specific peek procedure:
> (define infinite-ones
    (let ([one! (lambda (s)
                  (bytes-set! s 0 (char->integer #\1)) 1)])
      (make-input-port
       'ones
       one!
       (lambda (s skip progress-evt) (one! s))
       void)))
> (read-string 5 infinite-ones)
"11111"
; Now we can peek ahead arbitrarily far:
> (peek-string 5 (expt 2 5000) infinite-ones)
"11111"
; The port doesn't supply procedures to implement progress events:
> (port-provides-progress-evts? infinite-ones)
#f
> (port-progress-evt infinite-ones)
port-progress-evt: port does not provide progress evts:
#<input-port:ones>
; Non-byte port results:
> (define infinite-voids
    (make-input-port
     'voids
     (lambda (s) (lambda args 'void))
     (lambda (skip s evt) (lambda args 'void))
     void))
> (read-char infinite-voids)
read-char: non-character in an unsupported context, from
port: #<input-port:voids>
> (read-char-or-special infinite-voids)
void
; This port produces 0, 1, 2, 0, 1, 2, etc., but it is not
; thread-safe, because multiple threads might read and change n.
> (define mod3-cycle/one-thread
    (let* ([n 2]
           [mod! (lambda (s delta)
                   (bytes-set! s 0 (+ 48 (modulo (+ n delta) 3)))
                   1)])
      (make-input-port
       'mod3-cycle/not-thread-safe
       (lambda (s)
         (set! n (modulo (add1 n) 3))
         (mod! s 0))
       (lambda (s skip evt)
```

```
          (mod! s skip))
        void)))
> (read-string 5 mod3-cycle/one-thread)
"01201"
> (peek-string 5 (expt 2 5000) mod3-cycle/one-thread)
"20120"
; Same thing, but thread-safe and kill-safe, and with progress
; events. Only the server thread touches the stateful part
; directly. (See the output port examples for a simpler thread-safe
; example, but this one is more general.)
> (define (make-mod3-cycle)
    (define read-req-ch (make-channel))
    (define peek-req-ch (make-channel))
    (define progress-req-ch (make-channel))
    (define commit-req-ch (make-channel))
    (define close-req-ch (make-channel))
    (define closed? #f)
    (define n 0)
    (define progress-sema #f)
    (define (mod! s delta)
      (bytes-set! s 0 (+ 48 (modulo (+ n delta) 3)))
      1)
    ; ----------------------------------------
    ; The server has a list of outstanding commit requests,
    ; and it also must service each port operation (read,
    ; progress-evt, etc.)
    (define (serve commit-reqs response-evts)
      (apply
       sync
       (handle-evt read-req-ch
                   (handle-read commit-reqs response-evts))
       (handle-evt progress-req-ch
                   (handle-progress commit-reqs response-evts))
       (handle-evt commit-req-ch
                   (add-commit commit-reqs response-evts))
       (handle-evt close-req-ch
                   (handle-close commit-reqs response-evts))
       (append
        (map (make-handle-response commit-reqs response-evts)
             response-evts)
        (map (make-handle-commit commit-reqs response-evts)
             commit-reqs))))
    ; Read/peek request: fill in the string and commit
    (define ((handle-read commit-reqs response-evts) r)
      (let ([s (car r)]
            [skip (cadr r)]
```

```
        [ch (caddr r)]
        [nack (cadddr r)]
        [evt (car (cddddr r))]
        [peek? (cdr (cddddr r))]])
    (let ([fail? (and evt
                      (sync/timeout 0 evt))])
      (unless (or closed? fail?)
        (mod! s skip)
        (unless peek?
          (commit! 1)))
      ; Add an event to respond:
      (serve commit-reqs
             (cons (choice-evt
                    nack
                    (channel-put-evt ch (if closed?
                                            0
                                            (if fail? #f 1))))
                   response-evts)))))
; Progress request: send a peek evt for the current
; progress-sema
(define ((handle-progress commit-reqs response-evts) r)
  (let ([ch (car r)]
        [nack (cdr r)])
    (unless progress-sema
      (set! progress-sema (make-semaphore (if closed? 1 0))))
    ; Add an event to respond:
    (serve commit-reqs
           (cons (choice-evt
                  nack
                  (channel-put-evt
                   ch
                   (semaphore-peek-evt progress-sema)))
                 response-evts))))
; Commit request: add the request to the list
(define ((add-commit commit-reqs response-evts) r)
  (serve (cons r commit-reqs) response-evts))
; Commit handling: watch out for progress, in which case
; the response is a commit failure; otherwise, try
; to sync for a commit. In either event, remove the
; request from the list
(define ((make-handle-commit commit-reqs response-evts) r)
  (let ([k (car r)]
        [progress-evt (cadr r)]
        [done-evt (caddr r)]
        [ch (cadddr r)]
        [nack (cddddr r)])
```

```
    ; Note: we don't check that k is $\leq$ the sum of
    ; previous peeks, because the entire stream is actually
    ; known, but we could send an exception in that case.
    (choice-evt
     (handle-evt progress-evt
                 (lambda (x)
                   (sync nack (channel-put-evt ch #f))
                   (serve (remq r commit-reqs) response-evts)))
     ; Only create an event to satisfy done-evt if progress-evt
     ; isn't already ready.
     ; Afterward, if progress-evt becomes ready, then this
     ; event-making function will be called again, because
     ; the server controls all posts to progress-evt.
     (if (sync/timeout 0 progress-evt)
         never-evt
         (handle-evt done-evt
                     (lambda (v)
                       (commit! k)
                       (sync nack (channel-put-evt ch #t))
                       (serve (remq r commit-reqs)
                              response-evts)))))))
; Response handling: as soon as the respondee listens,
; remove the response
(define ((make-handle-response commit-reqs response-evts) evt)
  (handle-evt evt
              (lambda (x)
                (serve commit-reqs
                       (remq evt response-evts)))))
; Close handling: post the progress sema, if any, and set
; the closed? flag
(define ((handle-close commit-reqs response-evts) r)
  (let ([ch (car r)]
        [nack (cdr r)])
    (set! closed? #t)
    (when progress-sema
      (semaphore-post progress-sema))
    (serve commit-reqs
           (cons (choice-evt nack
                             (channel-put-evt ch (void)))
                 response-evts))))
; Helper for reads and post-peek commits:
(define (commit! k)
  (when progress-sema
    (semaphore-post progress-sema)
    (set! progress-sema #f))
  (set! n (+ n k)))
```

```
  ; Start the server thread:
  (define server-thread (thread (lambda () (serve null null))))
  ; ----------------------------------------
  ; Client-side helpers:
  (define (req-evt f)
    (nack-guard-evt
     (lambda (nack)
       ; Be sure that the server thread is running:
       (thread-resume server-thread (current-thread))
       ; Create a channel to hold the reply:
       (let ([ch (make-channel)])
         (f ch nack)
         ch))))
  (define (read-or-peek-evt s skip evt peek?)
    (req-evt (lambda (ch nack)
               (channel-put read-req-ch
                            (list* s skip ch nack evt peek?)))))
  ; Make the port:
  (make-input-port 'mod3-cycle
                   ; Each handler for the port just sends
                   ; a request to the server
                   (lambda (s) (read-or-peek-evt s 0 #f #f))
                   (lambda (s skip evt)
                     (read-or-peek-evt s skip evt #t))
                   (lambda () ; close
                     (sync (req-evt
                            (lambda (ch nack)
                              (channel-put progress-req-ch
                                           (list* ch nack))))))
                   (lambda () ; progress-evt
                     (sync (req-evt
                            (lambda (ch nack)
                              (channel-put progress-req-ch
                                           (list* ch nack))))))
                   (lambda (k progress-evt done-evt)  ; commit
                     (sync (req-evt
                            (lambda (ch nack)
                              (channel-put
                               commit-req-ch
                               (list* k progress-evt done-evt ch
                                      nack))))))))
> (define mod3-cycle (make-mod3-cycle))
> (let ([result1 #f]
        [result2 #f])
    (let ([t1 (thread
               (lambda ()
```

```
                (set! result1 (read-string 5 mod3-cycle))))]
          [t2 (thread
               (lambda ()
                 (set! result2 (read-string 5 mod3-cycle))))])
      (thread-wait t1)
      (thread-wait t2)
      (string-append result1 "," result2)))
"12012,00120"
> (define s (make-bytes 1))
> (define progress-evt (port-progress-evt mod3-cycle))
> (peek-bytes-avail! s 0 progress-evt mod3-cycle)
1
> s
#"1"
> (port-commit-peeked 1 progress-evt (make-semaphore 1)
                      mod3-cycle)
#t
> (sync/timeout 0 progress-evt)
#<progress-evt>
> (peek-bytes-avail! s 0 progress-evt mod3-cycle)
0
> (port-commit-peeked 1 progress-evt (make-semaphore 1)
                      mod3-cycle)
#f
> (close-input-port mod3-cycle)
```

```
(make-output-port  name
                   evt
                   write-out
                   close
                  [write-out-special
                   get-write-evt
                   get-write-special-evt
                   get-location
                   count-lines!
                   init-position]
                   buffer-mode)          → output-port?
  name : any/c
  evt : evt?
```

```
write-out : (bytes? nonnegative-exact-integer?
                    nonnegative-exact-integer?
                    boolean?
                    boolean?
                    . -> .
                    (or/c nonnegative-exact-integer?
                          false/c
                          evt?))
close : (-> any)
write-out-special : (or/c (any/c boolean? boolean? = #f
                                 . -> .
                                 (or/c any/c
                                       #f
                                       evt?))
                          false/c)
get-write-evt : (or/c                                = #f
                 (bytes? nonnegative-exact-integer?
                         nonnegative-exact-integer?
                         . -> .
                         evt?)
                 false/c)
get-write-special-evt : (or/c                 = #f
                         (any/c . -> . evt?)
                         false/c)
get-location :  (or/c
                 (()
                  . ->* .
                  ((or/c positive-exact-integer? false/c)
                   (or/c nonnegative-exact-integer? false/c)
                   (or/c positive-exact-integer? false/c)))
                 false/c)
              = #f
count-lines! : (-> any) = void
init-position : positive-exact-integer? = 1
buffer-mode : (or/c (case->
                     ((one-of/c 'block 'line 'none) . -> . any)
                     (-> (one-of/c 'block 'line 'none #f)))
                    false/c)
```

Creates an output port, which is immediately open for writing. If *close* procedure has no side effects, then the port need not be explicitly closed. The port can buffer data within its *write-out* and *write-out-special* procedures.

- *name* — the name for the output port.
- *evt* — a synchronization event (see §10.2.1 "Events"; e.g., a semaphore or another

port). The event is used in place of the output port when the port is supplied to synchronization procedures like sync. Thus, the event should be unblocked when the port is ready for writing at least one byte without blocking, or ready to make progress in flushing an internal buffer without blocking. The event must not unblock unless the port is ready for writing; otherwise, the guarantees of sync will be broken for the output port. Use always-evt if writes to the port always succeed without blocking.

- *write-out* — a procedure of five arguments:
  - an immutable byte string containing bytes to write;
  - a non-negative exact integer for a starting offset (inclusive) into the byte string;
  - a non-negative exact integer for an ending offset (exclusive) into the byte string;
  - a boolean; #f indicates that the port is allowed to keep the written bytes in a buffer, and that it is allowed to block indefinitely; #t indicates that the write should not block, and that the port should attempt to flush its buffer and completely write new bytes instead of buffering them;
  - a boolean; #t indicates that if the port blocks for a write, then it should enable breaks while blocking (e.g., using sync/enable-break); this argument is always #f if the fourth argument is #t.

  The procedure returns one of the following:

  - a non-negative exact integer representing the number of bytes written or buffered;
  - #f if no bytes could be written, perhaps because the internal buffer could not be completely flushed;
  - a pipe output port (when buffering is allowed and not when flushing) for buffering bytes as long as the pipe is not full and until *write-out* or *write-out-special* is called; or
  - a synchronizable event (see §10.2.1 "Events") other than a pipe output port that acts like the result of write-bytes-avail-evt to complete the write.

  Since *write-out* can produce an event, an acceptable implementation of *write-out* is to pass its first three arguments to the port's *get-write-evt*. Some port implementors, however, may choose not to provide *get-write-evt* (perhaps because writes cannot be made atomic), or may implement write-proc to enable a fast path for non-blocking writes or to enable buffering.

  From a user's perspective, the difference between buffered and completely written data is (1) buffered data can be lost in the future due to a failed write, and (2) flush-output forces all buffered data to be completely written. Under no circumstances is buffering required.

  If the start and end indices are the same, then the fourth argument to *write-out* will be #f, and the write request is actually a flush request for the port's buffer (if any), and the result should be 0 for a successful flush (or if there is no buffer).

The result should never be `0` if the start and end indices are different, otherwise the `exn:fail:contract` exception is raised. Similarly, the `exn:fail:contract` exception is raised if *`write-out`* returns a pipe output port when buffering is disallowed or when it is called for flushing. If a returned integer is larger than the supplied byte-string range, the `exn:fail:contract` exception is raised.

The `#f` result should be avoided, unless the next write attempt is likely to work. Otherwise, if data cannot be written, return an event instead.

An event returned by *`write-out`* can return `#f` or another event like itself, in contrast to events produced by `write-bytes-avail-evt` or *`get-write-evt`*. A writing process loops with `sync` until it obtains a non-event result.

The *`write-out`* procedure is always called with breaks disabled, independent of whether breaks were enabled when the write was requested by a client of the port. If breaks were enabled for a blocking operation, then the fifth argument to *`write-out`* will be `#t`, which indicates that *`write-out`* should re-enable breaks while blocking.

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

A port's writing procedure may be called in multiple threads simultaneously (if the port is accessible in multiple threads). The port is responsible for its own internal synchronization. Note that improper implementation of such synchronization mechanisms might cause a non-blocking write procedure to block.

- *`close`* — a procedure of zero arguments that is called to close the port. The port is not considered closed until the closing procedure returns. The port's procedures will never be used again via the port after it is closed. However, the closing procedure can be called simultaneously in multiple threads (if the port is accessible in multiple threads), and it may be called during a call to the other procedures in another thread; in the latter case, any outstanding writes or flushes should be terminated immediately with an error.

- *`write-out-special`* — either `#f` (the default), or a procedure to handle `write-special` calls for the port. If `#f`, then the port does not support special output, and `port-writes-special?` will return `#f` when applied to the port.

  If a procedure is supplied, it takes three arguments: the special value to write, a boolean that is `#f` if the procedure can buffer the special value and block indefinitely, and a boolean that is `#t` if the procedure should enable breaks while blocking. The result is one of the following:

  - a non-event true value, which indicates that the special is written;
  - `#f` if the special could not be written, perhaps because an internal buffer could not be completely flushed;
  - a synchronizable event (see §10.2.1 "Events") that acts like the result of *`get-write-special-evt`* to complete the write.

Since `write-out-special` can return an event, passing the first argument to an implementation of `get-write-special-evt` is acceptable as an `write-out-special`.

As for `write-out`, the `#f` result is discouraged, since it can lead to busy waiting. Also as for `write-out`, an event produced by `write-out-special` is allowed to produce `#f` or another event like itself. The `write-out-special` procedure is always called with breaks disabled, independent of whether breaks were enabled when the write was requested by a client of the port.

- `get-write-evt` — either `#f` (the default) or a procedure of three arguments:

  - an immutable byte string containing bytes to write;
  - a non-negative exact integer for a starting offset (inclusive) into the byte string, and
  - a non-negative exact integer for an ending offset (exclusive) into the byte string.

  The result is a synchronizable event (see §10.2.1 "Events") to act as the result of `write-bytes-avail-evt` for the port (i.e., to complete a write or flush), which becomes available only as data is committed to the port's underlying device, and whose result is the number of bytes written.

  If `get-write-evt` is `#f`, then `port-writes-atomic?` will produce `#f` with applied to the port, and the port will not be a valid argument to procedures such as `write-bytes-avail-evt`.

  Otherwise, an event returned by `get-write-evt` must not cause data to be written to the port unless the event is chosen in a synchronization, and it must write to the port if the event is chosen (i.e., the write must appear atomic with respect to the synchronization).

  If the event's result integer is larger than the supplied byte-string range, the `exn:fail:contract` exception is raised by a wrapper on the event. If the start and end indices are the same (i.e., no bytes are to be written), then the event should produce `0` when the buffer is completely flushed. (If the port has no buffer, then it is effectively always flushed.)

  If the event raises an exception, due either to write or commit operations, it must not have committed any new bytes (though it may have committed previously buffered bytes).

  Naturally, a port's events may be used in multiple threads simultaneously (if the port is accessible in multiple threads). The port is responsible for its own internal synchronization.

- `get-write-special-evt` — either `#f` (the default), or a procedure to handle `write-special-evt` calls for the port. This argument must be `#f` if either `write-out-special` or `get-write-evt` is `#f`, and it must be a procedure if both of those arguments are procedures.

If it is a procedure, it takes one argument: the special value to write. The resulting event (with its constraints) is analogous to the result of *get-write-evt*.

If the event raises an exception, due either to write or commit operations, it must not have committed the special value (though it may have committed previously buffered bytes and values).

- *get-location* — either `#f` (the default), or a procedure that takes no arguments and returns three values: the line number for the next item written to the port's stream (a positive number or `#f`), the column number for the next item written to port's stream (a non-negative number or `#f`), and the position for the next item written to port's stream (a positive number or `#f`). See also §12.1.4 "Counting Positions, Lines, and Columns".

  This procedure is only called if line counting is enabled for the port via `port-count-lines!` (in which case *count-lines!* is called).

- *count-lines!* — a procedure of no arguments that is called if and when line counting is enabled for the port. The default procedure is `void`.

- *init-position* — an exact, positive integer that determines the position of the port's first output item, used when line counting is *not* enabled for the port. The default is `1`.

- *buffer-mode* — either `#f` (the default) or a procedure that accepts zero or one arguments. If *buffer-mode* is `#f`, then the resulting port does not support a buffer-mode setting. Otherwise, the procedure is called with one symbol argument (`'block`, `'line`, or `'none`) to set the buffer mode, and it is called with zero arguments to get the current buffer mode. In the latter case, the result must be `'block`, `'line`, `'none`, or `#f` (unknown). See §12.1.3 "Port Buffers and Positions" for more information on buffer modes.

Examples:

```
; A port that writes anything to nowhere:
> (define /dev/null-out
    (make-output-port
     'null
     always-evt
     (lambda (s start end non-block? breakable?) (- end start))
     void
     (lambda (special non-block? breakable?) #t)
     (lambda (s start end) (wrap-evt
                            always-evt
                            (lambda (x)
                              (- end start))))
     (lambda (special) always-evt)))
> (display "hello" /dev/null-out)
> (write-bytes-avail #"hello" /dev/null-out)
5
```

```
> (write-special 'hello /dev/null-out)
#t
> (sync (write-bytes-avail-evt #"hello" /dev/null-out))
5
; A port that accumulates bytes as characters in a list,
; but not in a thread-safe way:
> (define accum-list null)
> (define accumulator/not-thread-safe
    (make-output-port
     'accum/not-thread-safe
     always-evt
     (lambda (s start end non-block? breakable?)
       (set! accum-list
             (append accum-list
                     (map integer->char
                          (bytes->list (subbytes s start end)))))
       (- end start))
     void))
> (display "hello" accumulator/not-thread-safe)
> accum-list
(#\h #\e #\l #\l #\o)
; Same as before, but with simple thread-safety:
> (define accum-list null)
> (define accumulator
    (let* ([lock (make-semaphore 1)]
           [lock-peek-evt (semaphore-peek-evt lock)])
      (make-output-port
       'accum
       lock-peek-evt
       (lambda (s start end non-block? breakable?)
         (if (semaphore-try-wait? lock)
             (begin
               (set! accum-list
                     (append accum-list
                             (map integer->char
                                  (bytes->list
                                   (subbytes s start end)))))
               (semaphore-post lock)
               (- end start))
             ; Cheap strategy: block until the list is unlocked,
             ; then return 0, so we get called again
             (wrap-evt
              lock-peek
              (lambda (x) 0))))
       void)))
> (display "hello" accumulator)
```

```
> accum-list
(#\h #\e #\l #\l #\o)
; A port that transforms data before sending it on
; to another port. Atomic writes exploit the
; underlying port's ability for atomic writes.
> (define (make-latin-1-capitalize port)
    (define (byte-upcase s start end)
      (list->bytes
       (map (lambda (b) (char->integer
                         (char-upcase
                          (integer->char b))))
            (bytes->list (subbytes s start end)))))
    (make-output-port
     'byte-upcase
     ; This port is ready when the original is ready:
     port
     ; Writing procedure:
     (lambda (s start end non-block? breakable?)
       (let ([s (byte-upcase s start end)])
         (if non-block?
             (write-bytes-avail* s port)
             (begin
               (display s port)
               (bytes-length s)))))
     ; Close procedure --- close original port:
     (lambda () (close-output-port port))
     #f
     ; Write event:
     (and (port-writes-atomic? port)
          (lambda (s start end)
            (write-bytes-avail-evt
             (byte-upcase s start end)
             port)))))
> (define orig-port (open-output-string))
> (define cap-port (make-latin-1-capitalize orig-port))
> (display "Hello" cap-port)
> (get-output-string orig-port)
"HELLO"
> (sync (write-bytes-avail-evt #"Bye" cap-port))
3
> (get-output-string orig-port)
"HELLOBYE"
```

12.1.10 More Port Constructors and Events

```
(require scheme/port)
```

The bindings documented in this section are provided by the `scheme/port` and `scheme` libraries, but not `scheme/base`.

[

Creating Ports]Creating Ports

```
(input-port-append close-at-eof? in ...) → input-port?
  close-at-eof? : any/c
  in : input-port?
```

Takes any number of input ports and returns an input port. Reading from the input port draws bytes (and special non-byte values) from the given input ports in order. If *close-at-eof?* is true, then each port is closed when an end-of-file is encountered from the port, or when the result input port is closed. Otherwise, data not read from the returned input port remains available for reading in its original input port.

See also `merge-input`, which interleaves data from multiple input ports as it becomes available.

```
(make-input-port/read-to-peek  name
                               read-in
                               fast-peek
                               close
                              [get-location
                               count-lines!
                               init-position
                               buffer-mode
                               buffering?
                               on-consume])   → input-port?
  name : any/c
  read-in : (bytes?
             . -> . (one-of/c exact-nonnegative-integer?
                              eof-object?
                              procedure?
                              evt?))
```

```
fast-peek : (or/c false/c
                  (bytes? exact-nonnegative-integer?
                   (bytes? exact-nonnegative-integer?
                    . -> . (one-of/c exact-nonnegative-integer?
                                     eof-object?
                                     procedure?
                                     evt?
                                     false/c))
                   . -> . (one-of/c exact-nonnegative-integer?
                                    eof-object?
                                    procedure?
                                    evt?
                                    false/c)))
close : (-> any)
get-location : (or/c
                 (()
                  . ->* .
                  ((or/c positive-exact-integer? false/c)
                   (or/c nonnegative-exact-integer? false/c)
                   (or/c positive-exact-integer? false/c)))
                 false/c)
             = #f
count-lines! : (-> any) = void
init-position : positive-exact-integer? = 1
buffer-mode : (or/c (case-> ((one-of/c 'block 'none) . -> . any)
                            (-> (one-of/c 'block 'none #f)))
                    false/c)
            = #f
buffering? : any/c = #f
on-consume : (or/c ((or/c exact-nonnegative-integer? eof-object?
                          procedure? evt?)
                    . -> . any)
                   false/c)
           = #f
```

Similar to `make-input-port`, but if the given *read-in* returns an event, the event's value must be `0`. The resulting port's peek operation is implemented automatically (in terms of *read-in*) in a way that can handle special non-byte values. The progress-event and commit operations are also implemented automatically. The resulting port is thread-safe, but not kill-safe (i.e., if a thread is terminated or suspended while using the port, the port may become damaged).

The *read-in*, *close*, `get-lcoation`, *count-lines!*, *init-position*, and *buffer-mode* procedures are the same as for `make-input-port`.

The *fast-peek* argument can be either `#f` or a procedure of three arguments: a byte string to receive a peek, a skip count, and a procedure of two arguments. The *fast-peek* procedure can either implement the requested peek, or it can dispatch to its third argument to implement the peek. The *fast-peek* is not used when a peek request has an associated progress event.

The *buffering?* argument determines whether *read-in* can be called to read more characters than are immediately demanded by the user of the new port. If `buffer` mode is not `#f`, then *buffering?* determines the initial buffer mode, and *buffering?* is enabled after a buffering change only if the new mode is `'block`.

If `on-consumed` is not `#f`, it is called when data is read from the port, as opposed to merely peeked. The argument to *on-consume* is the result value of the port's reading procedure, so it can be an integer or any result from *read-in*.

```
(make-limited-input-port in
                         limit
                        [close-orig?]) → input-port?
  in : input-port?
  limit : exact-nonnegative-integer?
  close-orig? : any/c = #t
```

Returns a port whose content is drawn from *in*, but where an end-of-file is reported after *limit* bytes (and non-byte special values) are read. If *close-orig?* is true, then the original port is closed if the returned port is closed.

Bytes are consumed from *in* only when they are consumed from the returned port. In particular, peeking into the returned port peeks into the original port.

If *in* is used directly while the resulting port is also used, then the *limit* bytes provided by the port need not be contiguous parts of the original port's stream.

```
(make-pipe-with-specials [limit
                          in-name
                          out-name]) → input-port? output-port?
  limit : exact-nonnegative-integer? = #f
  in-name : any/c = 'pipe
  out-name : any/c = 'pipe
```

Returns two ports: an input port and an output port. The ports behave like those returned by `make-pipe`, except that the ports support non-byte values written with procedures such as `write-special` and read with procedures such as `get-byte-or-special`.

The *limit* argument determines the maximum capacity of the pipe in bytes, but this limit is disabled if special values are written to the pipe before *limit* is reached. The limit is re-enabled after the special value is read from the pipe.

The optional *in-name* and *out-name* arguments determine the names of the result ports.

```
(merge-input a-in b-in [buffer-limit]) → input-port?
  a-in : input-port?
  b-in : input-port?
  buffer-limit : (or/c exact-nonnegative-integer? false/c)
               = 4096
```

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 an end-of-file has been read from an original port, it no longer contributes characters to the new port. After an end-of-file has been read from both original ports, the new port returns end-of-file. Closing the merged port does not close the original ports.

The optional *buffer-limit* argument limits the number of bytes to be buffered from *a-in* and *b-in*, so that the merge process does not advance arbitrarily beyond the rate of consumption of the merged data. A `#f` value disables the limit. As for `make-pipe-with-specials`, *buffer-limit* does not apply when a special value is produced by one of the input ports before the limit is reached.

See also `input-port-append`, which concatenates input streams instead of interleaving them.

```
(open-output-nowhere [name special-ok?]) → output-port?
  name : any/c = 'nowhere
  special-ok? : any/c = #t
```

Creates and returns an output port that discards all output sent to it (without blocking). The *name* argument is used as the port's name. If the *special-ok?* argument is true, then the resulting port supports `write-special`, otherwise it does not.

```
(peeking-input-port in [name skip]) → input-port
  in : input-port?
  name : any/c = (object-name in)
  skip : exact-nonnegative-integer? = 0
```

Returns an input port whose content is determined by peeking into *in*. In other words, the resulting port contains an internal skip count, and each read of the port peeks into *in* with the internal skip count, and then increments the skip count according to the amount of data successfully peeked.

The optional *name* argument is the name of the resulting port. The *skip* argument is the port initial skip count, and it defaults to `0`.

```
(reencode-input-port  in
                      encoding
                      error-bytes
                     [close?
                      name
                      convert-newlines?
                      enc-error])        → input-port?
  in : input-port?
  encoding : string?
  error-bytes : (or/c false/c bytes?)
  close? : any/c = #t
  name : any/c = (object-name in)
  convert-newlines? : any/c = #f
  enc-error : (string? input-port? . -> . any)
            = (lambda (msg port) (error ...))
```

Produces an input port that draws bytes from `in`, but converts the byte stream using `(bytes-open-converter encoding-str "UTF-8")`. In addition, if `convert-newlines?` is true, then decoded sequences that correspond to UTF-8 encodings of `"\r\n"`, `"\r\u0085"`, `"\r"`, `"\u0085"`, and `"\u2028"` are all converted to the UTF-8 encoding of `"\n"`.

If `error-bytes` is provided and not `#f`, then the given byte sequence is used in place of bytes from `in` that trigger conversion errors. Otherwise, if a conversion is encountered, `enc-error` is called, which must raise an exception.

If `close?` is true, then closing the result input port also closes `in`. The `name` argument is used as the name of the result input port.

In non-buffered mode, the resulting input port attempts to draw bytes from `in` only as needed to satisfy requests. Toward that end, the input port assumes that at least *n* bytes must be read to satisfy a request for *n* bytes. (This is true even if the port has already drawn some bytes, as long as those bytes form an incomplete encoding sequence.)

```
(reencode-output-port  out
                       encoding
                       error-bytes
                      [close?
                       name
                       newline-bytes
                       enc-error])    → output-port?
  out : output-port?
  encoding : string?
  error-bytes : (or/c false/c bytes?)
  close? : any/c = #t
  name : any/c = (object-name out)
```

```
newline-bytes : (or/c false/c bytes?) = #f
enc-error : (string? output-port? . -> . any)
          = (lambda (msg port) (error ...))
```

Produces an output port that directs bytes to *out*, but converts its byte stream using (bytes-open-converter "UTF-8" encoding-str). In addition, if *newline-bytes* is not #f, then byets written to the port that are the UTF-8 encoding of "\n" are first converted to *newline-bytes* (before applying the convert from UTF-8 to encoding-str).

If *error-bytes* is provided and not #f, then the given byte sequence is used in place of bytes send to the output port that trigger conversion errors. Otherwise, *enc-error* is called, which must raise an exception.

If *close?* is true, then closing the result output port also closes *out*. The *name* argument is used as the name of the result output port.

The resulting port supports buffering, and the initial buffer mode is (or (file-stream-buffer-mode *out*) 'block). In 'block mode, the port's buffer is flushed only when it is full or a flush is requested explicitly. In 'line mode, the buffer is flushed whenever a newline or carriage-return byte is written to the port. In 'none mode, the port's buffer is flushed after every write. Implicit flushes for 'line or 'none leave bytes in the buffer when they are part of an incomplete encoding sequence.

The resulting output port does not support atomic writes. An explicit flush or special-write to the output port can hang if the most recently written bytes form an incomplete encoding sequence.

```
(relocate-input-port in
                     line
                     column
                     position
                    [close?])  → input-port?
  in : input-port?
  line : (or/c exact-positive-integer? false/c)
  column : (or/c exact-nonnegative-integer? false/c)
  position : exact-positive-integer?
  close? : any/c = #t
```

Produces an input port that is equivalent to *in* except in how it reports location information. The resulting port's content starts with the remaining content of *in*, and it starts at the given line, column, and position. A #f for the line or column means that the line and column will always be reported as #f.

The *line* and *column* values are used only if line counting is enabled for *in* and for the resulting port, typically through port-count-lines!. The *column* value determines the column for the first line (i.e., the one numbered *line*), and later lines start at column 0. The given *position* is used even if line counting is not enabled.

When line counting is on for the resulting port, reading from *in* instead of the resulting port increments location reports from the resulting port. Otherwise, the resulting port's position does not increment when data is read from *in*.

If *close?* is true, then closing the resulting port also closes *in*. If *close?* is `#f`, then closing the resulting port does not close *in*.

```
(relocate-output-port out
                      line
                      column
                      position
                     [close?]) → output-port?
  out : output-port?
  line : (or/c exact-positive-integer? false/c)
  column : (or/c exact-nonnegative-integer? false/c)
  position : exact-positive-integer?
  close? : any/c = #t
```

Like `relocate-input-port`, but for output ports.

```
(transplant-input-port in
                       get-location
                       init-pos
                      [close?
                       count-lines!]) → input-port?
  in : input-port?
  get-location : (or/c
                  (()
                   . ->* .
                   ((or/c exact-positive-integer? false/c)
                   (or/c exact-nonnegative-integer? false/c)
                   (or/c exact-positive-integer? false/c)))
                 false/c)
  init-pos : (-> exact-positive-integer?)
  close? : any/c = #t
  count-lines! : (-> any) = void
```

Like `relocate-input-port`, except that arbitrary position information can be produced (when line counting is enabled) via *get-location*. If *get-location* is `#f`, then the port counts lines in the usual way starting from *init-pos*, independent of locations reported by *in*.

If *count-lines!* is supplied, it is called when line counting is enabled for the resulting port. The default is void.

```
(transplant-output-port  in
                         get-location
                         init-pos
                        [close?
                         count-lines!]) → output-port?
  in : input-port?
  get-location : (or/c
                  (()
                   . ->* .
                   ((or/c exact-positive-integer? false/c)
                    (or/c exact-nonnegative-integer? false/c)
                    (or/c exact-positive-integer? false/c)))
                 false/c)
  init-pos : (-> exact-positive-integer?)
  close? : any/c = #t
  count-lines! : (-> any) = void
```

Like transplant-input-port, but for output ports.

[

Port Events]Port Events

```
(eof-evt in) → evt?
  in : input-port?
```

Returns a synchronizable event is that is ready when in produces an eof. If in produces a mid-stream eof, the eof is consumed by the event only if the event is chosen in a synchronization.}

```
(read-bytes-evt k in) → evt?
  k : exact-nonnegative-integer?
  in : input-port?
```

Returns a synchronizable event is that is ready when *k* bytes can be read from *in*, or when an end-of-file is encountered in *in*. If *k* is 0, then the event is ready immediately with "". For non-zero *k*, if no bytes are available before an end-of-file, the event's result is eof. Otherwise the event's result is a byte string of up to *k* bytes, which contains as many bytes as are available (up to *k*) before an available end-of-file. (The result is a byte string on less than *k* bytes only when an end-of-file is encountered.)

Bytes are read from the port if and only if the event is chosen in a synchronization, and the returned bytes always represent contiguous bytes in the port's stream.

The event can be synchronized multiple times—event concurrently—and each synchronization corresponds to a distinct read request.

The *in* must support progress events, and it must not produce a special non-byte value during the read attempt.

```
(read-bytes!-evt bstr in) → evt?
  bstr : (and/c bytes? (not/c immutable?))
  in : input-port?
```

Like `read-bytes-evt`, except that the read bytes are placed into `bstr`, and the number of bytes to read corresponds to `(bytes-length bstr)`. The event's result is either `eof` or the number of read bytes.

The `bstr` may be mutated any time after the first synchronization attempt on the event. If the event is not synchronized multiple times concurrently, `bstr-bytes` is never mutated by the event after it is chosen in a synchronization (no matter how many synchronization attempts preceded the choice). Thus, the event may be sensibly used multiple times until a successful choice, but should not be used in multiple concurrent synchronizations.}

```
(read-bytes-avail!-evt bstr in) → evt?
  bstr : (and/c bytes? (not/c immutable?))
  in : input-port?
```

Like `read-bytes!-evt`, except that the event reads only as many bytes as are immediately available, after at least one byte or one `eof` becomes available.

```
(read-string-evt k in) → evt?
  k : exact-nonnegative-integer?
  in : input-port?
```

Like `read-bytes-evt`, but for character strings instead of byte strings.

```
(read-string!-evt str in) → evt?
  str : (and/c string? (not/c immutable?))
  in : input-port?
```

Like `read-bytes!-evt`, but for a character string instead of a byte string.

```
(read-line-evt in mode) → evt?
```

```
  in : input-port?
  mode : (one-of 'linefeed 'return 'return-linefeed 'any 'any-one)
```

Returns a synchronizable event that is ready when a line of characters or end-of-file can be read from `inport`. The meaning of *`mode`* is the same as for `read-line`. The event result is the read line of characters (not including the line separator).

A line is read from the port if and only if the event is chosen in a synchronization, and the returned line always represents contiguous bytes in the port's stream.

```
(read-bytes-line-evt in mode) → evt?
  in : input-port?
  mode : (one-of 'linefeed 'return 'return-linefeed 'any 'any-one)
```

Like `read-line`, but returns a byte string instead of a string.

```
(peek-bytes-evt k skip progress in) → evt?
  k : exact-nonnegative-integer?
  skip : exact-nonnegative-integer?
  progress : evt?
  in : input-port?
(peek-bytes!-evt bstr skip progress in) → evt?
  bstr : (and/c bytes? (not/c immutable?))
  skip : exact-nonnegative-integer?
  progress : (or/c evt? false/c)
  in : input-port?
(peek-bytes-avail!-evt bstr skip progress in) → evt?
  bstr : (and/c bytes? (not/c immutable?))
  skip : exact-nonnegative-integer?
  progress : (or/c evt? false/c)
  in : input-port?
(peek-string-evt k in) → evt?
  k : exact-nonnegative-integer?
  in : input-port?
(peek-string!-evt str in) → evt?
  str : (and/c string? (not/c immutable?))
  in : input-port?
```

Like the `read-...-evt` functions, but for peeking. The *`skip`* argument indicates the number of bytes to skip, and *`progress`* indicates an event that effectively cancels the peek (so that the event never becomes ready). The *`progress`* argument can be `#f`, in which case the event is never cancelled.

```
(regexp-match-evt pattern in) → any
```

```
pattern : (or/c string? bytes? regexp? byte-regexp?)
in : input-port?
```

Returns a synchronizable event that is ready when `pattern` matches the stream of bytes/characters from `in`; see also `regexp-match`. The event's value is the result of the match, in the same form as the result of `regexp-match`.

If `pattern` does not require a start-of-stream match, then bytes skipped to complete the match are read and discarded when the event is chosen in a synchronization.

Bytes are read from the port if and only if the event is chosen in a synchronization, and the returned match always represents contiguous bytes in the port's stream. If not-yet-available bytes from the port might contribute to the match, the event is not ready. Similarly, if `pattern` begins with a start-of-stream `^` and the `pattern` does not initially match, then the event cannot become ready until bytes have been read from the port.

The event can be synchronized multiple times—even concurrently—and each synchronization corresponds to a distinct match request.

The `in` port must support progress events. If `in` returns a special non-byte value during the match attempt, it is treated like `eof`.}

[

Copying Streams]Copying Streams

```
(convert-stream from-encoding
                in
                from-encoding
                out)          → void?
  from-encoding : string?
  in : input-port?
  from-encoding : string?
  out : output-port?
```

Reads data from *in*, converts it using `(bytes-open-converter from-encoding-string to-encoding-string)` and writes the converted bytes to *out*. The `convert-stream` procedure returns after reaching `eof` in *in*.

If opening the converter fails, the `exn:fail` exception is raised. Similarly, if a conversion error occurs at any point while reading *in*, then `exn:fail` exception is raised.

```
(copy-port in out ...+) → void?
  in : input-port?
```

```
out : output-port?
```

Reads data from *in* and writes it back out to *out*, returning when *in* produces eof. The copy is efficient, and it is without significant buffer delays (i.e., a byte that becomes available on *in* is immediately transferred to *out*, even if future reads on *in* must block). If *in* produces a special non-byte value, it is transferred to *out* using write-special.

This function is often called from a "background" thread to continuously pump data from one stream to another.

If multiple *out*s are provided, case data from *in* is written to every *out*. The different *out*s block output to each other, because each block of data read from *in* is written completely to one *out* before moving to the next *out*. The *out*s are written in the provided order, so non-blocking ports (e.g., to a file) should be placed first in the argument list.

12.2 Byte and String Input

```
(read-char [in]) → (or/c character? eof-object?)
  in : input-port? = (current-input-port)
```

Reads a single character from *in*—which may involve reading several bytes to UTF-8-decode them into a character (see §12.1 "Ports"); a minimal number of bytes are read/peeked to perform the decoding. If no bytes are available before an end-of-file, then eof is returned.

```
(read-byte [in]) → (or/c byte? eof-object?)
  in : input-port? = (current-input-port)
```

Reads a single byte from *in*. If no bytes are available before an end-of-file, then eof is returned.

```
(read-line [in mode]) → (or/c string? eof-object?)
  in : input-port? = (current-input-port)
  mode : (one-of 'linefeed 'return 'return-linefeed 'any 'any-one)
       = 'linefeed
```

Returns a string containing the next line of bytes from *in*.

Characters are read from *in* 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* argument determines the line separator(s). It must be one of the following symbols:

- `'linefeed` breaks lines on linefeed characters.
- `'return` breaks lines on return characters.
- `'return-linefeed` breaks lines on return-linefeed combinations. If a return character is not followed by a linefeed character, it is included in the result string; similarly, a linefeed that is not preceded by a return is included in the result string.
- `'any` breaks lines on any of a return character, linefeed character, or return-linefeed combination. If a return character is followed by a linefeed character, the two are treated as a combination.
- `'any-one` breaks lines on either a return or linefeed character, without recognizing return-linefeed combinations.

Return and linefeed characters are detected after the conversions that are automatically performed when reading a file in text mode. For example, reading a file in text mode under Windows automatically changes return-linefeed combinations to a linefeed. Thus, when a file is opened in text mode, `'linefeed` is usually the appropriate `read-line` mode.

```
(read-bytes-line [in mode]) → (or/c bytes? eof-object?)
  in : input-port? = (current-input-port)
  mode : (one-of 'linefeed 'return 'return-linefeed 'any 'any-one)
       = 'linefeed
```

Like `read-line`, but reads bytes and produces a byte string.

```
(read-string amt [in]) → (or/c string? eof-object)
  amt : nonnegative-exact-integer?
  in : input-port? = (current-input-port)
```

Returns a string containing the next *amt* characters from *in*.

If *amt* is `0`, then the empty string is returned. Otherwise, if fewer than *amt* characters are available before an end-of-file is encountered, then the returned string will contain only those characters before the end-of-file; that is, the returned string's length will be less than *amt*. (A temporary string of size *amt* is allocated while reading the input, even if the size of the result is less than *amt* characters.) If no characters are available before an end-of-file, then `eof` is returned.

If an error occurs during reading, some characters may be lost; that is, if `read-string` successfully reads some characters before encountering an error, the characters are dropped.

```
(read-bytes amt [in]) → (or/c bytes? eof-object)
  amt : nonnegative-exact-integer?
  in : input-port? = (current-input-port)
```

Like `read-string`, but reads bytes and produces a byte string.

```
(read-string! str [in start-pos end-pos])
 → (or/c positive-exact-integer? eof-object?)
  str : (and/c string? (not/c immutable?))
  in : input-port? = (current-input-port)
  start-pos : nonnegative-exact-integer? = 0
  end-pos : nonnegative-exact-integer? = (string-length str)
```

Reads characters from *in* like `read-string`, but puts them into *str* starting from index *start-pos* (inclusive) up to *end-pos* (exclusive). Like `substring`, the `exn:fail:contract` exception is raised if *start-pos* or *end-pos* is out-of-range for *str*.

If the difference between *start-pos* and *end-pos* is `0`, then `0` is returned and *str* is not modified. If no bytes are available before an end-of-file, then `eof` is returned. Otherwise, the return value is the number of characters read. If m characters are read and m<*end-pos*-*start-pos*, then *str* is not modified at indices *start-pos*+m though *end-pos*.

```
(read-bytes! bstr [in start-pos end-pos])
 → (or/c positive-exact-integer? eof-object?)
  bstr : bytes?
  in : input-port? = (current-input-port)
  start-pos : nonnegative-exact-integer? = 0
  end-pos : nonnegative-exact-integer? = (bytes-length bstr)
```

Like `read-string!`, but reads bytes, puts them into a byte string, and returns the number of bytes read.

```
(read-bytes-avail! bstr [in start-pos end-pos])
 → (or/c positive-exact-integer? eof-object? procedure?)
  bstr : bytes?
  in : input-port? = (current-input-port)
  start-pos : nonnegative-exact-integer? = 0
  end-pos : nonnegative-exact-integer? = (bytes-length bstr)
```

Like `read-bytes!`, but it returns without blocking after reading immediately-available bytes, and it may return a procedure for a “special” result. The `read-bytes-avail!` procedure blocks only if no bytes (or specials) are yet available. Also unlike `read-bytes!`, read-bytes-avail! never drops bytes; if read-bytes-avail! successfully reads some bytes and then encounters an error, it suppresses the error (treating it roughly like an end-of-file) and returns the read bytes. (The error will be triggered by future reads.) If an error is encountered before any bytes have been read, an exception is raised.

When *in* produces a special value, as described in §12.1.9 "Custom Ports", the result is a procedure of four arguments. The four arguments correspond to the location of the special value within the port, as described in §12.1.9 "Custom Ports". If the procedure is called more than once with valid arguments, the exn:fail:contract exception is raised. If read-bytes-avail returns a special-producing procedure, then it does not place characters in *bstr*. Similarly, read-bytes-avail places only as many bytes into *bstr* as are available before a special value in the port's stream.

```
(read-bytes-avail!* bstr
                   [in
                    start-pos
                    end-pos])
 → (or/c nonnegative-exact-integer? eof-object? procedure?)
  bstr : bytes?
  in : input-port? = (current-input-port)
  start-pos : nonnegative-exact-integer? = 0
  end-pos : nonnegative-exact-integer? = (bytes-length bstr)
```

Like read-bytes-avail!, but returns 0 immediately if no bytes (or specials) are available for reading and the end-of-file is not reached.

```
(read-bytes-avail!/enable-break bstr
                               [in
                                start-pos
                                end-pos])
 → (or/c positive-exact-integer? eof-object? procedure?)
  bstr : bytes?
  in : input-port? = (current-input-port)
  start-pos : nonnegative-exact-integer? = 0
  end-pos : nonnegative-exact-integer? = (bytes-length bstr)
```

Like read-bytes-avail!, but breaks are enabled during the read (see also §9.6 "Breaks"). If breaking is disabled when read-bytes-avail!/enable-break is called, and if the exn:break exception is raised as a result of the call, then no bytes will have been read from *in*.

```
(peek-string amt skip-bytes-amt [in]) → (or/c string? eof-object)
  amt : nonnegative-exact-integer?
  skip-bytes-amt : nonnegative-exact-integer?
```

```
  in : input-port? = (current-input-port)
```

Similar to `read-string`, except that the returned characters are preserved in the port for future reads. (More precisely, undecoded bytes are left for future reads.) The *`skip-bytes-amt`* argument indicates a number of bytes (*not* characters) in the input stream to skip before collecting characters to return; thus, in total, the next *`skip-bytes-amt`* bytes plus *`amt`* characters are inspected.

For most kinds of ports, inspecting *`skip-bytes-amt`* bytes and *`amt`* characters requires at least *`skip-bytes-amt`*+*`amt`* bytes of memory overhead associated with the port, at least until the bytes/characters are read. No such overhead is required when peeking into a string port (see §12.1.6 "String Ports"), a pipe port (see §12.1.7 "Pipes"), or a custom port with a specific peek procedure (depending on how the peek procedure is implemented; see §12.1.9 "Custom Ports").

If a port produces `eof` mid-stream, peek skips beyond the `eof` always produce `eof` until the `eof` is read.

```
(peek-bytes amt skip-bytes-amt [in]) → (or/c bytes? eof-object)
  amt : nonnegative-exact-integer?
  skip-bytes-amt : nonnegative-exact-integer?
  in : input-port? = (current-input-port)
```

Like `peek-string`, but peeks bytes and produces a byte string.

```
(peek-string!  str
               skip-bytes-amt
              [in
               start-pos
               end-pos])
 → (or/c positive-exact-integer? eof-object?)
  str : (and/c string? (not/c immutable?))
  skip-bytes-amt : nonnegative-exact-integer?
  in : input-port? = (current-input-port)
  start-pos : nonnegative-exact-integer? = 0
  end-pos : nonnegative-exact-integer? = (string-length str)
```

Like `read-string!`, but for peeking, and with a *`skip-bytes-amt`* argument like `peek-string`.

```
(peek-bytes! bstr
             skip-bytes-amt
            [in
             start-pos
             end-pos])
 → (or/c positive-exact-integer? eof-object?)
  bstr : (and/c bytes? (not/c immutable?))
  skip-bytes-amt : nonnegative-exact-integer?
  in : input-port? = (current-input-port)
  start-pos : nonnegative-exact-integer? = 0
  end-pos : nonnegative-exact-integer? = (bytes-length bstr)
```

Like `peek-string!`, but peeks bytes, puts them into a byte string, and returns the number of bytes read.

```
(peek-bytes-avail! bstr
                   skip-bytes-amt
                  [progress
                   in
                   start-pos
                   end-pos])
 → (or/c nonnegative-exact-integer? eof-object? procedure?)
  bstr : (and/c bytes? (not/c immutable?))
  skip-bytes-amt : nonnegative-exact-integer?
  progress : (or/c evt? false/c) = #f
  in : input-port? = (current-input-port)
  start-pos : nonnegative-exact-integer? = 0
  end-pos : nonnegative-exact-integer? = (bytes-length bstr)
```

Like `read-bytes-avail!`, but for peeking, and with two extra arguments. The *skip-bytes-amt* argument is as in `peek-bytes`. The *progress* argument must be either `#f` or an event produced by `port-progress-evt` for *in*.

To peek, `peek-bytes-avail!` blocks until finding an end-of-file, at least one byte (or special) past the skipped bytes, or until a non-`#f` *progress* becomes ready. Furthermore, if *progress* is ready before bytes are peeked, no bytes are peeked or skipped, and *progress* may cut short the skipping process if it becomes available during the peek attempt.

The result of `peek-bytes-avail!` is `0` only in the case that *progress* becomes ready before bytes are peeked.

```
(peek-bytes-avail!* bstr
                    skip-bytes-amt
                   [progress
                    in
                    start-pos
                    end-pos])
 → (or/c nonnegative-exact-integer? eof-object? procedure?)
  bstr : (and/c bytes? (not/c immutable?))
  skip-bytes-amt : nonnegative-exact-integer?
  progress : (or/c evt? false/c) = #f
  in : input-port? = (current-input-port)
  start-pos : nonnegative-exact-integer? = 0
  end-pos : nonnegative-exact-integer? = (bytes-length bstr)
```

Like `read-bytes-avail!*`, but for peeking, and with *skip-bytes-amt* and *progress* arguments like `peek-bytes-avail!`. Since this procedure never blocks, it may return before even `skip-amt` bytes are available from the port.

```
(peek-bytes-avail!/enable-break bstr
                                skip-bytes-amt
                               [progress
                                in
                                start-pos
                                end-pos])
 → (or/c nonnegative-exact-integer? eof-object? procedure?)
  bstr : (and/c bytes? (not/c immutable?))
  skip-bytes-amt : nonnegative-exact-integer?
  progress : (or/c evt? false/c) = #f
  in : input-port? = (current-input-port)
  start-pos : nonnegative-exact-integer? = 0
  end-pos : nonnegative-exact-integer? = (bytes-length bstr)
```

Like `read-bytes-avail!/enable-break`, but for peeking, and with *skip-bytes-amt* and *progress* arguments like `peek-bytes-avail!`.

```
(read-char-or-special [in]) → (or/c character? eof-object? any/c)
  in : input-port? = (current-input-port)
```

Like `read-char`, but that if the input port returns a non-byte value (through a value-generating procedure in a custom port; see §12.1.9 “Custom Ports” and §12.9.3 “Special Comments” for details), the non-byte value is returned.

```
(read-byte-or-special [in]) → (or/c byte? eof-object? any/c)
  in : input-port? = (current-input-port)
```

Like `read-char-or-special`, but reads and returns a byte instead of a character.

```
(peek-char [in skip-bytes-amt]) → (or/c character? eof-object?)
  in : input-port? = (current-input-port)
  skip-bytes-amt : nonnegative-exact-integer? = 0
```

Like `read-char`, but peeks instead of reading, and skipping *`skip-bytes-amt`* bytes (not characters) at the start of the port.

```
(peek-byte [in skip-bytes-amt]) → (or/c byte? eof-object?)
  in : input-port? = (current-input-port)
  skip-bytes-amt : nonnegative-exact-integer? = 0
```

Like `peek-char`, but reads and returns a byte instead of a character.

```
(peek-char-or-special [in skip-bytes-amt])
 → (or/c character? eof-object? any/c)
  in : input-port? = (current-input-port)
  skip-bytes-amt : nonnegative-exact-integer? = 0
```

Like `peek-char`, but if the input port returns a non-byte value after *`skip-bytes-amt`* byte positions, it is returned.

```
(peek-byte-or-special [in
                       skip-bytes-amt
                       progress])
 → (or/c character? eof-object? any/c)
  in : input-port? = (current-input-port)
  skip-bytes-amt : nonnegative-exact-integer? = 0
  progress : (or/c evt? false/c) = #f
```

Like `peek-char-or-special`, but reads and returns a byte instead of a character, and it supports a *`progress`* argument like `peek-bytes-avail!`.

```
(port-progress-evt [in]) → evt?
  in : input-port? = (current-input-port)
```

Returns an event that becomes ready after any subsequent read from *`in`*, or after *`in`* is closed. After the event becomes ready, it remains ready. If progress events are unavailable for *`in`* (as reported by `port-provides-progress-evts?`) the `exn:fail:contract` exception is raised.

```
(port-provides-progress-evts? in) → boolean
  in : input-port?
```

Returns `#t` if `port-progress-evt` can return an event for *in*. All built-in kinds of ports support progress events, but ports created with `make-input-port` (see §12.1.9 "Custom Ports") may not.

```
(port-commit-peeked amt progress evt [in]) → boolean?
  amt : nonnegative-exact-integer?
  progress : evt?
  evt : evt?
  in : input-port? = (current-input-port)
```

Attempts to commit as read the first *amt* previously peeked bytes, non-byte specials, and `eof`s from *in*, or the first `eof` or special value peeked from *in*. (Only mid-stream `eof`s can be committed. A `eof` when the port is exhausted does not correspond to data in the stream.)

The read commits only if *progress* does not become ready first (i.e., if no other process reads from *in* first), and only if *evt* is chosen by a `sync` within `port-commit-peeked` (in which case the event result is ignored); the *evt* must be either a channel-put event, channel, semaphore, semaphore-peek event, always event, or never event. Suspending the thread that calls `port-commit-peeked` may or may not prevent the commit from proceeding.

The result from `port-commit-peeked` is `#t` if data is committed, and `#f` otherwise.

If no data has been peeked from *in* and *progress* is not ready, then `exn:fail:contract` exception is raised. If fewer than *amt* items have been peeked at the current start of *in*'s stream, then only the peeked items are committed as read. If *in*'s stream currently starts at an `eof` or a non-byte special value, then only the `eof` or special value is committed as read.

If *progress* is not a result of `port-progress-evt` applied to *in*, then `exn:fail:contract` exception is raised.

```
(byte-ready? [in]) → boolean?
  in : input-port? = (current-input-port)
```

Returns `#t` if `(read-byte in)` would not block (at the time that `byte-ready?` was called, at least). Equivalent to `(and (sync/timeout 0 in) #t)`.

```
(char-ready? [in]) → boolean?
  in : input-port? = (current-input-port)
```

Returns `#t` if `(read-char in)` would not block (at the time that `char-ready?` was called, at least). Depending on the initial bytes of the stream, multiple bytes may be needed to form a UTF-8 encoding.

12.3 Byte and String Output

```
(write-char char [out]) → void?
  char : character?
  out : output-port? = (current-output-port)
```

Writes a single character to *out*; more precisely, the bytes that are the UTF-8 encoding of *char* are written to *out*.

```
(write-byte byte [out]) → void?
  byte : any/c
  out : output-port? = (current-output-port)
```

Writes a single byte to *out*.

```
(newline [out]) → void?
  out : output-port? = (current-output-port)
```

The same as `(write-char #\newline out)`.

```
(write-string str [out start-pos end-pos]) → void?
  str : string?
  out : output-port? = (current-output-port)
  start-pos : nonnegative-exact-integer? = 0
  end-pos : nonnegative-exact-integer? = (string-length str)
```

Writes characters to *out* from *str* starting from index *start-pos* (inclusive) up to *end-pos* (exclusive). Like `substring`, the `exn:fail:contract` exception is raised if *start-pos* or *end-pos* is out-of-range for *str*.

The result is the number of characters written to *out*, which is always `(- end-pos start-pos)`.

```
(write-bytes bstr [out start-pos end-pos]) → void?
  bstr : bytes?
  out : output-port? = (current-output-port)
  start-pos : nonnegative-exact-integer? = 0
  end-pos : nonnegative-exact-integer? = (bytes-length bstr)
```

Like write-string, but writes bytes instead of characters.

```
(write-bytes-avail  bstr
                   [out
                    start-pos
                    end-pos])  → nonnegative-exact-integer?
  bstr : bytes?
  out : output-port? = (current-output-port)
  start-pos : nonnegative-exact-integer? = 0
  end-pos : nonnegative-exact-integer? = (bytes-length bstr)
```

Like write-bytes, but returns without blocking after writing as many bytes as it can immediately flush. It blocks only if no bytes can be flushed immediately. The result is the number of bytes written and flushed to *out*; if *start-pos* is the same as *end-pos*, then the result can be 0 (indicating a successful flush of any buffered data), otherwise the result is at least 1 but possibly less than (- *end-pos* *start-pos*).

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

```
(write-bytes-avail*  bstr
                    [out
                     start-pos
                     end-pos])
 → (or/c nonnegative-exact-integer? false/c)
  bstr : bytes?
  out : output-port? = (current-output-port)
  start-pos : nonnegative-exact-integer? = 0
  end-pos : nonnegative-exact-integer? = (bytes-length bstr)
```

Like write-bytes-avail, but never blocks, returns #f if the port contains buffered data that cannot be written immediately, and returns 0 if the port's internal buffer (if any) is flushed but no additional bytes can be written immediately.

```
(write-bytes-avail/enable-break  bstr
                                [out
                                 start-pos
                                 end-pos])
 → nonnegative-exact-integer?
  bstr : bytes?
  out : output-port? = (current-output-port)
  start-pos : nonnegative-exact-integer? = 0
```

```
  end-pos : nonnegative-exact-integer? = (bytes-length bstr)
```

Like `write-bytes-avail`, except that breaks are enabled during the write. The procedure provides a guarantee about the interaction of writing and breaks: if breaking is disabled when `write-bytes-avail/enable-break` is called, and if the `exn:break` exception is raised as a result of the call, then no bytes will have been written to *out*. See also §9.6 "Breaks".

```
(write-special v [out]) → boolean?
  v : any/c
  out : output-port? = (current-output-port)
```

Writes *v* directly to *out* if the port supports special writes, or raises `exn:fail:contract` if the port does not support special write. The result is always `#t`, indicating that the write succeeded.

```
(write-special-avail* v [out]) → boolean?
  v : any/c
  out : output-port? = (current-output-port)
```

Like `write-special`, but without blocking. If *v* cannot be written immediately, the result is `#f` without writing *v*, otherwise the result is `#t` and *v* is written.

```
(write-bytes-avail-evt  bstr
                       [out
                        start-pos
                        end-pos]) → evt?
  bstr : bytes?
  out : output-port? = (current-output-port)
  start-pos : nonnegative-exact-integer? = 0
  end-pos : nonnegative-exact-integer? = (bytes-length bstr)
```

Similar to `write-bytes-avail`, but instead of writing bytes immediately, it returns a synchronizable event (see §10.2.1 "Events"). The *out* must support atomic writes, as indicated by `port-writes-atomic?`.

Synchronizing on the object starts a write from *bstr*, and the event becomes ready when bytes are written (unbuffered) to the port. If *start-pos* and *end-pos* are the same, then the synchronization result is `0` when the port's internal buffer (if any) is flushed, otherwise the result is a positive exact integer. If the event is not selected in a synchronization, then no bytes will have been written to *out*.

```
(write-special-evt v [out]) → evt?
  v : any/c
```

`out : output-port? = (current-output-port)`

Similar to `write-special`, but instead of writing the special value immediately, it returns a synchronizable event (see §10.2.1 "Events"). The *out* must support atomic writes, as indicated by `port-writes-atomic?`.

Synchronizing on the object starts a write of the special value, and the event becomes ready when the value is written (unbuffered) to the port. If the event is not selected in a synchronization, then no value will have been written to *out*.

```
(port-writes-atomic? out) → boolean?
  out : output-port?
```

Returns `#t` if `write-bytes-avail/enable-break` can provide an exclusive-or guarantee (break or write, but not both) for *out*, and if the port can be used with procedures like `write-bytes-avail-evt`. Scheme's file-stream ports, pipes, string ports, and TCP ports all support atomic writes; ports created with `make-output-port` (see §12.1.9 "Custom Ports") may support atomic writes.

```
(port-writes-special? out) → boolean?
  out : output-port?
```

Returns `#t` if procedures like `write-special` can write arbitrary values to the port. Scheme's file-stream ports, pipes, string ports, and TCP ports all reject special values, but ports created with `make-output-port` (see §12.1.9 "Custom Ports") may support them.

12.4 Reading

```
(read [in]) → any
  in : input-port? = (current-input-port)
```

Reads and returns a single datum from *in*. If *in* has a handler associated to it via `port-read-handler`, then the handler is called. Otherwise, the default reader is used, as parameterized by the `current-readtable` parameter, as well as many other parameters.

See §12.6 "The Reader" for information on the default reader.

```
(read-syntax [source-name in]) → (or/c syntax? eof-object?)
  source-name : any/c = (object-name in)
  in : input-port? = (current-input-port)
```

Like `read`, but produces a syntax object with source-location information. The *source-name* is used as the source field of the syntax object; it can be an arbitrary value, but it should generally be a path for the source file.

See §12.6 "The Reader" for information on the default reader in `read-syntax` mode.

```
(read/recursive [in start readtable graph?]) → any
  in : input-port? = (current-input-port)
  start : (or/c character? false/c) = #f
  readtable : readtable? = (current-readtable)
  graph? : any/c = #f
```

Similar to calling `read`, but normally used during the dynamic extent of `read` within a reader-extension procedure (see §12.9.2 "Reader-Extension Procedures"). The main effect of using `read/recursive` instead of `read` is that graph-structure annotations (see §12.6.16 "Reading Graph Structure") in the nested read are considered part of the overall read, at least when the *graph?* argument is true; since the result is wrapped in a placeholder, however, it is not directly inspectable.

If *start* is provided and not `#f`, it is effectively prefixed to the beginning of *in*'s stream for the read. (To prefix multiple characters, use `input-port-append`.)

The *readtable* argument is used for top-level parsing to satisfy the read request; recursive parsing within the read (e.g., to read the elements of a list) instead uses the current readtable as determined by the `current-readtable` parameter. A reader macro might call `read/recursive` with a character and readtable to effectively invoke the readtable's behavior for the character. If *readtable* is `#f`, the default readtable is used for top-level parsing.

When *graph?* is `#f`, graph structure annotations in the read datum are local to the datum.

When called within the dynamic extent of `read`, the `read/recursive` procedure produces either an opaque placeholder value, a special-comment value, or an end-of-file. The result is a special-comment value (see §12.9.3 "Special Comments") when the input stream's first non-whitespace content parses as a comment. The result is end-of-file when `read/recursive` encounters an end-of-file. Otherwise, the result is a placeholder that protects graph references that are not yet resolved. When this placeholder is returned within an S-expression that is produced by any reader-extension procedure (see §12.9.2 "Reader-Extension Procedures") for the same outermost `read`, it will be replaced with the actual read value before the outermost `read` returns.

See §12.9.1 "Readtables" for an extended example that uses `read/recursive`.

```
(read-syntax/recursive [source-name
                        in
                        start
                        readtable
                        graph?])     → any
  source-name : any/c = (object-name in)
  in : input-port? = (current-input-port)
  start : (or/c character? false/c) = #f
  readtable : readtable? = (current-readtable)
  graph? : any/c = #f
```

Analogous to calling `read/recursive`, but the resulting value encapsulates S-expression structure with source-location information. As with `read/recursive`, when `read-syntax/recursive` is used within the dynamic extent of `read-syntax`, the result of from `read-syntax/recursive` is either a special-comment value, end-of-file, or opaque graph-structure placeholder (not a syntax object). The placeholder can be embedded in an S-expression or syntax object returned by a reader macro, etc., and it will be replaced with the actual syntax object before the outermost `read-syntax` returns.

Using `read/recursive` within the dynamic extent of `read-syntax` does not allow graph structure for reading to be included in the outer `read-syntax` parsing, and neither does using `read-syntax/recursive` within the dynamic extent of `read`. In those cases, `read/recursive` and `read-syntax/recursive` produce results like `read` and `read-syntax`, except that a special-comment value is returned when the input stream starts with a comment (after whitespace).

See §12.9.1 “Readtables” for an extended example that uses `read-syntax/recursive`.

```
(read-case-sensitive) → boolean?
(read-case-sensitive on?) → void?
  on? : any/c
```

A parameter that controls parsing and printing of symbols. When this parameter’s value is `#f`, the reader case-folds symbols (e.g., producing `'hi` when the input is any one of \litchar{hi}, `Hi`, `HI`, or `hI`). The parameter also affects the way that `write` prints symbols containing uppercase characters; if the parameter’s value is `#f`, then symbols are printed with uppercase characters quoted by a `\` or `|`. The parameter’s value is overridden by quoting `\` or `|` vertical-bar quotes and the `#cs` and `#ci` prefixes; see §12.6.2 “Reading Symbols” for more information. While a module is loaded, the parameter is set to `#t` (see `current-load`).

```
(read-square-bracket-as-paren) → boolean?
(read-square-bracket-as-paren on?) → void?
  on? : any/c
```

A parameter that controls whether `[` and `]` are treated as parentheses. See §12.6.5 “Reading

Pairs and Lists" for more information.

```
(read-curly-brace-as-paren) → boolean?
(read-curly-brace-as-paren on?) → void?
  on? : any/c
```

A parameter that controls whether `{` and `}` are treated as parentheses. See §12.6.5 "Reading Pairs and Lists" for more information.

```
(read-accept-box) → boolean?
(read-accept-box on?) → void?
  on? : any/c
```

A parameter that controls parsing `#&` input. See §12.6.12 "Reading Boxes" for more information.

```
(read-accept-compiled) → boolean?
(read-accept-compiled on?) → void?
  on? : any/c
```

A parameter that controls parsing `#~` compiled input. See §12.6 "The Reader" and `current-compile` for more information.

```
(read-accept-bar-quote) → boolean?
(read-accept-bar-quote on?) → void?
  on? : any/c
```

A parameter that controls parsing and printing of `|` in symbols. See §12.6.2 "Reading Symbols" and §12.7 "The Printer" for more information.

```
(read-accept-graph) → boolean?
(read-accept-graph on?) → void?
  on? : any/c
```

A parameter value that controls parsing input with sharing. See §12.6.16 "Reading Graph Structure" for more information.

```
(read-decimal-as-inexact) → boolean?
(read-decimal-as-inexact on?) → void?
  on? : any/c
```

A parameter that controls parsing input numbers with a decimal point or exponent (but no explicit exactness tag). See §12.6.3 “Reading Numbers” for more information.

```
(read-accept-dot) → boolean?
(read-accept-dot on?) → void?
  on? : any/c
```

A parameter that controls parsing input with a dot, which is normally used for literal cons cells. See §12.6.5 “Reading Pairs and Lists” for more information.

```
(read-accept-infix-dot) → boolean?
(read-accept-infix-dot on?) → void?
  on? : any/c
```

A parameter that controls parsing input with two dots to trigger infix conversion. See §12.6.5 “Reading Pairs and Lists” for more information.

```
(read-accept-quasiquote) → boolean?
(read-accept-quasiquote on?) → void?
  on? : any/c
```

A parameter that controls parsing input with ` or , which is normally used for quasiquote, unquote, and unquote-splicing abbreviations. See §12.6.7 “Reading Quotes” for more information.

```
(read-accept-reader) → boolean?
(read-accept-reader on?) → void?
  on? : any/c
```

A parameter that controls whether #reader is allowed for selecting a parser. See §12.6.17 “Reading via an Extension” for more information.

```
(current-reader-guard) → (any/c . -> . any)
(current-reader-guard proc) → void?
  proc : (any/c . -> . any)
```

A parameter whose value converts or rejects (by raising an exception) a module-path datum following #reader. See §12.6.17 “Reading via an Extension” for more information.

```
(current-readtable) → (or/c readtable? false/c)
(current-readtable readtable) → void?
```

```
readtable : (or/c readtable? false/c)
```

A parameter whose value determines a readtable that adjusts the parsing of S-expression input, where `#f` implies the default behavior. See §12.9.1 "Readtables" for more information.

```
(read-on-demand-source) → (and/c path? complete-path?)
(read-on-demand-source path) → void?
  path : (and/c path? complete-path?)
```

A parameter that enables lazy parsing of compiled code, so that closure bodies and syntax objects are extracted (and validated) from marshaled compiled code on demand. Normally, this parameter is set by the default load handler when `load-on-demand-enabled` is `#t`.

Even when parsing is delayed, compiled code is loaded into memory. If the `PLT_DELAY_FROM_ZO` environment variable is set (to any value) on start up, however, even loading from disk is delayed. If the file at path changes before the delayed code or syntax object is demanded, the read-on-demand most likely will encounter garbage, leading to an exception.

```
(port-read-handler in) → (case->
                           (input-port? . -> . any)
                           (input-port?  any/c . -> . any))
  in : input-port?
(port-read-handler in proc) → void?
  in : input-port?
  proc : (case->
          (input-port? . -> . any)
          (input-port? any/c . -> . any))
```

Gets or sets the *port read handler* for `in`. The handler called to read from the port when the built-in `read` or `read-syntax` procedure is applied to the port. (The port read handler is not used for `read/recursive` or `read-syntax/recursive`.)

A port read handler is applied to either one argument or two arguments:

- A single argument is supplied when the port is used with `read`; the argument is the port being read. The return value is the value that was read from the port (or end-of-file).
- Two arguments are supplied when the port is used with `read-syntax`; the first argument is the port being read, and the second argument is a value indicating the source. The return value is a syntax object that was read from the port (or end-of-file).

The default port read handler reads standard Scheme expressions with Scheme's built-in parser (see §12.6 "The Reader"). It handles a special result from a custom input port (see `make-custom-input-port`) by treating it as a single expression, except that special-comment values (see §12.9.3 "Special Comments") are treated as whitespace.

The default port read handler itself can be customized through a readtable; see §12.9.1 "Readtables" for more information.

```
(read-honu [in]) → any
  in : input-port? = (current-input-port)
```

Like `read`, but for Honu mode (see §12.6.18 "Honu Parsing").

```
(read-honu-syntax [source-name in]) → (or/c syntax? eof-object?)
  source-name : any/c = (object-name in)
  in : input-port? = (current-input-port)
```

Like `read-syntax`, but for Honu mode (see §12.6.18 "Honu Parsing").

```
(read-honu/recursive [in
                      start
                      readtable
                      graph?])   → any
  in : input-port? = (current-input-port)
  start : (or/c character? false/c) = #f
  readtable : readtable? = (current-readtable)
  graph? : any/c = #f
```

Like `read/recursive`, but for Honu mode (see §12.6.18 "Honu Parsing").

```
(read-honu-syntax/recursive [source-name
                             in
                             start
                             readtable
                             graph?])     → any
  source-name : any/c = (object-name in)
  in : input-port? = (current-input-port)
  start : (or/c character? false/c) = #f
  readtable : readtable? = (current-readtable)
  graph? : any/c = #f
```

Like `read-syntax/recursive`, but for Honu mode (see §12.6.18 "Honu Parsing").

12.5 Writing

```
(write datum [out]) → void?
  datum : any/c
  out : output-port? = (current-output-port)
```

Writes *datum* to *out*, normally in such a way that instances of core datatypes can be read back in. If *out* has a handler associated to it via `port-write-handler`, then the handler is called. Otherwise, the default printer is used (in `write` mode), as configured by various parameters.

See §12.7 "The Printer" for more information about the default printer. In particular, note that `write` may require memory proportional to the depth of the value being printed, due to the initial cycle check.

```
(display datum [out]) → void?
  datum : any/c
  out : output-port? = (current-output-port)
```

Displays *datum* to *out*, similar to `write`, but usually in such a way that byte- and character-based datatypes are written as raw bytes or characters. If *out* has a handler associated to it via `port-display-handler`, then the handler is called. Otherwise, the default printer is used (in `display` mode), as configured by various parameters.

See §12.7 "The Printer" for more information about the default printer. In particular, note that `display` may require memory proportional to the depth of the value being printed, due to the initial cycle check.

```
(print datum [out]) → void?
  datum : any/c
  out : output-port? = (current-output-port)
```

Writes *datum* to *out*, normally the same way as `write`. If *out* has a handler associated to it via `port-print-handler`, then the handler is called. Otherwise, the handler specified by `global-port-print-handler` is called; the default handler uses the default printer in `write` mode.

The rationale for providing `print` is that `display` and `write` both have relatively 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.

```
(fprintf out form v ...) → void?
  out : output-port?
  form : string?
  v : any/c
```

Prints formatted output to *out*, where *form* is a string that is printed directly, except for special formatting escapes:

- `~n` or `~%` prints a newline, the same as `\n`
- `~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 `error-value->string-handler`) and current error printing width
- `~c` or `~C` `write-char`s the next argument in *v*s; if the next argument is not a character, the `exn:fail:contract` exception is raised
- `~b` or `~B` prints the next argument among the *v*s in binary; if the next argument is not an exact number, the `exn:fail:contract` exception is raised
- `~o` or `~O` prints the next argument among the *v*s in octal; if the next argument is not an exact number, the `exn:fail:contract` exception is raised
- `~x` or `~X` prints the next argument among the *v*s in hexadecimal; if the next argument is not an exact number, the `exn:fail:contract` exception is raised
- `~~` prints a tilde.
- ~⟨*w*⟩, where ⟨*w*⟩ is a whitespace character (see `char-whitespace?`), skips characters in *form* until a non-whitespace character is encountered or until a second end-of-line is encountered (whichever happens first). On all platforms, an end-of-line can be `#\return`, `#\newline`, or `#\return` followed immediately by `#\newline`.

The *form* string must not contain any ~ that is not one of the above escapes, otherwise the `exn:fail:contract` exception is raised. When the format string requires more *v*s than are supplied, the `exn:fail:contract` exception is raised. Similarly, when more *v*s are supplied than are used by the format string, the `exn:fail:contract` exception is raised.

Examples:

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

```
(printf form v ...) → void?
  form : string?
  v : any/c
```

The same as `(fprintf (current-output-port) form v ...)`.

```
(format form v ...) → string?
  form : string?
  v : any/c
```

Formats to a string. The result is the same as

```
(let ([o (open-output-string)])
  (fprintf o form v ...)
  (get-output-string o))
```

Examples:

```
> (format "~a as a string is ~s.~n" '(3 4) "(3 4)")
"(3 4) as a string is \"(3 4)\".\n"
```

```
(print-pair-curly-braces) → boolean?
(print-pair-curly-braces on?) → void?
  on? : any/c
```

A parameter that control pair printing. If the value is true, then pairs print using `{` and `}` instead of `(` and `)`. The default is `#f`.

```
(print-mpair-curly-braces) → boolean?
(print-mpair-curly-braces on?) → void?
  on? : any/c
```

A parameter that control pair printing. If the value is true, then mutable pairs print using `{` and `}` instead of `(` and `)`. The default is `#t`.

```
(print-unreadable) → boolean?
(print-unreadable on?) → void?
  on? : any/c
```

A parameter that controls printing values that have no `read`able form (using the default reader), including structures that have a custom-write procedure (see `prop:custom-write`); defaults to `#t`. See §12.7 “The Printer” for more information.

```
(print-graph) → boolean?
(print-graph on?) → void?
  on? : any/c
```

A parameter that controls printing data with sharing; defaults to `#f`. See §12.7 "The Printer" for more information.

```
(print-struct) → boolean?
(print-struct on?) → void?
  on? : any/c
```

A parameter that controls printing structure values in vector or prefab form; defaults to `#t`. See §12.7 "The Printer" for more information. This parameter has no effect on the printing of structures that have a custom-write procedure (see `prop:custom-write`).

```
(print-box) → boolean?
(print-box on?) → void?
  on? : any/c
```

A parameter that controls printing box values; defaults to `#t`. See §12.7.9 "Printing Boxes" for more information.

```
(print-vector-length) → boolean?
(print-vector-length on?) → void?
  on? : any/c
```

A parameter that controls printing vectors; defaults to `#f`. See §12.7.6 "Printing Vectors" for more information.

```
(print-hash-table) → boolean?
(print-hash-table on?) → void?
  on? : any/c
```

A parameter that controls printing hash tables; defaults to `#f`. See §12.7.8 "Printing Hash Tables" for more information.

```
(print-honu) → boolean?
(print-honu on?) → void?
  on? : any/c
```

A parameter that controls printing values in an alternate syntax. See §"**Honu**" for more information.

```
(current-write-relative-directory)
 → (or/c (and/c path? complete-path?) false/c)
(current-write-relative-directory path) → void?
  path : (or/c (and/c path-string? complete-path?) false/c)
```

A parameter that is used when writing compiled code that contains pathname literals, including source-location pathnames for procedure names. When not `#f`, paths that syntactically extend the parameter's value are converted to relative paths; when the resulting compiled code is read, relative paths are converted back to complete paths using the `current-load-relative-directory` parameter (if it is not `#f`, otherwise the path is left relative).

```
(port-write-handler out) → (any/c output-port? . -> . any)
  out : output-port?
(port-write-handler out proc) → void?
  out : output-port?
  proc : (any/c output-port? . -> . any)
```

```
(port-display-handler out) → (any/c output-port? . -> . any)
  out : output-port?
(port-display-handler out proc) → void?
  out : output-port?
  proc : (any/c output-port? . -> . any)
```

```
(port-print-handler out) → (any/c output-port? . -> . any)
  out : output-port?
(port-print-handler out proc) → void?
  out : output-port?
  proc : (any/c output-port? . -> . any)
```

Gets or sets the *port write handler*, *port display handler*, or *port print handler* for `out`. This handler is call to output to the port when `write`, `display`, or `print` (respectively) is applied to the port. Each handler takes a two arguments: the value to be printed and the destination port. The handler's return value is ignored.

The default port display and write handlers print Scheme expressions with Scheme's built-in printer (see §12.7 "The Printer"). The default print handler calls the global port print handler (the value of the `global-port-print-handler` parameter); the default global port print handler is the same as the default write handler.

```
(global-port-print-handler) → (any/c output-port? . -> . any)
```

```
(global-port-print-handler proc) → void?
  proc : (any/c output-port? . -> . any)
```

A parameter that determines *global port print handler*, which is called by the default port print handler (see `port-print-handler`) to `print` values into a port. The default value uses the built-in printer (see §12.7 "The Printer") in `write` mode.

12.6 The Reader

Scheme's reader is a recursive-descent parser that can be configured through a readtable and various other parameters. This section describes the reader's parsing when using the default readtable.

Reading from a stream produces one *datum*. If the result datum is a compound value, then reading the datum typically requires the reader to call itself recursively to read the component data.

The reader can be invoked in either of two modes: `read` mode, or `read-syntax` mode. In `read-syntax` mode, the result is always a syntax object that includes source-location and (initially empty) lexical information wrapped around the sort of datum that `read` mode would produce. In the case of pairs, vectors, and boxes, the content is also wrapped recursively as a syntax object. Unless specified otherwise, this section describes the reader's behavior in `read` mode, and `read-syntax` mode does the same modulo wrapping the final result.

Reading is defined in terms of Unicode characters; see §12.1 "Ports" for information on how a byte stream is converted to a character stream.

12.6.1 Delimiters and Dispatch

Along with whitespace, the following characters are *delimiters*:

`( ) [ ] [ ] " , ' ` ;`

A delimited sequence that starts with any other character is typically parsed as either a symbol or number, but a few non-delimiter characters play special roles:

- `#` has a special meaning as an initial character in a delimited sequence; its meaning depends on the characters that follow; see below.
- `|` starts a subsequence of characters to be included verbatim in the delimited sequence (i.e,. they are never treated as delimiters, and they are not case-folded when case-insensitivity is enabled); the subsequence is terminated by another `|`, and neither the initial nor terminating `|` is part of the subsequence.

- `\` outside of a `|` pair causes the folowing character to be included verbatim in a delimited sequence.

More precisely, after skipping whitespace, the reader dispatches based on the next character or characters in the input stream as follows:

`(` starts a pair or list; see §12.6.5 “Reading Pairs and Lists”
`[` starts a pair or list; see §12.6.5 “Reading Pairs and Lists”
`{` starts a pair or list; see §12.6.5 “Reading Pairs and Lists”
`)` matches `(` or raises `exn:fail:read`
`]` matches `[` or raises `exn:fail:read`
`}` matches `{` or raises `exn:fail:read`
`"` starts a string; see §12.6.6 “Reading Strings”
`'` starts a quote; see §12.6.7 “Reading Quotes”
`` ` `` starts a quasiquote; see §12.6.7 “Reading Quotes”
`,` starts an [splicing] unquote; see §12.6.7 “Reading Quotes”
`;` starts a line comment; see §12.6.8 “Reading Comments”
`#t` or `#T` true; see §12.6.4 “Reading Booleans”
`#f` or `#F` false; see §12.6.4 “Reading Booleans”
`#(` starts a vector; see §12.6.9 “Reading Vectors”
`#[` starts a vector; see §12.6.9 “Reading Vectors”
`#{` starts a vector; see §12.6.9 “Reading Vectors”
`#s(` starts a structure literal; see §12.6.10 “Reading Structures”
`#s[` starts a structure literal; see §12.6.10 “Reading Structures”
`#s{` starts a structure literal; see §12.6.10 “Reading Structures”
`#\` starts a character; see §12.6.13 “Reading Characters”
`#"` starts a byte string; see §12.6.6 “Reading Strings”
`#%` starts a symbol; see §12.6.2 “Reading Symbols”
`#:` starts a keyword; see §12.6.14 “Reading Keywords”
`#&` starts a box; see §12.6.12 “Reading Boxes”
`#|` starts a block comment; see §12.6.8 “Reading Comments”
`#;` starts an S-expression comment; see §12.6.8 “Reading Comments”
`#'` starts a syntax quote; see §12.6.7 “Reading Quotes”
`#! ` starts a line comment; see §12.6.8 “Reading Comments”
`#!/` starts a line comment; see §12.6.8 “Reading Comments”
`#!` may start a reader extension; see §12.6.17 “Reading via an Extension”
``#` `` starts a syntax quasiquote; see §12.6.7 “Reading Quotes”
`#,` starts an syntax [splicing] unquote; see §12.6.7 “Reading Quotes”
`#~` starts compiled code; see `current-compile`
`#i` or `#I` starts a number; see §12.6.3 “Reading Numbers”
`#e` or `#E` starts a number; see §12.6.3 “Reading Numbers”
`#x` or `#X` starts a number; see §12.6.3 “Reading Numbers”
`#o` or `#O` starts a number; see §12.6.3 “Reading Numbers”
`#d` or `#D` starts a number; see §12.6.3 “Reading Numbers”
`#b` or `#B` starts a number; see §12.6.3 “Reading Numbers”
`#<<` starts a string; see §12.6.6 “Reading Strings”

`#rx`	starts a regular expression; see §12.6.15 “Reading Regular Expressions”
`#px`	starts a regular expression; see §12.6.15 “Reading Regular Expressions”
`#ci`, `#cI`, `#Ci`, or `#CI`	switches case sensitivity; see §12.6.2 “Reading Symbols”
`#cs`, `#cS`, `#Cs`, or `#CS`	switches case sensitivity; see §12.6.2 “Reading Symbols”
`#sx`, `#sX`, `#Sx`, or `#SX`	starts a Scheme expression; see §12.6.18 “Honu Parsing”
`#hx`	starts a Honu expression; see §12.6.18 “Honu Parsing”
`#hash`	starts a hash table; see §12.6.11 “Reading Hash Tables”
`#reader`	starts a reader extension use; see §12.6.17 “Reading via an Extension”
`#lang`	starts a reader extension use; see §12.6.17 “Reading via an Extension”
`#`$\langle$*digit*$_{10}\rangle^{+}$`(`	starts a vector; see §12.6.9 “Reading Vectors”
`#`$\langle$*digit*$_{10}\rangle^{+}$`[`	starts a vector; see §12.6.9 “Reading Vectors”
`#`$\langle$*digit*$_{10}\rangle^{+}$`{`	starts a vector; see §12.6.9 “Reading Vectors”
`#`$\langle$*digit*$_{10}\rangle^{\{1,8\}}$`=`	binds a graph tag; see §12.6.16 “Reading Graph Structure”
`#`$\langle$*digit*$_{10}\rangle^{\{1,8\}}$`#`	uses a graph tag; see §12.6.16 “Reading Graph Structure”
otherwise	starts a symbol; see §12.6.2 “Reading Symbols”

12.6.2 Reading Symbols

§3.6 “Symbols” in §“**Guide**: PLT Scheme” introduces the syntax of symbols.

A sequence that does not start with a delimiter or `#` is parsed as either a symbol or a number (see §12.6.3 “Reading Numbers”), except that `.` by itself is never parsed as a symbol or character (unless the `read-accept-dot` parameter is set to `#f`). A `#%` also starts a symbol. A successful number parse takes precedence over a symbol parse.

When the `read-case-sensitive` parameter is set to `#f`, characters in the sequence that are not quoted by `|` or `\` are first case-normalized. If the reader encounters `#ci`, `#CI`, `#Ci`, or `#cI`, then it recursively reads the following datum in case-insensitive mode. If the reader encounters `#cs`, `#CS`, `#Cs`, or `#cS`, then recursively reads the following datum in case-sensitive mode.

Examples:

`Apple`	reads equal to	`(string->symbol "Apple")`
`Ap#ple`	reads equal to	`(string->symbol "Ap#ple")`
`Ap ple`	reads equal to	`(string->symbol "Ap")`
`Ap| |ple`	reads equal to	`(string->symbol "Ap ple")`
`Ap\ ple`	reads equal to	`(string->symbol "Ap ple")`
`#ci Apple`	reads equal to	`(string->symbol "apple")`
`#ci |A|pple`	reads equal to	`(string->symbol "Apple")`
`#ci \Apple`	reads equal to	`(string->symbol "Apple")`
`#ci#cs Apple`	reads equal to	`(string->symbol "Apple")`
`#%Apple`	reads equal to	`(string->symbol "#%Apple")`

12.6.3 Reading Numbers

§3.2 "Numbers" in §"**Guide**: PLT Scheme" introduces the syntax of numbers.

A sequence that does not start with a delimiter is parsed as a number when it matches the following grammar case-insenstively for $\langle\mathit{number}_{10}\rangle$ (decimal), where n is a meta-meta-variable in the grammar.

A number is optionally prefixed by an exactness specifier, `#e` (exact) or `#i` (inexact), which specifies its parsing as an exact or inexact number; see §3.2 "Numbers" for information on number exactness. As the non-terminal names suggest, a number that has no exactness specifier and matches only $\langle\mathit{inexact\text{-}number}_n\rangle$ is normally parsed as an inexact number, otherwise it is parsed as an excat number. If the `read-decimal-as-inexact` parameter is set to `#f`, then all numbers without an exactness specifier are instead parsed as exact.

If the reader encounters `#b` (binary), `#o` (octal), `#d` (decimal), or `#x` (hexadecimal), it must be followed by a sequence that is terminated by a delimiter or end-of-file, and that matches the $\langle\mathit{general\text{-}number}_2\rangle$, $\langle\mathit{general\text{-}number}_8\rangle$, $\langle\mathit{general\text{-}number}_{10}\rangle$, or $\langle\mathit{general\text{-}number}_{16}\rangle$ grammar, respectively.

An $\langle\mathit{exponent\text{-}mark}_n\rangle$ in an inexact number serves both to specify an exponent and specify a numerical precision. If single-precision IEEE floating point is supported (see §3.2 "Numbers"), the marks `f` and `s` specifies single-precision. Otherwise, or with any other mark, double-precision IEEE floating point is used.

$\langle\mathit{number}_n\rangle$	::=	$\langle\mathit{exact}_n\rangle$ $\mid$ $\langle\mathit{inexact}_n\rangle$
$\langle\mathit{exact}_n\rangle$	::=	$\langle\mathit{exact\text{-}integer}_n\rangle$ $\mid$ $\langle\mathit{exact\text{-}rational}_n\rangle$
	$\mid$	$\langle\mathit{exact\text{-}complex}_n\rangle$
$\langle\mathit{exact\text{-}integer}_n\rangle$	::=	[$\langle\mathit{sign}\rangle$] $\langle\mathit{digits}_n\rangle$
$\langle\mathit{digits}_n\rangle$	::=	$\langle\mathit{digit}_n\rangle^+$
$\langle\mathit{exact\text{-}rational}_n\rangle$	::=	$\langle\mathit{exact\text{-}integer}_n\rangle$ `/` $\langle\mathit{unsigned\text{-}integer}_n\rangle$
$\langle\mathit{exact\text{-}complex}_n\rangle$	::=	$\langle\mathit{exact\text{-}rational}_n\rangle$ $\langle\mathit{sign}\rangle$ $\langle\mathit{exact\text{-}rational}_n\rangle$ `i`
$\langle\mathit{inexact}_n\rangle$	::=	$\langle\mathit{inexact\text{-}real}_n\rangle$ $\mid$ $\langle\mathit{inexact\text{-}complex}_n\rangle$
$\langle\mathit{inexact\text{-}real}_n\rangle$	::=	[$\langle\mathit{sign}\rangle$] $\langle\mathit{inexact\text{-}normal}_n\rangle$
	$\mid$	$\langle\mathit{sign}\rangle$ $\langle\mathit{inexact\text{-}special}_n\rangle$
$\langle\mathit{inexact\text{-}unsigned}_n\rangle$	::=	$\langle\mathit{inexact\text{-}normal}_n\rangle$ $\mid$ $\langle\mathit{inexact\text{-}special}_n\rangle$
$\langle\mathit{inexact\text{-}normal}_n\rangle$	::=	$\langle\mathit{inexact\text{-}simple}_n\rangle$ [$\langle\mathit{exp\text{-}mark}_n\rangle$ [$\langle\mathit{sign}\rangle$] $\langle\mathit{digits\#}_n\rangle$]
$\langle\mathit{inexact\text{-}simple}_n\rangle$	::=	$\langle\mathit{digits\#}_n\rangle$ [`.`] `#`*
	$\mid$	[$\langle\mathit{exact\text{-}integer}_n\rangle$] `.` $\langle\mathit{digits\#}_n\rangle$
	$\mid$	$\langle\mathit{digits\#}_n\rangle$ `/` $\langle\mathit{digits\#}_n\rangle$
$\langle\mathit{inexact\text{-}special}_n\rangle$	::=	`inf.0` $\mid$ `nan.0`
$\langle\mathit{digits\#}_n\rangle$	::=	$\langle\mathit{digit}_n\rangle^+$ `#`*
$\langle\mathit{inexact\text{-}complex}_n\rangle$	::=	[$\langle\mathit{inexact\text{-}real}_n\rangle$] $\langle\mathit{sign}\rangle$ $\langle\mathit{inexact\text{-}unsigned}_n\rangle$ `i`
	$\mid$	$\langle\mathit{inexact\text{-}real}_n\rangle$ `@` $\langle\mathit{inexact\text{-}real}_n\rangle$
$\langle\mathit{sign}\rangle$	::=	`+` $\mid$ `-`
$\langle\mathit{digit}_{16}\rangle$	::=	$\langle\mathit{digit}_{10}\rangle$ $\mid$ `a` $\mid$ `b` $\mid$ `c` $\mid$ `d` $\mid$ `e` $\mid$ `f`
$\langle\mathit{digit}_{10}\rangle$	::=	$\langle\mathit{digit}_8\rangle$ $\mid$ `8` $\mid$ `9`
$\langle\mathit{digit}_8\rangle$	::=	$\langle\mathit{digit}_2\rangle$ $\mid$ `2` $\mid$ `3` $\mid$ `4` $\mid$ `5` $\mid$ `6` $\mid$ `7`

$\langle digit_2 \rangle$ ::= `0` | `1`
$\langle exp\text{-}mark_{16} \rangle$ ::= `s` | `d` | `l`
$\langle exp\text{-}mark_{10} \rangle$ ::= $\langle exp\text{-}mark_{16} \rangle$ | `e` | `f`
$\langle exp\text{-}mark_{8} \rangle$ ::= $\langle exp\text{-}mark_{10} \rangle$
$\langle exp\text{-}mark_{2} \rangle$ ::= $\langle exp\text{-}mark_{10} \rangle$
$\langle general\text{-}number_n \rangle$ ::= [$\langle exactness \rangle$] $\langle number_n \rangle$
$\langle exactness \rangle$ ::= `#e` | `#i`

Examples:

`-1`	reads equal to	`-1`
`1/2`	reads equal to	`(/ 1 2)`
`1.0`	reads equal to	`(inexact->exact 1)`
`1+2i`	reads equal to	`(make-complex 1 2)`
`1/2+3/4i`	reads equal to	`(make-complex (/ 1 2) (/ 3 4))`
`1.0+3.0e7i`	reads equal to	`(inexact->exact (make-complex 1 30000000))`
`2e5`	reads equal to	`(inexact->exact 200000)`
`#i5`	reads equal to	`(inexact->exact 5)`
`#e2e5`	reads equal to	`200000`
`#x2e5`	reads equal to	`741`
`#b101`	reads equal to	`5`

12.6.4 Reading Booleans

A `#t` or `#T` is the complete input syntax for the boolean constant true, and `#f` or `#F` is the complete input syntax for the boolean constant false.

12.6.5 Reading Pairs and Lists

When the reader encounters a `(`, `[`, or `{`, it starts parsing a pair or list; see §3.9 “Pairs and Lists” for information on pairs and lists.

To parse the pair or list, the reader recursively reads data until a matching `)`, `]`, or `}` (respectively) is found, and it specially handles a delimited `.`. Pairs `()`, `[]`, and `{}` are treated the same way, so the remainder of this section simply uses “parentheses” to mean any of these pair.

If the reader finds no delimited `.` among the elements between parentheses, then it produces a list containing the results of the recursive reads.

If the reader finds two data between the matching parentheses that are separated by a delimited `.`, then it creates a pair. More generally, if it finds two or more data where the last is preceeded by a delimited `.`, then it constructs nested pairs: the next-to-last element is paired with the last, then the third-to-last is paired with that pair, and so on.

If the reader finds three or more data between the matching parentheses, and if a pair of delimited `.`s surrounds any other than the first and last elements, the result is a list containing the element surrounded by `.`s as the first element, followed by the others in the read order. This convention supports a kind of infix notation at the reader level.

In `read-syntax` mode, the recursive reads for the pair/list elements are themselves in `read-syntax` mode, so that the result is list or pair of syntax objects that it itself wrapped as a syntax object. If the reader constructs nested pairs because the input included a single delimited `.`, then only the innermost pair and outtermost pair are wrapped as syntax objects. Whether wrapping a pair or list, if the pair or list was formed with `[` and `]`, then a `'paren-shape` property is attached to the result with the value `#\[`;if the list or pair was formed with `{` and `}`, then a `'paren-shape` property is attached to the result with the value `#\{`.

If a delimited `.` appears in any other configuration, then the `exn:fail:read` exception is raised. Similarly, if the reader encounters a `)`, `]`, or `}` that does not end a list being parsed, then the `exn:fail:read` exception is raised.

Examples:

`()`	reads equal to	`(list)`
`(1 2 3)`	reads equal to	`(list 1 2 3)`
`{1 2 3}`	reads equal to	`(list 1 2 3)`
`[1 2 3]`	reads equal to	`(list 1 2 3)`
`(1 (2) 3)`	reads equal to	`(list 1 (list 2) 3)`
`(1 . 3)`	reads equal to	`(cons 1 3)`
`(1 . (3))`	reads equal to	`(list 1 3)`
`(1 . 2 . 3)`	reads equal to	`(list 2 1 3)`

If the `read-square-bracket-as-paren` parameter is set to `#f`, then when then reader encounters `[` and `]`, the `"exn:fail:read"` exception is raised. Similarly, If the `read-curly-brace-as-paren` parameter is set to `#f`, then when then reader encounters `{` and `}`, the `"exn:fail:read"` exception is raised.

If the `read-accept-dot` parameter is set to `#f`, then a delimited `"."` is not treated specially; it is instead parsed as a symbol. If the `read-accept-infix-dot` parameter is set to `#f`, then multiple delimited `.`s trigger a `exn:fail:read`, instead of the infix conversion.

12.6.6 Reading Strings

§3.4 "Strings (Unicode)" in §"**Guide**: PLT Scheme" introduces the syntax of strings.

When the reader encounters `"`, it begins parsing characters to form a string. The string continues until it is terminated by another `"` (that is not escaped by `\`).

Within a string sequence, the following escape sequences are recognized:

- `\a`: alarm (ASCII 7)

- `\b`: backspace (ASCII 8)
- `\t`: tab (ASCII 9)
- `\n`: linefeed (ASCII 10)
- `\v`: vertical tab (ASCII 11)
- `\f`: formfeed (ASCII 12)
- `\r`: return (ASCII 13)
- `\e`: escape (ASCII 27)
- `\"`: double-quotes (without terminating the string)
- `\'`: quote (i.e., the backslash has no effect)
- `\\`: backslash (i.e., the second is not an escaping backslash)
- `\`$\langle$*digit*$_8\rangle^{\{1,3\}}$: Unicode for the octal number specified by digit$_8{}^{\{1,3\}}$ (i.e., 1 to 3 $\langle$*digit*$_8\rangle$s) where each $\langle$*digit*$_8\rangle$ is `0`, `1`, `2`, `3`, `4`, `5`, `6`, or `7`. A longer form takes precedence over a shorter form, and the resulting octal number must be between 0 and 255 decimal, otherwise the `exn:fail:read` exception is raised.
- `\x`$\langle$*digit*$_{16}\rangle^{\{1,2\}}$: Unicode for the hexadecimal number specified by $\langle$*digit*$_{16}\rangle^{\{1,2\}}$, where each $\langle$*digit*$_{16}\rangle$ is `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `a`, `b`, `c`, `d`, `e`, or `f` (case-insensitive). The longer form takes precedence over the shorter form.
- `\u`$\langle$*digit*$_{16}\rangle^{\{1,4\}}$: like `\x`, but with up to four hexadecimal digits (longer sequences take precedence). The resulting hexadecimal number must be a valid argument to `integer->char`, otherwise the `exn:fail:read` exception is raised.
- `\U`$\langle$*digit*$_{16}\rangle^{\{1,8\}}$: like `\x`, but with up to eight hexadecimal digits (longer sequences take precedence). The resulting hexadecimal number must be a valid argument to `integer->char`, otherwise the `exn:fail:read` exception is raised.
- `\`$\langle$*newline*$\rangle$: elided, where $\langle$*newline*$\rangle$ is either a linefeed, carriage return, or carriage return–linefeed combination. This convetion allows single-line strings to span multiple lines in the source.

If the reader encounteres any other use of a backslash in a string constant, the `exn:fail:read` exception is raised.

§3.5 "Bytes and Byte Strings" in §"**Guide**: PLT Scheme" introduces the syntax of byte strings.

A string constant preceded by `#` is parsed as a byte-string. (That is, `#"` starts a byte-string literal.) See §3.4 "Byte Strings" for information on byte strings. Byte string constants support the same escape sequences as character strings, except `\u` and `\U`.

When the reader encounters `#<<`, it starts parsing a *here string*. The characters following `#<<` until a newline character define a terminator for the string. The content of the string includes all characters between the `#<<` line and a line whose only content is the specified terminator. More precisely, the content of the string starts after a newline following `#<<`, and it ends before a newline that is followed by the terminator, where the terminator is itself followed by either a newline or end-of-file. No escape sequences are recognized between the starting and terminating lines; all characters are included in the string (and terminator) literally. A return character is not treated as a line separator in this context. If no characters appear between `#<<` and a newline or end-of-file, or if an end-of-file is encountered before a terminating line, the `exn:fail:read` exception is raised.

Examples:

`"Apple"`	reads equal to	`"Apple"`
`"\x41pple"`	reads equal to	`"Apple"`
`"\"Apple\""`	reads equal to	`"\x22Apple\x22"`
`"\\"`	reads equal to	`"\x5C"`
`#"Apple"`	reads equal to	`(bytes 65 112 112 108 101)`

12.6.7 Reading Quotes

When the reader enounters `'`, then it recursively reads one datum, and it forms a new list containing the symbol `'quote` and the following datum. This convention is mainly useful for reading Scheme code, where `'s` can be used as a shorthand for `(quote s)`.

Several other sequences are recognized and transformed in a similar way. Longer prefixes take precedence over short ones:

`'`	adds	`quote`
`` ` ``	adds	`quasiquote`
`,`	adds	`unquote`
`,@`	adds	`unquote-splicing`
`#'`	adds	`syntax`
``#` ``	adds	`quasisyntax`
`#,`	adds	`unsyntax`
`#,@`	adds	`unsyntax-splicing`

Examples:

`'apple`	reads equal to	`(list 'quote 'apple)`
`` `(1 ,2) ``	reads equal to	`(list 'quasiquote (list 1 (list 'unquote 2)))`

The `` ` ``, `,`, and `,@` forms are disabled when the `read-accept-quasiquote` parameter is set to `#f`, in which case the `exn:fail:read` exception is raised, instead.

12.6.8 Reading Comments

A `;` starts a line comment. When the reader encounters `;`, then it skips past all characters until the next linefeed (ASCII 10), carriage return (ASCII 13), next-line (Unicode 133), line-separator (Unicode 8232), or line-separator (Uunicode 8232) character.

A `#|` starts a nestable block comment. When the reader encounters `#|`, then it skips past all characters until a closing `|#`. Pairs of matching `#|` and `|#` can be nested.

A `#;` starts an S-expression comment. Then the reader encounters `#;`, it recursively reads one datum, and then discards the datum (continuing on to the next datum for the read result).

A `#! ` (which is `#!` followed by a space) or `#!/` starts a line comment that can be continued to the next line by ending a line with `\`. This form of comment normally appears at the beginning of a Unix script file.

Examples:

`; comment`	reads equal to *nothing*
`#| a |# 1`	reads equal to `1`
`#| #| a |# 1 |# 2`	reads equal to `2`
`#;1 2`	reads equal to `2`
`#!/bin/sh`	reads equal to *nothing*
`#! /bin/sh`	reads equal to *nothing*

12.6.9 Reading Vectors

When the reader encounters a `#(`, `#[`, or `#{`, it starts parsing a vector; see §3.11 "Vectors" for information on vectors. The `#[` and `#{` forms can be disabled through the `read-square-bracket-as-paren` and `read-curly-brace-as-paren` parameters.

The elements of the vector are recursively read until a matching `)`, `]`, or `}` is found, just as for lists (see §12.6.5 "Reading Pairs and Lists"). A delimited `.` is not allowed among the vector elements.

An optional vector length can be specified between the `#` and `(`, `[`, or `{`. The size is specified using a sequence of decimal digits, and the number of elements provided for the vector must be no more than the specified size. If fewer elements are provided, the last provided element is used for the remaining vector slots; if no elements are provided, then `0` is used for all slots.

In `read-syntax` mode, each recursive read for the vector elements is also in `read-syntax` mode, so that the wrapped vector's elements are also wraped as syntax objects, and the vector is immutable.

Examples:

`#(1 apple 3)`	reads equal to	`(vector 1 'apple 3)`
`#3("apple" "banana")`	reads equal to	`(vector "apple" "banana" "banana")`
`#3()`	reads equal to	`(vector 0 0 0)`

12.6.10 Reading Structures

When the reader encounters a `#s(`, `#s[`, or `#s{`, it starts parsing an instance of a prefab structure type; see §4 "Structures" for information on structure types. The `#s[` and `#s{` forms can be disabled through the `read-square-bracket-as-paren` and `read-curly-brace-as-paren` parameters.

The elements of the structure are recursively read until a matching `)`, `]`, or `}` is found, just as for lists (see §12.6.5 "Reading Pairs and Lists"). A delimited `.` is not allowed among the elements.

The first element is used as the structure descriptor, and it must have the form (when quoted) of a possible argument to `make-prefab-struct`; in the simplest case, it can be a symbol. The remaining elements correspond to field values within the structure.

In `read-syntax` mode, the structure type must not have any mutable fields. The structure's elements are read in `read-syntax` mode, so that the wrapped structure's elements are also wraped as syntax objects.

If the first structure element is not a valid prefab structure type key, or if the number of provided fields is inconsistent with the indicated prefab structure type, the `exn:fail:read` exception is raised.

12.6.11 Reading Hash Tables

A `#hash` starts an immutable hash-table constant with key matching based on `equal?`. The characters after `hash` must parse as a list of pairs (see §12.6.5 "Reading Pairs and Lists") with a specific use of delimited `.`: it must appear between the elements of each pair in the list, and nowhere in the sequence of list elements. The first element of each pair is used as the key for a table entry, and the second element of each pair is the associated value.

A `#hasheq` starts a hash table like `#hash`, except that it constructs a hash table based on `eq?` instead of `equal?`.

In either case, the table is constructed by adding each mapping to the hash table from left to right, so later mappings can hide earlier mappings if the keys are equivalent.

Examples, where `make-...` stands for `make-immutable-hash`:

`#hash()`	reads equal to	`(make-... '())`
`#hasheq()`	reads equal to	`(make-...eq '())`
`#hash(("a" . 5))`	reads equal to	`(make-... '(("a" . 5)))`
`#hasheq((a . 5) (b . 7))`	reads equal to	`(make-...eq '((a . 5) (b . 7)))`
`#hasheq((a . 5) (a . 7))`	reads equal to	`(make-...eq '((a . 7)))`

12.6.12 Reading Boxes

When the reader encounters a `#&`, it starts parsing a box; see §3.12 "Boxes" for information on boxes. The content of the box is determined by recursively reading the next datum.

In `read-syntax` mode, the recursive read for the box content is also in `read-syntax` mode, so that the wrapped box's content is also wraped as a syntax object, and the box is immutable.

Examples:
`#&17` reads equal to `(box 17)`

12.6.13 Reading Characters

§3.3 "Characters" in §"**Guide**: PLT Scheme" introduces the syntax of characters.

A `#\` starts a character constant, which has one of the following forms:

- `#\nul` or `#\null`: NUL (ASCII 0); the next character must not be alphabetic.
- `#\backspace`: backspace (ASCII 8); the next character must not be alphabetic.
- `#\tab`: tab (ASCII 9); the next character must not be alphabetic.
- `#\newline` or `#\linefeed`: linefeed (ASCII 10); the next character must not be alphabetic.
- `#\vtab`: vertical tab (ASCII 11); the next character must not be alphabetic.
- `#\page`: page break (ASCII 12); the next character must not be alphabetic.
- `#\return`: carriage return (ASCII 13); the next character must not be alphabetic.
- `#\space`: space (ASCII 32); the next character must not be alphabetic.
- `#\rubout`: delete (ASCII 127); the next character must not be alphabetic.
- `#\`$\langle digit_8 \rangle^{\{1,3\}}$: Unicode for the octal number specified by $\langle digit_8 \rangle^{\{1,3\}}$, as in string escapes (see §12.6.6 "Reading Strings").
- `#\x`$\langle digit_{16} \rangle^{\{1,2\}}$: Unicode for the hexadecimal number specified by $\langle digit_{16} \rangle^{\{1,2\}}$, as in string escapes (see §12.6.6 "Reading Strings").
- `#\u`$\langle digit_{16} \rangle^{\{1,4\}}$: like `#\x`, but with up to four hexadecimal digits.

- `#\U`$\langle digit_{16}\rangle^{\{1,6\}}$: like `#\x`, but with up to six hexadecimal digits.
- `#\`$\langle c\rangle$: the character $\langle c\rangle$, as long as `#\`$\langle c\rangle$ and the characters following it do not match any of the previous cases, and as long as the character after $\langle c\rangle$ is not alphabetic.

Examples:

`#\newline`	reads equal to	`(integer->char 10)`
`#\n`	reads equal to	`(integer->char 110)`
`#\u3BB`	reads equal to	`(integer->char 955)`
`#\λ`	reads equal to	`(integer->char 955)`

12.6.14 Reading Keywords

A `#:` starts a keyword. The parsing of a keyword after the `#:` is the same as for a symbol, including case-folding in case-insensitive mode, except that the part after `#:` is never parsed as a number.

Examples:

`#:Apple`	reads equal to	`(string->keyword "#:Apple")`
`#:1`	reads equal to	`(string->keyword "#:1")`

12.6.15 Reading Regular Expressions

A `#rx` or `#px` starts a regular expression. The characters immediately after `#rx` or `#px` must parse as a string or byte string (see §12.6.6 "Reading Strings"). A `#rx` prefix starts a regular expression as would be constructed by `regexp`, `#px` as constructed by `pregexp`, `#rx#` as constructed by `byte-regexp`, and `#px#` as constructed by `byte-pregexp`.

Examples:

`#rx".*"`	reads equal to	`(regexp ".*")`
`#px"[\\s]*"`	reads equal to	`(pregexp "[\\s]*")`
`#rx#".*"`	reads equal to	`(byte-regexp #".*")`
`#px#"[\\s]*"`	reads equal to	`(byte-pregexp #"[\\s]*")`

12.6.16 Reading Graph Structure

A `#`$\langle digit_{10}\rangle^{\{1,8\}}$`=` tags the following datum for reference via `#`$\langle digit_{10}\rangle^{\{1,8\}}$`#`, which allows the reader to produce a datum that have graph structure.

For a specific $\langle digit_{10}\rangle^{\{1,8\}}$ in a single read result, each `#`$\langle digit_{10}\rangle^{\{1,8\}}$`#` reference is replaced by the datum read for the corresponding `#`$\langle digit_{10}\rangle^{\{1,8\}}$`=`; the definition `#`$\langle digit_{10}\rangle^{\{1,8\}}$`=` also produces just the datum after it. A `#`$\langle digit_{10}\rangle^{\{1,8\}}$`=` definition can appear at most once, and a `#`$\langle digit_{10}\rangle^{\{1,8\}}$`=` definition must appear before a `#`$\langle digit_{10}\rangle^{\{1,8\}}$`#` reference appears, otherwise the `exn:fail:read` exception is raised. If the `read-accept-graph` parameter is set to `#f`, then `#`$\langle digit_{10}\rangle^{\{1,8\}}$`=` or `#`$\langle digit_{10}\rangle^{\{1,8\}}$`#` triggers a `exn:fail:read` exception.

Although a comment parsed via `#;` discards the datum afterward, `#`$\langle digit_{10}\rangle^{\{1,8\}}$`=` definitions in the discarded datum still can be referenced by other parts of the reader input, as long as both the comment and the reference are grouped together by some other form (i.e., some recursive read); a top-level `#;` comment neither defines nor uses graph tags for other top-level forms.

Examples:

`(#1=100 #1# #1#)`	reads equal to	`(list 100 100 100)`
`#0=(1 . #0#)`	reads equal to	`(let ([v (cons 1 #f)])` `  (set-cdr! v v) v)`

12.6.17 Reading via an Extension

When the reader encounters `#reader`, then it loads an external reader procedure and applies it to the current input stream.

The reader recursively reads the next datum after `#reader`, and passes it to the procedure that is the value of the `current-reader-guard` parameter; the result is used as a module path. The module path is passed to `dynamic-require` with either `'read` or `'read-syntax` (depending on whether the reader is in `read` or `read-syntax` mode).

The arity of the resulting procedure determines whether it accepts extra source-location information: a `read` procedure accepts either one argument (an input port) or five, and a `read-syntax` procedure accepts either two arguments (a name value and an input port) or six. In either case, the four optional arguments are the module path (as a syntax object in `read-syntax` mode) followed by the line (positive exact integer or `#f`), column (non-negative exact integer or `#f`), and position (positive exact integer or `#f`) of the start of the `#reader` form. The input port is the one whose stream contained `#reader`, where the stream position is immediately after the recursively-read module path.

The procedure should produce a datum result. If the result is a syntax object in `read` mode, then it is converted to a datum using `syntax->datum`; if the result is not a syntax object in `read-syntax` mode, then it is converted to one using `datum->syntax`. See also §12.9.2 "Reader-Extension Procedures" for information on the procedure's results.

If the `read-accept-reader` parameter is set to `#f`, then if the reader encounters `#reader`, the `exn:fail:read` exception is raised.

§6.2.2 "The `#lang` Shorthand" in §"**Guide**: PLT Scheme" introduces `#lang`.

The `#lang` reader form is similar to `#reader`, but more constrained: the `#lang` must be followed by a single space (ASCII 32), and then a non-empty sequence of alphanumeric ASCII, `+`, `-`, `_`, and/or `/` characters terminated by whitespace or an end-of-file. The sequence must not start or end with `/`. A sequence `#lang` $\langle name\rangle$ is equivalent to

`#reader` ⟨*name*⟩`/lang/reader`, except that the terminating whitespace (if any) is consumed before the external reading procedure is called.

Finally, `#!` followed by alphanumeric ASCII, `+`, `-`, or `_` is a synonym for `#lang` followed by a space. Use of this synonym is discourage except as needed to construct programs that conform to certain grammars, such as that of R^{6}RS [Sperber07].

By convention, `#lang` normally appears at the beginning of a file, possibly after comment forms, to specify the syntax of a module.

12.6.18 Honu Parsing

See §"**Honu**" for information on `#hx` and `#sx`.

12.7 The Printer

The default printer generally prints core datatypes in such a way that using `read` on the output produces a value that is `equal?` to the printed value—when the printed is used in `write`. When the printer is used in `display` mode, the printing of strings, byte strings, characters, and symbols changes to render the character/byte content directly to the output port.

When the `print-graph` parameter is set to `#t`, then the printer first scans an object to detect cycles. The scan traverses the components of pairs, mutable pairs, vectors, boxes (when `print-box` is `#t`), hash tables (when `print-hash-table` is `#t`), and fields of structures exposed by `struct->vector` (when `print-struct` is `#t`). If `print-graph` is `#t`, then this information is used to display sharing by printing graph definitions and references (see §12.6.16 "Reading Graph Structure"). If a cycle is detected in the initial scan, then `print-graph` is effectively set to `#t` automatically.

With the exception of displaying byte strings, printing is defined in terms of Unicode characters; see §12.1 "Ports" for information on how a character stream is written to an port's underlying byte stream.

12.7.1 Printing Symbols

Symbols containing spaces or special characters `write` using escaping `\` and quoting `|`s. When the `read-case-sensitive` parameter is set to `#f`, then symbols containing uppercase characters also use escaping escaping `\` and quoting `|`s. In addition, symbols are quoted with `|`s or leading `\` when they would otherwise print the same as a numerical constant or as a delimited `"."` (when `read-accept-dot` is `#t`).

When read-accept-bar-quote is #t, |s are used in printing when one | at the beginning and one | at the end suffices to correctly print the symbol. Otherwise, \s are always used to escape special characters, instead of quoting them with |s.

When read-accept-bar-quote is #f, then | is not treated as a special character. The following are always special characters:

`( ) [ ] [ ] " , ' ` ; \`

In addition, # is a special character when it appears at the beginning of the symbol, and when it is not followed by %.

Symbols display without escaping or quoting special characters. That is, the display form of a symbol is the same as the display form of symbol->string applied to the symbol.

12.7.2 Printing Numbers

A number prints the same way in write and display modes.

A complex number that is not a real number always prints as ⟨*m*⟩+⟨*n*⟩i, where ⟨*m*⟩ and ⟨*n*⟩ are the printed forms of its real and imaginary parts, respectively.

An inexact real number prints with either a . decimal point, an e exponent marker, or both. The form is selected so that the output is as short as possible, with the constraint that reading the printed form back in produces an "equal?" number.

An exact 0 prints as 0.

A positive, exact integer prints as a sequence of decimal digits that does not start with 0.

A positive, exact, real, non-integer number prints as ⟨*m*⟩/⟨*n*⟩, where ⟨*m*⟩ and ⟨*n*⟩ are the printed forms of the number's numerators and denominator (as determined by numerator and denominator).

A negative exact number prints with a - prefix on the printed form of its exact negation.

12.7.3 Printing Booleans

The constant #t prints as #t, and the constant #f prints as #f in both display and write modes.

12.7.4 Printing Pairs and Lists

A pair prints starting with `(` followed by the printed form of its `car`. The rest of the printed form depends on the `cdr`:

- If the `cdr` is a pair or the empty list, then the printed form of the pair completes with the printed form of the `cdr`, except that the leading `(` in the `cdr`'s printed form is omitted.
- Otherwise, the printed for of the pair continues with a space, `.`, another space, the printed form of the `cdr`, and a `)`.

The printed form of a pair is the same in both `write` and `display` modes, except as the printed form of the pair's `car`and `cdr` vary with the mode.

By default, mutable pairs (as created with `mcons`) print the same as pairs, except that `{` and `}` are used instead of `(` and `)`. Note that the reader treats `{`...`}` and `(`...`)` equivalently on input, creating immutable pairs in both cases.

If the `print-pair-curly-braces` parameter is set to `#t`, then immutable pairs print using `{` and `}`. If the `print-mpair-curly-braces` parameter is set to `#f`, then mutable pairs print using `(` and `)`.

12.7.5 Printing Strings

All strings `display` as their literal character sequences.

The `write` form of a string starts with `"` and ends with another `"`. Between the `"`s, each character is represented. Each graphic or blank character is represented as itself, with two exceptions: `"` is printed as `\"`, and `\` is printed as `\\`. Each non-graphic, non-blank character (according to `char-graphic?` and `char-blank?`) is printed using the escape sequences described in §12.6.6 "Reading Strings", using `\a`, `\b`, `\t`, `\n`, `\v`, `\f`, `\r`, or `\e` if possible, otherwise using `\u` with four hexadecimal digits or `\U` with eight hexadecimal digits (using the latter only if the character value does not fit into four digits).

All byte strings `display` as their literal byte sequence; this byte sequence may not be a valid UTF-8 encoding, so it may not correspond to a sequence of characters.

The `write` form a byte string starts with `#"` and ends with another `"`. Between the `"`s, each byte is written using the corresponding ASCII decoding if the byte is between 0 and 127 and the character is graphic or blank (according to `char-graphic?` and `char-blank?`). Otherwise, the byte is written using `\a`, `\b`, `\t`, `\n`, `\v`, `\f`, `\r`, or `\e` if possible, otherwise using `\` followed by one to three octal digits (only as many as necessary).

12.7.6 Printing Vectors

In `display` mode, the printed form of a vector is `#` followed by the printed form of `vector->list` applied to the vector. In `write` mode, the printed form is the same, except that when the `print-vector-length` parameter is `#t`, a decimal integer is printed after the `#`, and a repeated last element is printed only once..

12.7.7 Printing Structures

When the `print-struct` parameter is set to `#t`, then the way that structures print depends on details of the structure type for which the structure is an instance:

- If the structure type is a prefab structure type, then it prints using `#s(` followed by the prefab structure type key, then the printed form each field in the structure, and then `)`.
- If the structure has a `prop:custom-write` property value, then the associated procedure is used to print the structure.
- If the structure type is transparent, or if any ancestor is transparent, then the structure prints as the vector produced by `struct->vector`.
- For any other structure type, the structure prints as an unreadable value; see §12.7.13 "Printing Unreadable Values" for more information.

If the `print-struct` parameter is set to `#f`, then all structures without a `prop:custom-write` property print as unreadable values.

12.7.8 Printing Hash Tables

When the `print-hash-table` parameter is set to `#t`, a hash table prints starting with `#hash(` or `#hasheq(` for a table using `equal?` or `eq?` key comparisons, respectively. After this prefix, each key–value mapping is shown as `(`, the printed form of a key, a space, `.`, a space, the printed form the corresponding value, and `)`, with an addition space if the key–value pairs is not the last to be printed. After all key-value pairs, the printed form completes with `)`.

When the `print-hash-table` parameter is set to `#f`, a hash table prints (un`read`ably) as `#<hash>`.

12.7.9 Printing Boxes

When the `print-box` parameter is set to `#t`, a box prints as `#&` followed by the printed form of its content.

When the `print-box` parameter is set to `#f`, a box prints (unreadably) as `#<box>`.

12.7.10 Printing Characters

Characters with the special names described in §12.6.13 "Reading Characters" `write` using the same name. (Some characters have multiple names; the `#\newline` and `#\nul` names are used instead of `#\newline` and `#\nul`). Other graphic characters (according to `char-graphic?`) `write` as `#\` followed by the single character, and all others characters are written in `#\u` notation with four digits or `#\U` notation with eight digits (using the latter only if the character value it does not fit in four digits).

All characters `display` directly as themselves (i.e., a single character).

12.7.11 Printing Keywords

Keywords `write` and `display` the same as symbols, except (see §12.7.1 "Printing Symbols") with a leading `#:`, and without special handing for an initial `#` or when the printed form would matches a number or a delimited `.` (since `#:` distinguishes the keyword).

12.7.12 Printing Regular Expressions

Regexp values in both `write` and `display` mode print starting with `#px` (for `pregexp`-based regexps) or `#rx` (for `regexp`-based regexps) followed by the `write` form of the regexp's source string or byte string.

12.7.13 Printing Unreadable Values

For any value with no other printing specification, the output form is `#<`⟨*something*⟩`>`, where ⟨*something*⟩ is specific to the type of the value and sometimes to the value itself.

12.8 Pretty Printing

```
(require scheme/pretty)
```

The bindings documented in this section are provided by the `scheme/pretty` and `scheme` libraries, but not `scheme/base`.

```
(pretty-print v [port]) → void?
  v : any/c
  port : output-port? = (current-output-port)
```

Pretty-prints the value *v* using the same printed form as `write`, but with newlines and whitespace inserted to avoid lines longer than `(pretty-print-columns)`, as controlled by `(pretty-print-current-style-table)`. The printed form ends in a newline, unless the `pretty-print-columns` parameter is set to `'infinity`.

In addition to the parameters defined in this section, `pretty-print` conforms to the `print-graph`, `print-struct`, `print-hash-table`, `print-vector-length`, and `print-box` parameters.

The pretty printer also detects structures that have the `prop:custom-write` property and it calls the corresponding custom-write procedure. The custom-write procedure can check the parameter `pretty-printing` to cooperate with the pretty-printer. Recursive printing to the port automatically uses pretty printing, but if the structure has multiple recursively printed sub-expressions, a custom-write procedure may need to cooperate more to insert explicit newlines. Use `port-next-location` to determine the current output column, use `pretty-print-columns` to determine the target printing width, and use `pretty-print-newline` to insert a newline (so that the function in the `pretty-print-print-line` parameter can be called appropriately). Use `make-tentative-pretty-print-output-port` to obtain a port for tentative recursive prints (e.g., to check the length of the output).

```
(pretty-display v [port]) → void?
  v : any/c
  port : output-port? = (current-output-port)
```

Same as `pretty-print`, but *v* is printed like `display` instead of like `write`.

```
(pretty-format v [columns]) → string?
  v : any/c
  columns : nonnegative-exact-integer? = (pretty-print-columns)
```

Like `pretty-print`, except that it returns a string containing the pretty-printed value, rather than sending the output to a port.

The optional argument *columns* argument is used to parameterize `pretty-print-columns`.

```
(pretty-print-handler v) → void?
  v : any/c
```

Pretty-prints `v` if `v` is not `#<void>`, or prints nothing if `v` is `#<void>`. Pass this procedure to `current-print` to install the pretty printer into the REPL run by `read-eval-print-loop`.

12.8.1 Basic Pretty-Print Options

```
(pretty-print-columns)
 → (or/c positive-exact-integer? (one-of/c 'infinity))
(pretty-print-columns width) → void?
  width : (or/c positive-exact-integer? (one-of/c 'infinity))
```

A parameter that determines the default width for pretty printing.

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)
 → (or/c nonnegative-exact-integer? false/c)
(pretty-print-depth depth) → void?
  depth : (or/c nonnegative-exact-integer? false/c)
```

Parameter that controls the default depth for recursive pretty printing. Printing to `depth` means that elements nested more deeply than `depth` are replaced with "..."; in particular, a depth of 0 indicates that only simple values are printed. A depth of `#f` (the default) allows printing to arbitrary depths.

```
(pretty-print-exact-as-decimal) → boolean?
(pretty-print-exact-as-decimal as-decimal?) → void?
  as-decimal? : any/c
```

A parameter that determines how exact non-integers are printed. If the parameter's value is `#t`, then an exact non-integer with a decimal representation is printed as a decimal number instead of a fraction. The initial value is `#f`.

```
(pretty-print-.-symbol-without-bars) → boolean?
(pretty-print-.-symbol-without-bars on?) → void?
  on? : any/c
```

A parameter that controls the printing of the symbol whose print name is just a period. If set to a true value, then such a symbol is printed as only the period. If set to a false value, it is printed as a period with vertical bars surrounding it.

```
(pretty-print-show-inexactness) → boolean?
(pretty-print-show-inexactness show?) → void?
  show? : any/c
```

A 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`.

12.8.2 Per-Symbol Special Printing

```
(pretty-print-abbreviate-read-macros) → boolean?
(pretty-print-abbreviate-read-macros abbrev?) → void?
  abbrev? : any/c
```

A parameter that controls whether or not `quote`, `unquote`, `unquote-splicing`, etc. are abbreviated with `'`, `,`, `,@`, etc. By default, the abbreviations are enabled.

```
(pretty-print-style-table? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a style table for use with `pretty-print-current-style-table`, `#f` otherwise.

```
(pretty-print-current-style-table) → pretty-print-style-table?
(pretty-print-current-style-table style-table) → void?
  style-table : pretty-print-style-table?
```

A parameter that holds a table of style mappings. See `pretty-print-extend-style-table`.

```
(pretty-print-extend-style-table style-table
                                 symbol-list
                                 like-symbol-list)
 → pretty-print-style-table?
  style-table : pretty-print-style-table?
  symbol-list : (listof symbol?)
  like-symbol-list : (listof symbol?)
```

Creates a new style table by extending an existing *style-table*, so that the style mapping for each symbol of *like-symbol-list* in the original table is used for the corresponding symbol of *symbol-list* in the new table. The *symbol-list* and *like-symbol-list* lists must have the same length. The *style-table* argument can be `#f`, in which case with default mappings are used for the original table (see below).

The style mapping for a symbol controls the way that whitespace is inserted when printing a list that starts with the symbol. In the absence of any mapping, when a list is broken across multiple lines, each element of the list is printed on its own line, each with the same indentation.

The default style mapping includes mappings for the following symbols, so that the output follows popular code-formatting rules:

```
'lambda 'case-lambda
'define 'define-macro 'define-syntax
'let letrec 'let*
'let-syntax 'letrec-syntax
'let-values 'letrec-values 'let*-values
'let-syntaxes 'letrec-syntaxes
'begin 'begin0 'do
'if 'set! 'set!-values
'unless 'when
'cond 'case 'and 'or
'module
'syntax-rules 'syntax-case 'letrec-syntaxes+values
'import 'export 'link
'require 'require-for-syntax 'require-for-template 'provide
'public 'private 'override 'rename 'inherit 'field 'init
'shared 'send 'class 'instantiate 'make-object
```

```
(pretty-print-remap-stylable)
 → (any/c . -> . (or/c symbol? false/c))
(pretty-print-remap-stylable proc) → void?
  proc : (any/c . -> . (or/c symbol? false/c))
```

A parameter that controls remapping for styles. This procedure is called with each subexpression that appears as the first element in a sequence. If it returns a symbol, the style table is used, as if that symbol were at the head of the sequence. If it returns `#f`, the style table is treated normally.

12.8.3 Line-Output Hook

```
(pretty-print-newline port width) → void?
  port : out-port?
  width : nonnegative-exact-integer?
```

Calls the procedure associated with the `pretty-print-print-line` parameter to print a newline to *port*, if *port* is the output port that is redirected to the original output port for printing, otherwise a plain newline is printed to *port*. The *width* argument should be the target column width, typically obtained from `pretty-print-columns`.

```
(pretty-print-print-line)
 → ((or/c nonnegative-exact-integer? false/c)
     output-port?
     nonnegative-exact-integer?
     (or/c nonnegative-exact-integer? (one-of/c 'infinity))
     . -> .
     nonnegative-exact-integer?)
(pretty-print-print-line proc) → void?
  proc : ((or/c nonnegative-exact-integer? false/c)
          output-port?
          nonnegative-exact-integer?
          (or/c nonnegative-exact-integer? (one-of/c 'infinity))
          . -> .
          nonnegative-exact-integer?)
```

A parameter that determines a procedure for printing the newline separator between lines of a pretty-printed value. The 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 *proc* is the number of extra characters it printed at the beginning of the new line.

The *proc* procedure is called before any characters are printed with `0` as the line number and `0` as the old line length; *proc* 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, *proc* 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 *proc* is always the total width of the destination printing area, or `'infinity` if pretty-printed values are not broken into lines.

The default *proc* procedure prints a newline whenever the line number is not `0` and the column count is not `'infinity`, always returning `0`. A custom *proc* 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 *proc* so that the next line break can be chosen correctly.

The destination port supplied to *proc* is generally not the port supplied to `pretty-print` or `pretty-display` (or the current output port), but output to this port is ultimately redirected to the port supplied to `pretty-print` or `pretty-display`.

12.8.4 Value Output Hook

```
(pretty-print-size-hook)
 → (any/c boolean? output-port . -> . (or/c false/c nonnegative-exact-integer?))
(pretty-print-size-hook proc) → void?
  proc : (any/c boolean? output-port . -> . (or/c false/c nonnegative-exact-integer?))
```

A parameter that determines a sizing hook for pretty-printing.

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 port is the one supplied to `pretty-print` or `pretty-display` (or the current output port). The sizing hook may be applied to a single value multiple times during pretty-printing.

```
(pretty-print-print-hook)
 → (any/c boolean? output-port . -> . void?)
(pretty-print-print-hook proc) → void?
  proc : (any/c boolean? output-port . -> . void?)
```

A parameter that determines a print hook for pretty-printing. The print-hook procedure 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; this port is generally not the port supplied to `pretty-print` or `pretty-display` (or the current output port), but output to this port is ultimately redirected to the port supplied to `pretty-print` or `pretty-display`.

```
(pretty-print-pre-print-hook)
 → (any/c output-port? . -> . void)
(pretty-print-pre-print-hook proc) → void?
  proc : (any/c output-port? . -> . void)
```

A parameter that determines a hook procedure to be called just before an object is printed. The hook receives two arguments: the object and the output port. The port is the one supplied to `pretty-print` or `pretty-display` (or the current output port).

```
(pretty-print-post-print-hook)
 → (any/c output-port? . -> . void)
(pretty-print-post-print-hook proc) → void?
  proc : (any/c output-port? . -> . void)
```

A parameter that determines a hook procedure to be called just after an object is printed. The hook receives two arguments: the object and the output port. The port is the one supplied to `pretty-print` or `pretty-display` (or the current output port).

12.8.5 Additional Custom-Output Support

```
(pretty-printing) → boolean?
(pretty-printing on?) → void?
  on? : any/c
```

A parameter that is set to `#t` when the pretty printer calls a custom-write procedure (see `prop:custom-write`) for output. When pretty printer calls a custom-write procedure merely to detect cycles, it sets this parameter to `#f`.

```
(make-tentative-pretty-print-output-port out
                                         width
                                         overflow-thunk)
 → output-port?
  out : output-port?
  width : nonnegative-exact-integer?
  overflow-thunk : (-> any)
```

Produces an output port that is suitable for recursive pretty printing without actually producing output. Use such a port to tentatively print when proper output depends on the size of recursive prints. Determine the size of the tentative print using `port-count-lines`.

The *out* argument should be a pretty-printing port, such as the one supplied to a custom-write procedure when `pretty-printing` is set to true, or another tentative output port. The *width* argument should be a target column width, usually obtained from `pretty-print-column-count`, possibly decremented to leave room for a terminator. The *overflow-thunk* procedure is called if more than *width* items are printed to the port; it can escape from the recursive print through a continuation as a short cut, but *overflow-thunk* can also return, in which case it is called every time afterward that additional output is written to the port.

After tentative printing, either accept the result with `tentative-pretty-print-port-transfer` or reject it with `tentative-pretty-print-port-cancel`. Failure to accept or cancel properly interferes with graph-structure printing, calls to hook procedures, etc. Explicitly cancel the tentative print even when *overflow-thunk* escapes from a recursive print.

```
(tentative-pretty-print-port-transfer tentative-out
                                      orig-out)      → void?
  tentative-out : output-port?
  orig-out : output-port?
```

Causes the data written to *tentative-out* to be transferred as if written to *orig-out*. The *tentative-out* argument should be a port produced by `make-tentative-pretty-print-output-port`, and *orig-out* should be either a pretty-printing port (provided to a custom-write procedure) or another tentative output port.

```
(tentative-pretty-print-port-cancel tentative-out) → void?
  tentative-out : output-port?
```

Cancels the content of *tentative-out*, which was produced by `make-tentative-pretty-print-output-port`. The main effect of canceling is that graph-reference definitions are undone, so that a future print of a graph-referenced object includes the defining `#`⟨*n*⟩`=`.

12.9 Reader Extension

Scheme's reader can be extended in three ways: through a reader-macro procedure in a readtable (see §12.9.1 "Readtables"), through a `#reader` form (see §12.6.17 "Reading via an Extension"), or through a custom-port byte reader that returns a "special" result procedure (see §12.1.9 "Custom Ports"). All three kinds of *reader extension procedures* accept similar arguments, and their results are treated in the same way by `read` and `read-syntax` (or, more precisely, by the default read handler; see `port-read-handler`).

12.9.1 Readtables

The dispatch table in §12.6.1 "Delimiters and Dispatch" corresponds to the default *readtable*. By creating a new readtable and installing it via the `current-readtable` parameter, the reader's behavior can be extended.

A readtable is consulted at specific times by the reader:

- when looking for the start of a datum;

- when determining how to parse a datum that starts with `#`;
- when looking for a delimiter to terminate a symbol or number;
- when looking for an opener (such as `(`), closer (such as `)`), or `.` after the first character parsed as a sequence for a pair, list, vector, or hash table; or
- when looking for an opener after `#`⟨*n*⟩ in a vector of specified length ⟨*n*⟩.

The readtable is ignored at other times. In particular, after parsing a character that is mapped to the default behavior of `;`, the readtable is ignored until the comment's terminating newline is discovered. Similarly, the readtable does not affect string parsing until a closing double-quote is found. Meanwhile, if a character is mapped to the default behavior of `(`, then it starts sequence that is closed by any character that is mapped to a close parenthesis `)`. An apparent exception is that the default parsing of `|` quotes a symbol until a matching character is found, but the parser is simply using the character that started the quote; it does not consult the readtable.

For many contexts, `#f` identifies the default readtable. In particular, `#f` is the initial value for the `current-readtable` parameter, which causes the reader to behave as described in §12.6 "The Reader".

```
(readtable? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a readtable, `#f` otherwise.

```
(make-readtable readtable
                key
                mode
                action ...+) → readtable?
  readtable : readtable?
  key : (or/c character? false/c)
  mode : (or/c (one-of 'terminating-macro
                       'non-terminating-macro
                       'dispatch-macro)
               character?)
  action : (or/c procedure?
                 readtable?)
```

Creates a new readtable that is like *readtable* (which can be `#f`), except that the reader's behavior is modified for each *key* according to the given *mode* and *action*. The `...+` for `make-readtable` applies to all three of *key*, *mode*, and *action*; in other words, the total number of arguments to `make-readtable` must be 1 modulo 3.

The possible combinations for `key`, `mode`, and `action` are as follows:

- `char 'terminating-macro proc` — causes `char` to be parsed as a delimiter, and an unquoted/uncommented `char` in the input string triggers a call to the *reader macro* `proc`; the activity of `proc` is described further below. Conceptually, characters like `;`, `(`, and `)` are mapped to terminating reader macros in the default readtable.

- `char 'non-terminating-macro proc` — like the `'terminating-macro` variant, but `char` is not treated as a delimiter, so it can be used in the middle of an identifier or number. Conceptually, `#` is mapped to a non-terminating macro in the default readtable.

- `char 'dispatch-macro proc` — like the `'non-terminating-macro` variant, but for `char` only when it follows a `#` (or, more precisely, when the character follows one that has been mapped to the behavior of `#`hash in the default readtable).

- `char like-char readtable` — causes `char` to be parsed in the same way that `like-char` is parsed in `readtable`, where `readtable` can be `#f` to indicate the default readtable. Mapping a character to the same actions as `|` in the default reader means that the character starts quoting for symbols, and the same character terminates the quote; in contrast, mapping a character to the same action as a `"` means that the character starts a string, but the string is still terminated with a closing `"`. Finally, mapping a character to an action in the default readtable means that the character's behavior is sensitive to parameters that affect the original character; for example, mapping a character to the same action is a curly brace `{` in the default readtable means that the character is disallowed when the `read-curly-brace-as-paren` parameter is set to `#f`.

- `#f 'non-terminating-macro proc` — replaces the macro used to parse characters with no specific mapping: i.e., characters (other than `#` or `|`) that can start a symbol or number with the default readtable.

If multiple `'dispatch-macro` mappings are provided for a single `char`, all but the last one are ignored. Similarly, if multiple non-`'dispatch-macro` mappings are provided for a single `char`, all but the last one are ignored.

A reader macro `proc` must accept six arguments, and it can optionally accept two arguments. The first two arguments are always the character that triggered the reader macro and the input port for reading. When the reader macro is triggered by `read-syntax` (or `read-syntax/recursive`), the procedure is passed four additional arguments that represent a source location. When the reader macro is triggered by `read` (or `read/recursive`), the procedure is passed only two arguments if it accepts two arguments, otherwise it is passed six arguments where the last four are all `#f`. See §12.9.2 "Reader-Extension Procedures" for information on the procedure's results.

A reader macro normally reads characters from the given input port to produce a value to be used as the "reader macro-expansion" of the consumed characters. The reader macro might produce a special-comment value (see §12.9.3 "Special Comments") to cause the consumed character to be treated as whitespace, and it might use `read/recursive` or `read-syntax/recursive`.

```
(readtable-mapping readtable char)
 → (or/c character?
         (one-of 'terminating-macro
                 'non-terminating-macro))
   (or/c false/c procedure?)
   (or/c false/c procedure?)
  readtable : readtable?
  char : character?
```

Produces information about the mappings in *readtable* for *char*. The result is three values:

- either a character (mapping is to same behavior as the character in the default readtable), `'terminating-macro`, or `'non-terminating-macro`; this result reports the main (i.e., non-`'dispatch-macro`) mapping for `key`. When the result is a character, then `key` is mapped to the same behavior as the returned character in the default readtable.
- either `#f` or a reader-macro procedure; the result is a procedure when the first result is `'terminating-macro` or `'non-terminating-macro`.
- either `#f` or a reader-macro procedure; the result is a procedure when the character has a `'dispatch-macro` mapping in *readtable* to override the default dispatch behavior.

Note that reader-macro procedures for the default readtable are not directly accessible. To invoke default behaviors, use `read/recursive` or `read-syntax/recursive` with a character and the `#f` readtable.

Examples:

```
 ; Provides raise-read-error and raise-read-eof-error
 > (require syntax/readerr)
 > (define (skip-whitespace port)
     ; Skips whitespace characters, sensitive to the current
     ; readtable's definition of whitespace
     (let ([ch (peek-char port)])
       (unless (eof-object? ch)
         ; Consult current readtable:
         (let-values ([(like-ch/sym proc dispatch-proc)
                       (readtable-mapping (current-readtable) ch)])
           ; If like-ch/sym is whitespace, then ch is whitespace
```

```
        (when (and (char? like-ch/sym)
                   (char-whitespace? like-ch/sym))
          (read-char port)
          (skip-whitespace port))))))
> (define (skip-comments read-one port src)
    ; Recursive read, but skip comments and detect EOF
    (let loop ()
      (let ([v (read-one)])
        (cond
         [(special-comment? v) (loop)]
         [(eof-object? v)
          (let-values ([(l c p) (port-next-location port)])
            (raise-read-eof-error
             "unexpected EOF in tuple" src l c p 1))]
         [else v]))))
> (define (parse port read-one src)
    ; First, check for empty tuple
    (skip-whitespace port)
    (if (eq? #\> (peek-char port))
        null
        (let ([elem (read-one)])
          (if (special-comment? elem)
              ; Found a comment, so look for > again
              (parse port read-one src)
              ; Non-empty tuple:
              (cons elem
                    (parse-nonempty port read-one src))))))
> (define (parse-nonempty port read-one src)
    ; Need a comma or closer
    (skip-whitespace port)
    (case (peek-char port)
      [(#\>) (read-char port)
       ; Done
       null]
      [(#\,) (read-char port)
       ; Read next element and recur
       (cons (skip-comments read-one port src)
             (parse-nonempty port read-one src))]
      [else
       ; Either a comment or an error; grab location (in case
       ; of error) and read recursively to detect comments
       (let-values ([(l c p) (port-next-location port)]
                    [(v) (read-one)])
         (cond
          [(special-comment? v)
           ; It was a comment, so try again
```

```
            (parse-nonempty port read-one src)]
           [else
            ; Wasn't a comment, comma, or closer; error
            ((if (eof-object? v)
                 raise-read-eof-error
                 raise-read-error)
             "expected `,' or `>'" src l c p 1)]))]))
> (define (make-delims-table)
    ; Table to use for recursive reads to disallow delimiters
    ; (except those in sub-expressions)
    (letrec ([misplaced-delimiter
              (case-lambda
               [(ch port) (unexpected-delimiter ch port #f #f #f #f)]
               [(ch port src line col pos)
                (raise-read-error
                 (format "misplaced `~a' in tuple" ch)
                 src line col pos 1)])])
      (make-readtable (current-readtable)
                      #\, 'terminating-macro misplaced-delimiter
                      #\> 'terminating-macro misplaced-delimiter)))
> (define (wrap l)
    `(make-tuple (list ,@l)))
> (define parse-open-tuple
    (case-lambda
     [(ch port)
      ; `read' mode
      (wrap (parse port
                   (lambda ()
                     (read/recursive port #f
                                     (make-delims-table)))
                   (object-name port)))]
     [(ch port src line col pos)
      ; `read-syntax' mode
      (datum->syntax
       #f
       (wrap (parse port
                    (lambda ()
                      (read-syntax/recursive src port #f
                                             (make-delims-table)))
                    src))
       (let-values ([(l c p) (port-next-location port)])
         (list src line col pos (and pos (- p pos)))))]))
> (define tuple-readtable
    (make-readtable #f #\< 'terminating-macro parse-open-tuple))
> (parameterize ([current-readtable tuple-readtable])
    (read (open-input-string "<1 , 2 , \"a\">")))
```

```
(make-tuple (list 1 2 "a"))
> (parameterize ([current-readtable tuple-readtable])
    (read (open-input-string
           "< #||# 1 #||# , #||# 2 #||# , #||# \"a\" #||# >")))
(make-tuple (list 1 2 "a"))
> (define tuple-readtable+
    (make-readtable tuple-readtable
                    #\* 'terminating-macro (lambda a (make-special-comment #f))
                    #\_ #\space #f))
> (parameterize ([current-readtable tuple-readtable+])
    (read (open-input-string "< * 1 __,__  2 __,__ * \"a\" * >")))
(make-tuple (list 1 2 "a"))
```

12.9.2 Reader-Extension Procedures

Calls to reader extension procedures can be triggered through `read`, `read/recursive`, `read-syntax`, or `read-honu-syntax`. In addition, a special-read procedure can be triggered by calls to `read-honu`, `read-honu/recursive`, `read-honu-syntax`, `read-honu-syntax/recursive`, `read-char-or-special`, or by the context of `read-bytes-avail!`, `read-bytes-avail!*`, `read-bytes-avail!`, and `peek-bytes-avail!*`.

Optional arities for reader-macro and special-result procedures allow them to distinguish reads via `read`, etc. from reads via `read-syntax`, etc. (where the source value is `#f` and no other location information is available).

When a reader-extension procedure is called in syntax-reading mode (via `read-syntax`, etc.), it should generally return a syntax object that has no lexical context (e.g., a syntax object created using `datum->syntax` with `#f` as the first argument and with the given location information as the third argument). Another possible result is a special-comment value (see §12.9.3 “Special Comments”). If the procedure’s result is not a syntax object and not a special-comment value, it is converted to one using `datum->syntax`.

When a reader-extension procedure is called in non-syntax-reading modes, it should generally not return a syntax object. If a syntax object is returned, it is converted to a plain value using `syntax-object->datum`.

In either context, when the result from a reader-extension procedure is a special-comment value (see §12.9.3 “Special Comments”), then `read`, `read-syntax`, etc. treat the value as a delimiting comment and otherwise ignore it.

Also, in either context, the result may be copied to prevent mutation to vectors or boxes before the read result is completed, and to support the construction of graphs with cycles. Mutable boxes, vectors, and prefab structures are copied, along with any pairs, boxes, vectors, pre prefab structures that lead to such mutable values, to placeholders produced by a recursive read (see `read/recursive`), or to references of a shared value. Graph structure (including cycles) is preserved in the copy.

12.9.3 Special Comments

```
(make-special-comment v) → special-comment?
  v : any/c
```

Creates a special-comment value that encapsulates `v`. The `read`, `read-syntax`, etc. procedures treat values constructed with `make-special-comment` as delimiting whitespace when returned by a reader-extension procedure (see §12.9.2 "Reader-Extension Procedures").

```
(special-comment? v) → boolean?
  v : any/c
```

Returns `#t` if `v` is the result of `make-special-comment`, `#f` otherwise.

```
(special-comment-value sc) → any
  sc : special-comment?
```

Returns the value encapsulated by the special-comment value `sc`. This value is never used directly by a reader, but it might be used by the context of a `read-char-or-special`, etc. call that detects a special comment.

12.10 Printer Extension

```
prop:custom-write : struct-type-property?
```

Associates a procedure to a structure type to used by the default printer to `display` or `write` (or `print`) instances of the structure type.

§4.3 "Structure Type Properties" provides more information on structure type properties.

The procedure for a `prop:custom-write` value takes three arguments: the structure to be printed, the target port, and a boolean that is `#t` for `write` mode and `#f` for `display` mode. The procedure should print the value to the given port using `write`, `display`, `fprintf`, `write-special`, etc.

The write handler, display handler, and print handler are specially configured for a port given to a custom-write procedure. Printing to the port through `display`, `write`, or `print` prints a value recursively with sharing annotations. To avoid a recursive print (i.e., to print without regard to sharing with a value currently being printed), print instead to a string or pipe and transfer the result to the target port using `write-string` and `write-special`. To recursively print but to a port other than the one given to the custom-write procedure, copy the given port's write handler, display handler, and print handler to the other port.

The port given to a custom-write handler is not necessarily the actual target port. In particular, to detect cycles and sharing, the printer invokes a custom-write procedure with a port that records recursive prints, and does not retain any other output.

Recursive print operations may trigger an escape from the call to the custom-write procedure (e.g., for pretty-printing where a tentative print attempt overflows the line, or for printing error output of a limited width).

The following example definition of a `tuple` type includes custom-write procedures that print the tuple's list content using angle brackets in `write` mode and no brackets in `display` mode. Elements of the tuple are printed recursively, so that graph and cycle structure can be represented.

Examples:

```
(define (tuple-print tuple port write?)
  (when write? (write-string "<" port))
  (let ([l (tuple-ref tuple 0)])
    (unless (zero? (vector-length l))
      ((if write? write display) (vector-ref l 0) port)
      (for-each (lambda (e)
                  (write-string ", " port)
                  ((if write? write display) e port))
                (cdr (vector->list l)))))
  (when write? (write-string ">" port)))

(define-values (s:tuple make-tuple tuple? tuple-ref tuple-set!)
  (make-struct-type 'tuple #f 1 0 #f
                    (list (cons prop:custom-write tuple-print))))

> (display (make-tuple #(1 2 "a")))
1, 2, a
> (let ([t (make-tuple (vector 1 2 "a"))])
    (vector-set! (tuple-ref t 0) 0 t)
    (write t))
#0=<#0#, 2, "a">
```

```
(custom-write? v) → boolean?
  v : any/c
```

Returns `#t` if *v* has the `prop:custom-write` property, `#f` otherwise.

```
(custom-write-accessor v)
```

```
→ (custom-write? output-port? boolean?. -> . any)
  v : custom-write?
```

Returns the custom-write procedure associated with `v`.

12.11 Serialization

```
(require scheme/serialize)
```

The bindings documented in this section are provided by the `scheme/serialize` library, not `scheme/base` or `scheme`.

```
(serializable? v) → boolean?
  v : any/c
```

Returns `#t` if `v` appears to be serializable, without checking the content of compound values, and `#f` otherwise. See `serialize` for an enumeration of serializable values.

```
(serialize v) → any
  v : serializable?
```

Returns a value that encapsulates the value `v`. This value includes only readable values, so it can be written to a stream with `write`, later read from a stream using `read`, and then converted to a value like the original using `deserialize`. Serialization followed by deserialization produces a value with the same graph structure and mutability as the original value, but the serialized value is a plain tree (i.e., no sharing).

The following kinds of values are serializable:

- structures created through `define-serializable-struct` or `define-serializable-struct/version`, or more generally structures with the `prop:serializable` property (see `prop:serializable` for more information);
- structures that instantiate prefab structure types;
- instances of classes defined with `define-serializable-class` or `define-serializable-class`;
- booleans, numbers, characters, symbols, strings, byte strings, paths (for a specific convention), `#<void>`, and the empty list;
- pairs, mutable pairs, vectors, boxes, and hash tables;
- `date` and `arity-at-least` structures; and

- module path index values.

Serialization succeeds for a compound value, such as a pair, only if all content of the value is serializable. If a value given to `serialize` is not completely serializable, the `exn:fail:contract` exception is raised.

See `deserialize` for information on the format of serialized data.

```
(deserialize v) → any
  v : any/c
```

Given a value *v* that was produced by `serialize`, produces a value like the one given to `serialize`, including the same graph structure and mutability.

A serialized representation *v* is a list of six or seven elements:

- An optional list `'(1)` that represents the version of the serialization format. If the first element of a representation is not a list, then the version is `0`. Version 1 adds support for mutable pairs.
- A non-negative exact integer *s-count* that represents the number of distinct structure types represented in the serialized data.
- A list *s-types* of length *s-count*, where each element represents a structure type. Each structure type is encoded as a pair. The `car` of the pair is `#f` for a structure whose deserialization information is defined at the top level, otherwise it is a quoted module path or a byte string (to be converted into a platform-specific path using `bytes->path`) for a module that exports the structure's deserialization information. The `cdr` of the pair is the name of a binding (at the top level or exported from a module) for deserialization information. These two are used with either `namespace-variable-binding` or `dynamic-require` to obtain deserialization information. See `make-deserialization-info` for more information on the binding's value. See also `deserialize-module-guard`.
- A non-negative exact integer, *g-count* that represents the number of graph points contained in the following list.
- A list *graph* of length *g-count*, where each element represents a serialized value to be referenced during the construction of other serialized values. Each list element is either a box or not:
  - A box represents a value that is part of a cycle, and for deserialization, it must be allocated with `#f` for each of its fields. The content of the box indicates the shape of the value:
    - a non-negative exact integer *i* for an instance of a structure type that is represented by the *i*th element of the *s-types* list;

* `'c` for a pair, which fails on deserialization (since pairs are immutable; this case does not appear in output generated by `serialize`);
* `'m` for a mutable pair;
* `'b` for a box;
* a pair whose `car` is `'v` and whose `cdr` is a non-negative exact integer *`s`* for a vector of length *`s`*; or
* a list whose first element is `'h` and whose remaining elements are symbols that determine the hash-table type:
  - `'equal` — `(make-hash)`
  - `'equal 'weak` — `(make-weak-hash)`
  - `'weak` — `(make-weak-hasheq)`
  - no symbols — `(make-hasheq)`
* `'date` for a `date` structure, which fails on deserialization (since dates are immutable; this case does not appear in output generated by `serialize`);
* `'arity-at-least` for an `arity-at-least` structure, which fails on deserialization (since dates are immutable; this case does not appear in output generated by `serialize`); or
* `'mpi` for a module path index, which fails on deserialization (since dates are immutable; this case does not appear in output generated by `serialize`).

The `#f`-filled value will be updated with content specified by the fifth element of the serialization list *`v`*.

– A non-box represents a *serial* value to be constructed immediately, and it is one of the following:

* a boolean, number, character, symbol, or empty list, representing itself.
* a string, representing an immutable string.
* a byte string, representing an immutable byte string.
* a pair whose `car` is `'?` and whose `cdr` is a non-negative exact integer *`i`*; it represents the value constructed for the *`i`*th element of *`graph`*, where *`i`* is less than the position of this element within *`graph`*.
* a pair whose `car` is a number *`i`*; it represents an instance of a structure type that is described by the *`i`*th element of the *`s-types`* list. The `cdr` of the pair is a list of serials representing arguments to be provided to the structure type's deserializer.
* a pair whose `car` is `'f`; it represents an instance of a prefab structure type. The `cadr` of the pair is prefab structure type key, and the `cddr` is a list of serials representing the field values.
* a pair whose `car` is `'void`, representing `#<void>`.
* a pair whose `car` is `'u` and whose `cdr` is either a byte string or character string; it represents a mutable byte or character string.
* a pair whose `car` is `'p` and whose `cdr` is a byte string; it represents a path using the serializer's path convention (deprecated in favor of `'p+`).

- a pair whose `car` is `'p+`, whose `cadr` is a byte string, and whose `cddr` is one of the possible symbol results of `system-path-convetion-type`; it represents a path using the specified convention.
- a pair whose `car` is `'c` and whose `cdr` is a pair of serials; it represents an immutable pair.
- a pair whose `car` is `'c!` and whose `cdr` is a pair of serials; it represents a pair (but formerly presented a mutable pair), and does not appear in output generated by `serialize`.
- a pair whose `car` is `'m` and whose `cdr` is a pair of serials; it represents a mutable pair.
- a pair whose `car` is `'v` and whose `cdr` is a list of serials; it represents an immutable vector.
- a pair whose `car` is `'v!` and whose `cdr` is a list of serials; it represents a mutable vector.
- a pair whose `car` is `'b` and whose `cdr` is a serial; it represents an immutable box.
- a pair whose `car` is `'b!` and whose `cdr` is a serial; it represents a mutable box.
- a pair whose `car` is `'h`, whose `cadr` is either `'!` or `'-` (mutable or immutable, respectively), whose `caddr` is a list of symbols (containing `'equal`, `'weak`, both, or neither) that determines the hash table type, and whose `cdddr` is a list of pairs, where the `car` of each pair is a serial for a hash-table key and the `cdr` is a serial for the corresponding value.
- a pair whose `car` is `'date` and whose `cdr` is a list of serials; it represents a `date` structure.
- a pair whose `car` is `'arity-at-least` and whose `cdr` is a serial; it represents an `arity-at-least` structure.
- a pair whose `car` is `'mpi` and whose `cdr` is a pair; it represents an module path index that joins the paired values.

• A list of pairs, where the `car` of each pair is a non-negative exact integer *`i`* and the `cdr` is a serial (as defined in the previous bullet). Each element represents an update to an *`i`*th element of *`graph`* that was specified as a box, and the serial describes how to construct a new value with the same shape as specified by the box. The content of this new value must be transferred into the value created for the box in *`graph`*.

• A final serial (as defined in the two bullets back) representing the result of `deserialize`.

The result of `deserialize` shares no mutable values with the argument to `deserialize`.

If a value provided to `serialize` is a simple tree (i.e., no sharing), then the fourth and fifth elements in the serialized representation will be empty.

```
(deserialize-module-guard)
 → (module-path? symbol? . -> . void?)
(deserialize-module-guard guard) → void?
  guard : (module-path? symbol? . -> . void?)
```

A parameter whose value is called by deserialize before dynamically loading a module via dynamic-require. The two arguments provided to the procedure are the same as the arguments to be passed to dynamic-require. The procedure can raise an exception to disallow the dynamic-require.

```
(define-serializable-struct id-maybe-super (field ...)
                            struct-option ...)
```

Like define-struct, but instances of the structure type are serializable with serialize. This form is allowed only at the top level or in a module's top level (so that deserialization information can be found later).

Serialization only supports cycles involving the created structure type when all fields are mutable (or when the cycle can be broken through some other mutable value).

In addition to the bindings generated by define-struct, define-serializable-struct binds deserialize-info:*id*-v0 to deserialization information. Furthermore, in a module context, it automatically provides this binding.

The define-serializable-struct form enables the construction of structure instances from places where make*id* is not accessible, since deserialization must construct instances. Furthermore, define-serializable-struct provides limited access to field mutation, but only for instances generated through the deserialization information bound to deserialize-info:*id*-v0. See make-deserialize-info for more information.

The -v0 suffix on the deserialization enables future versioning on the structure type through define-serializable-struct/version.

When a supertype is supplied in *id-maybe-super* is supplied, compile-time information bound to the supertype identifier must include all of the supertype's field accessors. If any field mutator is missing, the structure type will be treated as immutable for the purposes of marshaling (so cycles involving only instances of the structure type cannot be handled by the deserializer).

Examples:

```
> (define-serializable-struct point (x y))
> (point-x (deserialize (serialize (make-point 1 2))))
1
```

```
(define-serializable-struct/versions id-maybe-super vers (field ...)
```

```
                                        (other-version-clause ...)
                                        struct-option ...)

other-version-clause = (other-vers make-proc-expr
                                   cycle-make-proc-expr)
```

Like `define-serializable-struct`, but the generated deserializer binding is `deserialize-info:id-vvers`. In addition, `deserialize-info:id-vother-vers` is bound for each *`other-vers`*. The *`vers`* and each *`other-vers`* must be a literal, exact, nonnegative integer.

Each *`make-proc-expr`* should produce a procedure, and the procedure should accept as many argument as fields in the corresponding version of the structure type, and it produce an instance of `id`. Each `graph-make-proc-expr` should produce a procedure of no arguments; this procedure should return two values: an instance `x` of `id` (typically with `#f` for all fields) and a procedure that accepts another instance of `id` and copies its field values into `x`.

Examples:

```
> (define-serializable-struct point (x y) #:mutable #:transparent)
> (define ps (serialize (make-point 1 2)))
> (deserialize ps)
#(struct:point 1 2)
> (define x (make-point 1 10))
> (set-point-x! x x)
> (define xs (serialize x))
> (deserialize xs)
#0=#(struct:point #0# 10)
> (define-serializable-struct/versions point 1 (x y z)
     ([0
       ; Constructor for simple v0 instances:
       (lambda (x y) (make-point x y 0))
       ; Constructor for v0 instance in a cycle:
       (lambda ()
         (let ([p0 (make-point #f #f 0)])
           (values
             p0
             (lambda (p)
               (set-point-x! p0 (point-x p))
               (set-point-y! p0 (point-y p))))))])
     #:mutable #:transparent)
> (deserialize (serialize (make-point 4 5 6)))
#(struct:point 4 5 6)
> (deserialize ps)
#(struct:point 1 2 0)
> (deserialize xs)
#0=#(struct:point #0# 10 0)
```

```
(make-deserialize-info make cycle-make) → any
  make : procedure?
  cycle-make : (() . ->* . (any/c procedure?))
```

Produces a deserialization information record to be used by `deserialize`. This information is normally tied to a particular structure because the structure has a `prop:serializable` property value that points to a top-level variable or module-exported variable that is bound to deserialization information.

The *`make`* procedure should accept as many argument as the structure's serializer put into a vector; normally, this is the number of fields in the structure. It should return an instance of the structure.

The *`cycle-make`* procedure should accept no arguments, and it should return two values: a structure instance `x` (with dummy field values) and an update procedure. The update procedure takes another structure instance generated by the *`make`*, and it transfers the field values of this instance into `x`.

```
prop:serializable : property?
```

This property identifies structures and structure types that are serializable. The property value should be constructed with `make-serialize-info`.

```
(make-serialize-info to-vector
                     deserialize-id
                     can-cycle?
                     dir)           → any
  to-vector : (any/c . -> . vector?)
  deserialize-id : (or identifier?
                       symbol?
                       (cons/c symbol?
                               module-path-index?))
  can-cycle? : any/c
  dir : path-string?
```

Produces a value to be associated with a structure type through the `prop:serializable` property. This value is used by `serialize`.

The *`to-vector`* procedure should accept a structure instance and produce a vector for the instance's content.

The *`deserialize-id`* value indicates a binding for deserialize information, to either a module export or a top-level definition. It must be one of the following:

- If *deserialize-id* is an identifier, and if (identifier-binding *deserialize-id*) produces a list, then the third element is used for the exporting module, otherwise the top-level is assumed. In either case, syntax-e is used to obtain the name of an exported identifier or top-level definition.
- If *deserialize-id* is a symbol, it indicates a top-level variable that is named by the symbol.
- If *deserialize-id* is a pair, the car must be a symbol to name an exported identifier, and the cdr must be a module path index to specify the exporting module.

See make-deserialize-info and deserialize for more information.

The *can-cycle?* argument should be false if instances should not be serialized in such a way that deserialization requires creating a structure instance with dummy field values and then updating the instance later.

The *dir* argument should be a directory path that is used to resolve a module reference for the binding of *deserialize-id*. This directory path is used as a last resort when *deserialize-id* indicates a module that was loaded through a relative path with respect to the top level. Usually, it should be (or (current-load-relative-directory) (current-directory)).

13 Reflection and Security

13.1 Namespaces

See §1.2.5 “Namespaces” for basic information on the namespace model.

A new namespace is created with procedures like `make-empty-namespace`, and `make-base-namespace`, which return a first-class namespace value. A namespace is used by setting the `current-namespace` parameter value, or by providing the namespace to procedures such as `eval` and `eval-syntax`.

```
(namespace? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a namespace value, `#f` otherwise.

```
(make-empty-namespace) → namespace?
```

Creates a new namespace that is empty, and whose module registry contains no mappings. Attach modules from an existing namespace to the new one with `namespace-attach-module`.

```
(make-base-empty-namespace) → namespace?
```

Creates a new empty namespace, but with `scheme/base` attached.

```
(make-base-namespace) → namespace?
```

Creates a new namespace with `scheme/base` attached and `require`d into the top-level environment.

```
(define-namespace-anchor id)
```

Binds *id* to a namespace anchor that can be used with `namespace-anchor->empty-namespace` and `namespace-anchor->namespace`.

This form can be used only in a top-level context or in a module-context.

```
(namespace-anchor? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a namespace-anchor value, `#f` otherwise.

```
(namespace-anchor->empty-namespace a) → namespace?
  a : namespace-anchor?
```

Returns an empty namespace that shares a module registry with the source of the anchor.

If the anchor is from a `define-namespace-anchor` form in a module context, then the source is the namespace in which the containing module is instantiated. If the anchor is from a `define-namespace-anchor` form in a top-level content, then the source is the namespace in which the anchor definition was evaluated. The resulting namespace corresponds to phase 0, independent of the phase of `a`'s definition.

```
(namespace-anchor->namespace a) → namespace?
  a : namespace-anchor?
```

Returns a namespace corresponding to the source of the anchor.

If the anchor is from a `define-namespace-anchor` form in a module context, then the result is a namespace obtained via `module->namespace` using the resolved name of the enclosing module and the module registry of the module instance at phase 0.

If the anchor is from a `define-namespace-anchor` form in a top-level content, then the result is the namespace in which the anchor definition was evaluated.

```
(current-namespace) → namespace?
(current-namespace n) → void?
  n : namespace?
```

A parameter that determines the current namespace.

```
(namespace-symbol->identifier sym) → identifier?
  sym : symbol?
```

Similar to `datum->syntax` restricted to symbols. The lexical context of the resulting identifier corresponds to the top-level environment of the current namespace; the identifier has no source location or properties.

```
(namespace-variable-value  sym
                          [use-mapping?
                           failure-thunk
                           namespace])   → any
  sym : symbol?
```

```
use-mapping? : any/c = #t
failure-thunk : (or/c (-> any) false/c) = #f
namespace : namespace? = (current-namespace)
```

Returns a value for *sym* in *namespace*. The returned value depends on *use-mapping?*:

- If *use-mapping?* is true (the default), and if *sym* maps to a top-level variable or an imported variable (see §1.2.5 "Namespaces"), then the result is the same as evaluating *sym* as an expression. If *sym* maps to syntax or imported syntax, then *failure-thunk* is called or the `exn:fail:syntax` exception is raised. If *sym* is mapped to an undefined variable or an uninitialized module variable, then *failure-thunk* is called of the `exn:fail:contract:variable` exception is raised.
- If *use-mapping?* is `#f`, the namespace's syntax and import mappings are ignored. Instead, the value of the top-level variable named *sym* in namespace is returned. If the variable is undefined, then *failure-thunk* is called or the `exn:fail:contract:variable` exception is raised.

If *failure-thunk* is not `#f`, `namespace-variable-value` calls *failure-thunk* to produce the return value in place of raising an `exn:fail:contract:variable` or `exn:fail:syntax` exception.

```
(namespace-set-variable-value! sym
                               v
                              [map?
                               namespace]) → void?
  sym : symbol?
  v : any/c
  map? : any/c = #f
  namespace : namespace? = (current-namespace)
```

Sets the value of *sym* in the top-level environment of *namespace* for phase level 0, defining *sym* if it is not already defined.

If *map?* is supplied as true, then the namespace's identifier mapping is also adjusted (see §1.2.5 "Namespaces") so that *sym* maps to the variable.

```
(namespace-undefine-variable! sym
                             [namespace]) → void?
  sym : symbol?
  namespace : namespace? = (current-namespace)
```

Removes the *sym* variable, if any, in the top-level environment of *namespace* at phase level

0. The namespace's identifier mapping (see §1.2.5 "Namespaces") is unaffected.

```
(namespace-mapped-symbols [namespace]) → (listof symbol?)
  namespace : namespace? = (current-namespace)
```

Returns a list of all symbols that are mapped to variables, syntax, and imports in *namespace* for phase level 0.

```
(namespace-require quoted-raw-require-spec) → void?
  quoted-raw-require-spec : any/c
```

Performs the import corresponding to *quoted-raw-require-spec* in the top-level environment of the current namespace, like a top-level `#%require`. The *quoted-raw-require-spec* argument must be a datum that corresponds to a quoted *raw-require-spec* for `#%require`, which includes module paths.

Module paths in *quoted-raw-require-spec* are resolved with respect to `current-load-relative-directory` or `current-directory` (if the former is `#f`), even if the current namespace corresponds to a module body.

```
(namespace-require/copy quoted-raw-require-spec) → void?
  quoted-raw-require-spec : any/c
```

Like `namespace-require` for syntax exported from the module, but exported variables are treated differently: the export's current value is copied to a top-level variable in the current namespace.

```
(namespace-require/expansion-time quoted-raw-require-spec)
 → void?
  quoted-raw-require-spec : any/c
```

Like `namespace-require`, but only the transformer part of the module is executed; that is, the module is merely visited, and not instantiated (see §1.2.3.8 "Module Phases"). If the required module has not been instantiated before, the module's variables remain undefined.

```
(namespace-attach-module  src-namespace
                          modname
                         [dest-namespace]) → any
  src-namespace : namespace?
  modname : module-path?
  dest-namespace : namespace? = (current-namespace)
```

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

```
(namespace-unprotect-module  inspector
                             modname
                            [namespace]) → void?
  inspector : inspector?
  modname : module-path?
  namespace : namespace? = (current-namespace)
```

Changes the inspector for the instance of the module referenced by *modname* in *namespace*'s module registry so that it is controlled by the current code inspector. The given *inspector* must currently control the invocation of the module in *namespace*'s module registry, otherwise the `exn:fail:contract` exception is raised. See also §13.9 "Code Inspectors".

```
(namespace-module-registry namespace) → any
  namespace : namespace?
```

Returns the module registry of the given namespace. This value is useful only for identification via `eq?`.

```
(module->namespace modname) → namespace?
  modname : module-path?
```

Returns a namespace that corresponds to the body of an instantiated module in the current namespace's module registry. The returned namespace has the same module registry as the current namespace. Modifying a binding in the namespace changes the binding seen in modules that require the namespace's module.

Module paths in a top-level `require` expression are resolved with respect to the namespace's module. New `provide` declarations are not allowed.

If the current code inspector does not control the invocation of the module in the current namespace's module registry, the `exn:fail:contract` exception is raised; see also §13.9 "Code Inspectors".

Bindings in the namespace cannot be modified if the `compile-enforce-module-constants` parameter was true when the module was declared, unless the module declaration itself included assignments to the binding via `set!`.

```
(namespace-syntax-introduce stx) → syntax-object?
  stx : syntax-object?
```

Returns a syntax object like *stx*, except that the current namespace's bindings are included in the syntax object's lexical information (see §1.2.2 "Syntax Objects"). The additional context is overridden by any existing top-level bindings in the syntax object's lexical information, or by any existing or future module bindings in the lexical information.

```
(module-provide-protected? module-path-index
                           sym)              → boolean?
  module-path-index : (or/c symbol? module-path-index?)
  sym : symbol?
```

Returns `#f` if the module declaration for *module-path-index* defines *sym* and exports it unprotected, `#t` otherwise (which may mean that the symbol corresponds to an unexported definition, a protected export, or an identifier that is not defined at all within the module).

The *module-path-index* argument can be a symbol; see §13.4.2 "Compiled Modules and References" for more information on module path indices.

Typically, the arguments to `module-provide-protected?` correspond to the first two elements of a list produced by `identifier-binding`.

```
(variable-reference->empty-namespace varref) → namespace?
  varref : variable-reference?
```

Returns an empty namespace that shares module declarations and instances with the namespace in which *varref* is instantiated. The namespace corresponds to phase 0, independent of the phase of *varref*'s binding.

```
(variable-reference->top-level-namespace varref) → namespace?
  varref : variable-reference?
```

If *varref* refers to a top-level binding, the result is *varref*'s namespace if it corresponds to a phase 0 binding, otherwise it is the phase 0 namespace associated with *varref*'s namespace.

If *varref* refers to a module binding, then the `exn:fail:contract` exception is raised.

```
(variable-reference->resolved-module-path varref)
 → resolved-module-path?
  varref : variable-reference?
```

If *varref* refers to a module binding, the result is a resolved module path naming the module.

If *varref* refers to a top-level binding, then the `exn:fail:contract` exception is raised.

13.2 Evaluation and Compilation

```
(current-eval) → (any/c . -> . any)
(current-eval proc) → void?
  proc : (any/c . -> . any)
```

A parameter that determines the current *evaluation handler*. The evaluation handler is a procedure that takes a top-level form and evaluates it, returning the resulting values. The evaluation handler is called by `eval`, `eval-syntax`, the default load handler, and `read-eval-print-loop` to evaluate a top-level form. The handler should evaluate its argument in tail position.

The *top-level-form* provided to the handler can be a syntax object, a compiled form, a compiled form wrapped as a syntax object, or an arbitrary datum.

The default handler converts an arbitrary datum to a syntax object using `datum->syntax`, and then enriches its lexical information in the same way as `eval`. (If *top-level-form* is a syntax object, then its lexical information is not enriched.) The default evaluation handler partially expands the form to splice the body of top-level `begin` forms into the top level (see `expand-to-top-form`), and then individually compiles and evaluates each spliced form before continuing to expand, compile, and evaluate later forms.

```
(eval top-level-form [namespace]) → any
  top-level-form : any/c
  namespace : namespace? = (current-namespace)
```

Calls the current evaluation handler to evaluate *top-level-form*. The evaluation handler is called in tail position with respect to the `eval` call, and `parameterize`d to set `current-namespace` to *namespace*.

If *top-level-form* is a syntax object whose datum is not a compiled form, then its lexical information is enriched before it is sent to the evaluation handler:

- If *top-level-form* is a pair whose `car` is a symbol or identifier, and if applying `namespace-syntax-introduce` to the (`datum->syntax`-converted) identifier produces an identifier bound to `module`, then only that identifier is enriched.
- For any other *top-level-form*, `namespace-syntax-introduce` is applied to the entire syntax object.

For interactive evaluation in the style of `read-eval-print-loop` and `load`, wrap each expression with `#%top-interaction`, which is normally bound to `#%top-interaction`, before passing it to `eval`.

```
(eval-syntax stx [namespace]) → any
  stx : syntax?
  namespace : namespace? = (current-namespace)
```

Like `eval`, except that *stx* must be a syntax object, and its lexical context is not enriched before it is passed to the evaluation handler.

```
(current-load) → (path? (or/c symbol? false/c) . -> . any)
(current-load proc) → void?
  proc : (path? (or/c symbol? false/c) . -> . any)
```

A parameter that determines the current *load handler* to load top-level forms from a file. The load handler is called by `load`, `load-relative`, `load/cd`, and the default compiled-load handler.

A load handler takes two arguments: a path (see §14.1 “Paths”) and an expected module name. The expected module name is a symbol when the call is to load a module declaration in response to a `require` (in which case the file should contain a module declaration), or `#f` for any other load.

The default load handler reads forms from the file in `read-syntax` mode with line-counting enabled for the file port, unless the path has a `".zo"` suffix. It also `parameterize`s each read to set both `read-accept-compiled` and `read-accept-reader` to `#t`. In addition, if `load-on-demand-enabled` is `#t`, then `read-on-demand-source` is effectively set to the cleansed, absolute form of `path` during the `read-syntax` call. After reading a single form, the form is passed to the current evaluation handler, wrapping the evaluation in a continuation prompt (see `call-with-continuation-prompt`) for the default continuation prompt tag with handler that propagates the abort to the continuation of the `load` call.

If the second argument to the load handler is a symbol, then:

- The `read-syntax` from the file is additionally `parameterize`d as follows (to provide consistent reading of module source):

```
(current-readtable #f)
(read-case-sensitive #t)
(read-square-bracket-as-paren #t)
(read-curly-brace-as-paren #t)
(read-accept-box #t)
(read-accept-compiled #t)
(read-accept-bar-quote #t)
(read-accept-graph #t)
(read-decimal-as-inexact #t)
(read-accept-dot #t)
(read-accept-infix-dot #t)
(read-accept-quasiquote #t)
(read-accept-reader #t)
```

- If the read result is not a `module` form with the expected name, or if a second `read-syntax` does not produce an end-of-file, then the `exn:fail` exception is raised without evaluating the form that was read from the file.
- The lexical information of the initial `module` identifier is enriched with a binding for `module`, so that the form corresponds to a module declaration independent of the current namespace's bindings.

If the second argument to the load handler is `#f`, then each expression read from the file is wrapped with `#%top-interaction`, which is normally bound to `#%top-interaction`, before passing it to the evaluation handler.

The return value from the default load handler is the value of the last form from the loaded file, or `#<void>` if the file contains no forms. If the given path is a relative path, then it is resolved using the value of `current-directory`.

```
(load file) → any
  file : path-string?
```

Calls the current load handler in tail position. The call is `parameterized` to set `current-load-relative-directory` to the directory of *`file`*, which is resolved relative to the value of `current-directory`.

```
(load-relative file) → any
  file : path-string?
```

Like `load/use-compiled`, but when *`file`* is a relative path, it is resolved using the value of `current-load-relative-directory` instead of the value of `current-directory` if the former is not `#f`, otherwise `current-directory` is used.

```
(load/cd file) → any
  file : path-string?
```

Like `load`, but `load/cd` sets both `current-directory` and `current-load-relative-directory` before calling the load handler.

```
(current-load-extension)
 → (path? (or/c symbol? false/c) . -> . any)
(current-load-extension proc) → void?
  proc : (path? (or/c symbol? false/c) . -> . any)
```

A parameter that determines a *extension-load handler*, which is called by `load-extension` and the default compiled-load handler.

An extension-load handler takes the same arguments as a load handler, but the file should be a platform-specific *dynamic extension*, typically with the file suffix `".so"` (Unix), `".dll"` (Windows), or `".dylib"` (Mac OS X). The file is loaded using internal, OS-specific primitives. See §"**Inside**: PLT Scheme C API" for more information on dynamic extensions.

```
(load-extension file) → any
  file : path-string?
```

Sets `current-load-relative-directory` like `load`, and calls the extension-load handler in tail position.

```
(load-relative-extension file) → any
  file : path-string?
```

Like `load-exension`, but resolves *file* using `current-load-relative-directory` like `load-relative`.

```
(current-load/use-compiled)
 → (path? (or/c symbol? false/c) . -> . any)
(current-load/use-compiled proc) → void?
  proc : (path? (or/c symbol? false/c) . -> . any)
```

A parameter that determines the current *compiled-load handler* to load from a file that may have a compiled form. The compiled-load handler is called by `load/use-compiled`.

The protocol for a compiled-load handler is the same as for the load handler (see `current-load`), except that a compiled-load handler is expected to set `current-load-relative-directory` itself. The default compiled-load handler, however, checks for `".zo"` files (usually produced with `compile-file`) and `".so"` (Unix), `".dll"` (Windows), or `".dylib"`

(Mac OS X) files.

The check for a compiled file occurs whenever the given path *file* ends with any extension (e.g., ".ss" or ".scm"), and the check consults the subdirectories indicated by the `use-compiled-file-paths` parameter relative to *file*. The subdirectories are checked in order. A ".zo" version of the file is loaded if it exists directly in one of the indicated subdirectories, or a ".so"/".dll"/".dylib" version of the file is loaded if it exists within a "native" subdirectory of a `use-compiled-file-paths` directory, in an even deeper subdirectory as named by `system-library-subpath`. A compiled file is loaded only if its modification date is not older than the date for *file*. If both ".zo" and ".so"/".dll"/".dylib" files are available, the ".so"/".dll"/".dylib" file is used.

While a ".zo", ".so", ".dll", or ".dylib" file is loaded, the current `load-relative` directory is set to the directory of the original *file*.

If the original *file* is loaded or a ".zo" variant is loaded, the load handler is called to load the file. If any other kind of file is loaded, the extension-load handler is called.

```
(load/use-compiled file) → any
  file : path-string?
```

Calls the current compiled-load handler in tail position.

```
(current-load-relative-directory) → (and/c path-string?
                                            complete-path?)
(current-load-relative-directory path) → void?
  path : (and/c path-string?
                complete-path?)
```

A parameter that is set by `load`, `load-relative`, `load-extension`, `load-relative-extension`, and the default compiled-load handler, and used by `load-relative`, `load-relative-extension`, and the default compiled-load handler.

When a new path or string is provided as the parameter's value, it is immediately expanded (see §14.1 "Paths") and converted to a path. (The directory need not exist.)

```
(use-compiled-file-paths) → (listof path?)
(use-compiled-file-paths paths) → void?
  paths : (listof path-string?)
```

A list of relative paths, which defaults to `(list (string->path "compiled"))`. It is used by the compiled-load handler (see `current-load/use-compiled`).

```
(read-eval-print-loop) → any
```

Starts a new *REPL* using the current input, output, and error ports. The REPL wraps each expression to evaluate with `#%top-interaction`, which is normally bound to `#%top-interaction`, and it wraps each evaluation with a continuation prompt using the default continuation prompt tag and prompt handler (see `call-with-continuation-prompt`). The REPL also wraps the read and print operations with a prompt for the default tag whose handler ignores abort arguments and continues the loop. The `read-eval-print-loop` procedure does not return until `eof` is read, at which point it returns `#<void>`.

The `read-eval-print-loop` procedure can be configured through the `current-prompt-read`, `current-eval`, and `current-print` parameters.

```
(current-prompt-read) → (-> any)
(current-prompt-read proc) → void?
  proc : (-> any)
```

A parameter that determines a procedure that takes no arguments, displays a prompt string, and returns a top-level form to evaluate. This procedure is called by the read phase of `read-eval-print-loop`. The default prompt read handler prints `> ` and returns the result of

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

```
(current-print) → (any/c -> any)
(current-print proc) → void?
  proc : (any/c -> any)
```

A parameter that determines the *print handler* that is called by `read-eval-print-loop` 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 (as determined by the `current-output-port` parameter) and then outputs a newline, except that it prints nothing when the value is `#<void>`.

```
(current-compile)
 → (any/c boolean? . -> . compiled-expression?)
(current-compile proc) → void?
  proc : (any/c boolean? . -> . compiled-expression?)
```

A parameter that determines the current *compilation handler*. The compilation handler is a procedure that takes a top-level form and returns a compiled form; see see §1.2.4 “Compilation” for more information on compilation.

The compilation handler is called by `compile`, and indirectly by the default evaluation handler and the default load handler.

The handler's second argument is `#t` if the compiled form will be used only for immediate evaluation, or `#f` if the compiled form may be saved for later use; the default compilation handler is optimized for the special case of immediate evaluation.

When a compiled form is written to an output port, the written form starts with `#~`. These forms are essentially assembly code for PLT Scheme, and reading such an form produces a compiled form (as long as the `read-accept-compiled` parameter is set to `#t`).

When a compiled form contains syntax object constants, the `#~`-marshaled form drops source-location information and properties (§11.6 "Syntax Object Properties") for the syntax objects.

Compiled code parsed from `#~` may contain references to unexported or protected bindings from a module. At read time, such references are associated with the current code inspector (see `current-code-inspector`), and the code will only execute if that inspector controls the relevant module invocation (see §13.9 "Code Inspectors").

A compiled-form object may contain uninterned symbols (see §3.6 "Symbols") that were created by `gensym` or `string->uninterned-symbol`. When the compiled object is read via `#~`, each uninterned symbol in the original form is mapped to a new uninterned symbol, where multiple instances of a single symbol are consistently mapped to the same new symbol. The original and new symbols have the same printed representation.

Due to the above restrictions, do not use `gensym` or `string->uninterned-symbol` to construct an identifier for a top-level or module binding. Instead, generate distinct identifiers either with `generate-temporaries` or by applying the result of `make-syntax-introducer` to an existing identifier.

```
(compile top-level-form) → compiled-expression?
  top-level-form : any/c
```

Like `eval`, but calls the current compilation handler in tail position with *`top-level-form`*.

```
(compile-syntax stx) → compiled-expression?
  stx : syntax?
```

Like `eval-syntax`, but calls the current compilation handler in tail position with *`stx`*.

```
(compiled-expression? v) → boolean?
  v : any/c
```

Returns `#t` if *`v`* is a compiled form, `#f` otherwise.

```
(compile-enforce-module-constants) → boolean?
```

`(compile-enforce-module-constants on?) → void?`
`  on? : any/c`

A parameter that determines how a module declaration is compiled.

When constants are enforced, and when the macro-expanded body of a module contains no `set!` assignment to a particular variable defined within the module, then the variable is marked as constant when the definition is evaluated. Afterward, the variable's value cannot be assigned or undefined through `module->namespace`, and it cannot be defined by redeclaring the module.

Enforcing constants allows the compiler to inline some variable values, and it allows the native-code just-in-time compiler to generate code that skips certain run-time checks.

```
(compile-allow-set!-undefined) → boolean?
(compile-allow-set!-undefined allow?) → void?
  allow? : any/c
```

A parameter that determines how a `set!` expression is compiled when it mutates a global variable. If the value of this parameter is a true value, `set!` expressions for global variables are compiled so that the global variable is set even if it was not previously defined. Otherwise, `set!` expressions for global variables are compiled to raise the `exn:fail:contract:variable` exception if the global variable is not defined at the time the `set!` is performed. Note that this parameter is used when an expression is *compiled*, not when it is *evaluated*.

```
(eval-jit-enabled) → boolean?
(eval-jit-enabled on?) → void?
  on? : any/c
```

A parameter that determines whether the native-code just-in-time compiler (JIT) is enabled for code (compiled or not) that is passed to the default evaluation handler. The default is `#t`, unless the JIT is disabled through the `-j`/`--no-jit` command-line flag to stand-alone MzScheme (or MrEd), or through the `PLTNOMZJIT` environment variable (set to any value).

```
(load-on-demand-enabled) → boolean?
(load-on-demand-enabled on?) → void?
  on? : any/c
```

A parameter that determines whether the default load handler sets `read-on-demand-source`. See `current-load` for more information. The default is `#t`, unless it is disabled through the `-d`/`--no-delay` command-line flag.

13.3 The scheme/load Language

```
#lang scheme/load
```

The scheme/load language supports traditional Scheme evaluation, where each top-level form in the module body is separately passed to eval in the same way as for load.

The namespace for evaluation shares the module registry with the scheme/load module instance, but it has a separate top-level environment, and it is initialized with the bindings of scheme. A single namespace is created for each instance of the scheme/load module (i.e., multiple modules using the scheme/load language share a namespace). The scheme/load library exports only #%module-begin and #%top-interaction forms that effectively swap in the evaluation namespace and call eval.

For example, the body of a module using scheme/load can include module forms, so that running the following module prints 5:

```
#lang scheme/load

(module m scheme/base
  (provide x)
  (define x 5))

(module n scheme/base
  (require 'm)
  (display x))

(require 'n)
```

Definitions in a module using scheme/load are evaluated in the current namespace, which means that load and eval can see the definitions. For example, running the following module prints 6:

```
#lang scheme/load

(define x 6)
(display (eval 'x))
```

Since all forms within a scheme/load module are evaluated in the top level, bindings cannot be exported from the module using provide. Similarly, since evaluation of the module-body forms is inherently dynamic, compilation of the module provides essentially no benefit. For these reasons, use scheme/load for interactive exploration of top-level forms only, and not for constructing larger programs.

13.4 Module Names and Loading

13.4.1 Resolving Module Names

The name of a declared module is represented by a *resolved module path*, which encapsulates either a symbol or a complete filesystem path (see §14.1 "Paths"). A symbol normally refers to a predefined module or module declared through reflective evaluation (e.g., `eval`). A filesystem path normally refers to a module declaration that was loaded on demand via `require` or other forms.

A *module path* is a datum that matches the grammar for *`module-path`* for `require`. A module path is relative to another module.

```
(resolved-module-path? v) → boolean?
  v : any/c
```

Returns `#f` if *`v`* is a resolved module path, `#f` otherwise.

```
(make-resolved-module-path path) → resolved-module-path?
  path : (or/c symbol? (and/c path? complete-path?))
```

Returns a resolved module path that encapsulates *`path`*. If *`path`* is not a symbol, it normally should be cleansed (see `cleanse-path`) and simplified (see `simplify-path`).

A resolved module path is interned. That is, if two resolved module path values encapsulate paths that are `equal?`, then the resolved module path values are `eq?`.

```
(resolved-module-path-name module-path) → path?
  module-path : resolved-module-path?
```

Returns the path encapsulated by a resolved module path.

```
(module-path? v) → boolean?
  v : any/c
```

Returns `#t` if *`v`* corresponds to a datum that matches the grammar for *`module-path`* for `require`, `#f` otherwise.

```
(current-module-name-resolver)
```

```
 → (case->
    (resolved-module-path?
     . -> .
     any)
    ((or/c module-path? path?)
     (or/c false/c resolved-module-path?)
     (or/c false/c syntax?)
     boolean?
     . -> .
     resolved-module-path?))
(current-module-name-resolver proc) → void?
  proc : (case->
          (resolved-module-path?
           . -> .
           any)
          ((or/c module-path? path?)
           (or/c false/c resolved-module-path?)
           (or/c false/c syntax?)
           boolean?
           . -> .
           resolved-module-path?))
```

A parameter that determines the current *module name resolver*, which manages the conversion from other kinds of module references to a symbol or resolved module path. For example, when the expander encounters `(require module-path)` where *module-path* is not an identifier, then the expander passes `'module-path` to the module name resolver to obtain a symbol or resolved module path. When such a `require` appears within a module, the *module path resolver* is also given the name of the enclosing module, so that a relative reference can be converted to an absolute symbol or resolved module path.

A module name resolver takes one and four arguments: %

- When given one argument, it is a name for a module declaration that is already loaded. Such a call to the module name resolver is a notification that the corresponding module does not need to be loaded (for the current namespace, or any other namespace that shares the same module registry). The module name resolver's result is ignored.
- When given four arguments, the first is a module path, either equivalent to a quoted `module-path` for `require` or a file system path. The second is name for the source module, if any, to which the path is relative; f the second argument is `#f`, the module path is relative to `(or (current-load-relative-directory) (current-directory))`. The third argument is a syntax object that can be used for error reporting, if it is not `#f`. If the last argument is `#t`, then the module declaration should be loaded (if it is not already), otherwise the module path should be simply resolved to a name. The result is the resolved name.

For the second case, the standard module name resolver keeps a per-registry table of loaded module name. If a resolved module path is not in the table, and #f is not provided as the third argument to the module name resolver, then the name is put into the table and the corresponding file is loaded with a variant of load/use-compiled that passes the expected module name to the compiled-load handler.

While loading a file, the default module name resolver sets the current-module-declare-name parameter to the resolved module name. Also, the default module name resolver records in a private continuation mark the filename being loaded, and it checks whether such a mark already exists; if such a continuation mark does exist in the current continuation, then the exn:fail exception is raised with a message about a dependency cycle.

Module loading is suppressed (i.e., #f is supplied as a third argument to the module name resolver) when resolving module paths in syntax objects (see §1.2.2 "Syntax Objects"). When a syntax object is manipulated, the current namespace might not match the original namespace for the syntax object, and the module should not necessarily be loaded in the current namespace.

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

```
(current-module-declare-name)
 → (or/c resolved-module-path? false/c)
(current-module-declare-name name) → void?
  name : (or/c resolved-module-path? false/c)
```

A parameter that determines a module name that is used when evaluating a `module` declaration (when the parameter value is not #f). In that case, the *id* from the `module` declaration is ignored, and the parameter's value is used as the name of the declared module.

13.4.2 Compiled Modules and References

While expanding a `module` declaration, the expander resolves module paths for imports to load module declarations as necessary and to determine imported bindings, but the compiled form of a `module` declaration preserves the original module path. Consequently, a compiled module can be moved to another filesystem, where the module name resolver can resolve inter-module references among compiled code.

When a module reference is extracted from compiled form (see module-compiled-imports) or from syntax objects in macro expansion (see §11.2 "Syntax Object Content"), the module reference is reported in the form of a *module path index*. A module path index is a semi-interned (multiple references to the same relative module tend to use the same module path index value, but not always) opaque value that encodes a module path (see `module-path?`) and another module path index to which it is relative.

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

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

```
(module-path-index? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a module path index, `#f` otherwise.

```
(module-path-index-resolve mpi) → resolved-module-path?
  mpi : module-path-index?
```

Returns a resolved module path for the resolved module name, computing the resolved name (and storing it in *mpi*) if it has not been computed before.

Resolving a module path index uses the current module name resolver (see `current-module-name-resolver`). Depending on the kind of module paths encapsulated by *mpi*, the computed resolved name can depend on the value of `current-load-relative-directory` or `current-directory`.

```
(module-path-index-split mpi)
 → (or/c module-path? false/c)
   (or/c module-path-index? false/c)
  mpi : module-path-index?
```

Returns two values: a module path, and a base module path index or `#f` to which the module path is relative.

A `#f` second result means that the path is relative to an unspecified directory (i.e., its resolution depends on the value of `current-load-relative-directory` and/or `current-directory`).

A `#f` for the first result implies a `#f` for the second result, and means that *mpi* represents "self" (see above).

```
(module-path-index-join path mpi) → module-path-index?
  path : (or/c module-path? false/c)
  mpi : (or/c module-path-index? false/c)
```

Combines *path* and *mpi* to create a new module path index. The *path* argument can `#f` only if *mpi* is also `false`.

```
(compiled-module-expression? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a compiled `module` declaration, `#f` otherwise. See also `current-compile`.

```
(module-compiled-name compiled-module-code) → symbol?
  compiled-module-code : compiled-module-expression?
```

Takes a module declaration in compiled form and returns a symbol for the module's declared name.

```
(module-compiled-imports compiled-module-code)
 → (listof (cons/c (or/c exact-integer? false/c)
                   (listof module-path-index?)))
  compiled-module-code : compiled-module-expression?
```

Takes a module declaration in compiled form and returns an association list mapping phase level shifts (where `#f` corresponds to a shift into the label phase level) to module references for the module's explicit imports.

```
(module-compiled-exports compiled-module-code)
 → (listof (cons/c (or/c exact-integer? false/c) list?))
   (listof (cons/c (or/c exact-integer? false/c) list?))
  compiled-module-code : compiled-module-expression?
```

Returns two association lists mapping phase level values (where #f corresponds to the label phase level) to exports at the corresponding phase. The first association list is for exported variables, and the second is for exported syntax.

Each associated list more precisely matches the contract

```
(listof (list/c symbol?
                (listof
                 (or/c module-path-index?
                       (list/c module-path-index?
                               (or/c exact-integer? false/c)
                               symbol?
                               (or/c exact-integer? false/c))))))
```

For each element of the list, the leading symbol is the name of the export.

The second part—the list of module path index values, etc.—describes the origin of the exported identifier. If the origin list is `null`, then the exported identifier is defined in the module. If the exported identifier is re-exported, instead, then the origin list provides information on the import that was re-exported. The origin list has more than one element if the binding was imported multiple times from (possibly) different sources.

For each origin, a module path index by itself means that the binding was imported with a phase level shift of 0 (i.e., a plain `require` without `for-meta`, `for-syntax`, etc.), and imported identifier has the same name as the re-exported name. An origin represented with a list indicates explicitly the import, the import phase level shift (where #f corresponds to a `for-label` import), the import name of the re-exported binding, and the phase level of the import.}

13.4.3 Dynamic Module Access

```
(dynamic-require mod provided) → any
  mod : module-path?
  provided : (or/c symbol? false/c void?)
```

Dynamically instantiates the module specified by *mod* for phase 0 in the current namespace's registry, if it is not yet instantiated. If *mod* is not a symbol, the current module name resolver may load a module declaration to resolve it (see `current-module-name-resolver`); the path is resolved relative to `current-load-relative-directory` and/or `current-directory`.

If *provided* is #f, then the result is `#<void>`. Otherwise, when *provided* is a symbol, the value of the module's export with the given name is returned. If the module exports `provide` as syntax, then a use of the binding is expanded and evaluated (in a fresh namespace to which the module is attached). If the module has no such exported variable or syntax, or if the variable is protected (see §13.9 "Code Inspectors"), the `exn:fail:contract` exception is raised.

If *provided* is `#<void>`, then the module is visited (see §1.2.3.8 "Module Phases"), but not instantiated. The result is `#<void>`.

```
(dynamic-require-for-syntax mod provided) → any
  mod : module-path?
  provided : (or/c symbol? false/c)
```

Like `dynamic-require`, but in phase 1.

13.5 Security Guards

```
(security-guard? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a security guard value as created by `make-security-guard`, `#f` otherwise.

A *security guard* provides a set of access-checking procedures to be called when a thread initiates access of a file, directory, or network connection through a primitive procedure. For example, when a thread calls `open-input-file`, the thread's current security guard is consulted to check whether the thread is allowed read access to the file. If access is granted, the thread receives a port that it may use indefinitely, regardless of changes to the security guard (although the port's custodian could shut down the port; see §13.6 "Custodians").

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

```
(make-security-guard  parent
                      file-guard
                      network-guard
                     [link])         → security-guard?
  parent : security-guard?
  file-guard : (symbol?
                (or/c path? false/c)
                (listof symbol?)
                . -> . any)
```

```
network-guard : (symbol?
                 (or/c (and/c string? immutable?) false/c)
                 (or/c (integer-in 1 65535) false/c)
                 (one-of/c 'server 'client)
                 . -> . any)
link : (or/c (symbol? path? path? . -> . any) = #f
             false/c)
```

Creates a new security guard as child of *parent*.

The *file-guard* procedure must accept three arguments:

- a symbol for the primitive procedure that triggered the access check, which is useful for raising an exception to deny access.
- a path (see §14.1 "Paths") or `#f` for pathless queries, such as `(current-directory)`, `(filesystem-root-list)`, and `(find-system-path symbol)`. A path provided to *file-guard* is not expanded or otherwise normalized before checking access; it may be a relative path, for example.
- a list containing one or more of the following symbols:
  - `'read` — read a file or directory
  - `'write` — modify or create a file or directory
  - `'execute` — execute a file
  - `'delete` — delete a file or directory
  - `'exists` — determine whether a file or directory exists, or that a path string is well-formed

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

The *network-guard* procedure must accept four arguments:

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

- a symbol, either `'client` or `'server`, indicating whether the check is for the creation of a client connection or a listening server. The opening of an unbound UDP socket is identified as a `'client` connection; explicitly binding the socket is identified as a `'server` action.

The `link-guard` argument can be `#f` or a procedure of three arguments:

- a symbol for the primitive procedure that triggered the access check, which is useful for raising an exception to deny access.
- a complete path (see §14.1 "Paths") representing the file to create as link.
- a path representing the content of the link, which may be relative the second-argument path; this path is not expanded or otherwise normalized before checking access.

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

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

The *`file-guard`*, *`network-guard`*, and `link-guard` procedures are invoked in the thread that called the access-checked primitive. Breaks may or may not be enabled (see §9.6 "Breaks"). Full continuation jumps are blocked going into or out of the *`file-guard`* or *`network-guard`* call (see §1.1.12 "Prompts, Delimited Continuations, and Barriers").

```
(current-security-guard) → security-guard?
(current-security-guard guard) → void?
  guard : security-guard?
```

A parameter that determines the current security guard that controls access to the filesystem and network.

13.6 Custodians

See §1.1.16 "Custodians" for basic information on the PLT Scheme custodian model.

```
(custodian? v) → boolean?
  v : any/c
```

Returns `#t` if *`v`* is a custodian value, `#f` otherwise.

```
(make-custodian [cust]) → custodian?
  cust : custodian? = (current-custodian)
```

Creates a new custodian that is subordinate to `cust`. When `cust` 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.

```
(custodian-shutdown-all cust) → void?
  cust : custodian?
```

Closes all open ports and closes all active TCP listeners and UDP sockets that are managed by `cust`. It also removes `cust` (and its subordinates) as managers of all threads; when a thread has no managers, it is killed (or suspended; see `thread/suspend-to-kill`) If the current thread is to be killed, all other shut-down actions take place before killing the thread.

```
(current-custodian) → custodian?
(current-custodian cust) → void?
  cust : custodian?
```

A parameter that determines a custodian that assumes responsibility for newly created threads, ports, TCP listeners, UDP sockets, and byte converters.

```
(custodian-managed-list cust super) → list?
  cust : custodian?
  super : custodian?
```

Returns a list of immediately managed objects and subordinate custodians for `cust`, where `cust` is itself subordinate to `super` (directly or indirectly). If `cust` is not strictly subordinate to `super`, the `exn:fail:contract` exception is raised.

```
(custodian-memory-accounting-available?) → boolean?
```

Returns `#t` if MzScheme is compiled with support for per-custodian memory accounting, `#f` otherwise.

Memory accounting is normally available in PLT Scheme 3m, which is the main variant of PLT Scheme, and not normally available in PLT Scheme CGC.

```
(custodian-require-memory limit-cust
                          need-amt
                          stop-cust) → void?
  limit-cust : custodian?
  need-amt : nonnegative-exact-integer?
  stop-cust : custodian?
```

Registers a require check if PLT Scheme is compiled with support for per-custodian memory accounting, otherwise the `exn:fail:unsupported` exception is raised.

If a check is registered, and if PLT Scheme later reaches a state after garbage collection (see §1.1.7 "Garbage Collection") where allocating *need-amt* bytes charged to *limit-cust* would fail or tigger some shutdown, then *stop-cust* is shut down.

```
(custodian-limit-memory  limit-cust
                         limit-amt
                        [stop-cust]) → void?
  limit-cust : custodian?
  limit-amt : nonnegative-exact-integer?
  stop-cust : custodian? = limit-cust
```

Registers a limit check if PLT Scheme is compiled with support for per-custodian memory accounting, otherwise the `exn:fail:unsupported` exception is raised.

If a check is registered, and if PLT Scheme later reaches a state after garbage collection (see §1.1.7 "Garbage Collection") where *limit-cust* owns more than *limit-amt* bytes, then *stop-cust* is shut down.

For reliable shutdown, *limit-amt* for `custodian-limit-memory` must be much lower than the total amount of memory available (minus the size of memory that is potentially used and not charged to *limit-cust*). Moreover, if indvidual allocations that are initially charged to *limit-cust* can be arbitrarily large, then *stop-cust* must be the same as *limit-cust*, so that excessively large immediate allocations can be rejected with an `exn:fail:out-of-memory` exception.

```
(make-custodian-box cust v) → custodian-box?
  cust : custodian?
  v : any/c
```

Returns a *custodian box* that contains *v* as long as *cust* has not been shut down.

```
(custodian-box? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a custodian box produced by `make-custodian-box`, `#f` otherwise.

```
(custodian-box-value cb) → any
  cb : custodian-box?
```

Rturns the value in the given custodian box, or `#f` if the value has been removed.

13.7 Thread Groups

A *thread group* is a collection of threads and other thread groups that have equal claim to the CPU. By nesting thread groups and by creating certain threads within certain groups, a programmer can control the amount of CPU allocated to a set of threads. Every thread belongs to a thread group, which is determined by the `current-thread-group` parameter when the thread is created. Thread groups and custodians (see §13.6 "Custodians") are independent.

The root thread group receives all of the CPU that the operating system gives Scheme. Every thread or nested group in a particular thread group receives equal allocation of the CPU (a portion of the group's access), although a thread may relinquish part of its allocation by sleeping or synchronizing with other processes.

```
(make-thread-group [group]) → thread-group?
  group : thread-group? = (current-thread-group)
```

Creates a new thread group that belongs to *group*.

```
(thread-group? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a thread group value, `#f` otherwise.

```
(current-thread-group) → thread-group?
(current-thread-group group) → void?
  group : thread-group?
```

A parameter that determines the thread group for newly created threads.

13.8 Structure Inspectors

An *inspector* provides access to structure fields and structure type information without the normal field accessors and mutators. (Inspectors are also used to control access to module bindings; see §13.9 "Code Inspectors".) Inspectors are primarily intended for use by debuggers.

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

The `current-inspector` parameter determines a default inspector argument for new structure types. An alternate inspector can be provided though the `#:inspector` option of the `define-struct` form (see §4.1 "Defining Structure Types: `define-struct`"), or through an optional `inspector` argument to `make-struct-type`.

```
(inspector? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is an inspector, `#f` otherwise.

```
(make-inspector [inspector]) → inspector?
  inspector : inspector? = (current-inspector)
```

Returns a new inspector that is a subinspector of *inspector*. Any structure type controlled by the new inspector is also controlled by its ancestor inspectors, but no other inspectors.

```
(make-sibling-inspector [inspector]) → inspector?
  inspector : inspector? = (current-inspector)
```

Returns a new inspector that is a subinspector of the same inspector as *inspector*. That is, *inspector* and the result inspector control mutually disjoint sets of structure types.

```
(current-inspector) → inspector?
(current-inspector insp) → void?
  insp : inspector?
```

A parameter that determines the default inspector for newly created structure types.

```
(struct-info v) → (or/c struct-type? false/c) boolean?
  v : any/c
```

Returns two values:

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

```
(struct-type-info struct-type)
 → symbol?
   nonnegative-exact-integer?
   nonnegative-exact-integer?
   struct-accessor-procedure?
   struct-mutator-procedure?
   (listof nonnegative-exact-integer?)
   (or/c struct-type? false/c)
   boolean?
  struct-type : struct-type?
```

Returns eight values that provide information about the structure type descriptor *struct-type*, assuming that the type is controlled by the current inspector:

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

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

```
(struct-type-make-constructor struct-type)
 → struct-constructor-procedure?
  struct-type : struct-type?
```

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

```
(struct-type-make-predicate struct-type) → any
  struct-type : any/c
```

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

```
(object-name v) → any
  v : any/c
```

Returns a value for the name of *v* if *v* has a name, `#f` otherwise. The argument *v* can be any value, but only (some) procedures, structs, struct types, struct type properties, regexp values, and ports have names. The name of a procedure, struct, struct type, or struct type property is always a symbol. The name of a regexp value is a string, and a byte-regexp value's name is a byte string. The name of a port is typically a path or a string, but it can be arbitrary. See also §1.2.6 "Inferred Value Names".

13.9 Code Inspectors

In the same way that inspectors control access to structure fields (see §13.8 "Structure Inspectors"), inspectors also control access to module bindings. The default inspector for module bindings is determined by the `current-code-inspector` parameter, instead of the `current-inspector` parameter.

When a `module` declaration is evaluated, the value of the `current-code-inspector` parameter is associated with the module declaration. When the module is invoked via `require` or `dynamic-require`, a sub-inspector of the module's declaration-time inspector is created, and this sub-inspector is associated with the module invocation. Any inspector that controls the sub-inspector (i.e., the declaration-time inspector and its superior) controls the module invocation.

Control over a module invocation enables

- the use of `module->namespace` on the module;
- access to the module's protected identifiers, i.e. those identifiers exported from the module with `protect`; and
- access to the module's protected and unexported variables within compiled code from `read` (see `current-compile`).

If the value of `current-code-inspector` never changes, then no control is lost for any module invocation, since the module's invocation is associated with a sub-inspector of `current-code-inspector`.

The inspector for a module invocation is specific to a particular module registry, in case a module is attached to a new registry via `namespace-attach-module`. The invocation inspector in a particular registry can be changed via `namespace-unprotect-module` (but changing the inspector requires control over the old one).

Control over a module declaration (as opposed to a mere invocation) enables the reconstruction of syntax objects that contain references to the module's unexported identifiers. Otherwise, the compiler and macro expander prevent any reference to an unexported identifier, unless the reference appears within an expression that was generated by the module's macros (or, more precisely, a macro from a module whose declaration inspector controls the invocation of the identifier's module). See §11.7 "Syntax Certificates" for further information.

```
(current-code-inspector) → inspector?
(current-code-inspector insp) → void?
  insp : inspector?
```

A parameter that determines an inspector to control access to module bindings and redefinitions.

13.10 Sandboxed Evaluation

```
(require scheme/sandbox)
```

The bindings documented in this section are provided by the `scheme/sandbox` library, not `scheme/base` or `scheme`.

The `scheme/sandbox` module provides utilities for creating "sandboxed" evaluators, which are configured in a particular way and can have restricted resources (memory and time), filesystem access, and network access.

```
(make-evaluator language
                input-program ...
                #:requires requires
                #:allow-read allow) → (any/c . -> . any)
  language : (or/c module-path?
                   (list/c (one-of/c 'special) symbol?)
                   (cons/c (one-of/c 'begin) list?))
  input-program : any/c
```

```
  requires : (listof (or/c module-path? path?))
  allow : (listof (or/c module-path? path?))
(make-module-evaluator module-decl
                        #:allow-read allow) → (any/c . -> . any)
  module-decl : (or/c syntax? pair?)
  allow : (listof (or/c module-path? path?))
```

The `make-evaluator` function creates an evaluator with a *`language`* and *`requires`* specification, and starts evaluating the given *`input-program`*s. The `make-module-evaluator` function creates an evaluator that works in the context of a given module. The result in either case is a function for further evaluation.

The returned evaluator operates in an isolated and limited environment. In particular, filesystem access is restricted. The *`allow`* argument extends the set of files that are readable by the evaluator to include the specified modules and their imports (transitively). When *`language`* is a module path and when *`requires`* is provided, the indicated modules are implicitly included in the *`allow`* list.

Each *`input-program`* or *`module-decl`* argument provides a program in one of the following forms:

- an input port used to read the program;
- a string or a byte string holding the complete input;
- a path that names a file holding the input; or
- an S-expression or a syntax object, which is evaluated as with `eval` (see also `get-uncovered-expressions`).

In the first three cases above, the program is read using `sandbox-reader`, with line-counting enabled for sensible error messages, and with `'program` as the source (used for testing coverage). In the last case, the input is expected to be the complete program, and is converted to a syntax object (using `'program` as the source), unless it already is a syntax object.

The returned evaluator function accepts additional expressions (each time it is called) in essentially the same form: a string or byte string holding a sequence of expressions, a path for a file holding expressions, an S-expression, or a syntax object. If the evaluator receives an `eof` value, it is terminated and raises errors thereafter. See also `kill-evaluator`, which terminates the evaluator without raising an exception.

For `make-evaluator`, multiple *`input-program`*s are effectively concatenated to form a single program. The way that the *`input-program`*s are evaluated depends on the *`language`* argument:

- The *language* argument can be a module path (i.e., a datum that matches the grammar for *module-path* of `require`).

  In this case, the *input-program*s are automatically wrapped in a `module`, and the resulting evaluator works within the resulting module's namespace.

- The *language* argument can be a list starting with `'special`, which indicates a built-in language with special input configuration. The possible values are `'(special r5rs)` or a value indicating a teaching language: `'(special beginner)`, `'(special beginner-abbr)`, `'(special intermediate)`, `'(special intermediate-lambda)`, or `'(special advanced)`.

  In this case, the *input-program*s are automatically wrapped in a `module`, and the resulting evaluator works within the resulting module's namespace. In addition, certain parameters (such as such as `read-accept-infix-dot`) are set to customize reading programs from strings and ports.

  This option is provided mainly for older test systems. Using `make-module-evaluator` with input starting with `#lang` is generally better.

- Finally, *language* can be a list whose first element is `'begin`.

  In this case, a new namespace is created using `sandbox-namespace-specs`, which by default creates a new namespace using `make-base-namespace` or `make-gui-namespace` (depending on `gui?`).

  In the new namespace, *language* is evaluated as an expression to further initialize the namespace.

The *requires* list adds additional imports to the module or namespace for the *input-program*s, even in the case that `require` is not made available through the *language*.

The following examples illustrate the difference between an evaluator that puts the program in a module and one that merely initializes a top-level namespace:

```
> (define base-module-eval
    ; a module cannot have free variables...
    (make-evaluator 'scheme/base '(define (f) later)))
program:1:0: compile: unbound variable in module in: later
> (define base-module-eval
    (make-evaluator 'scheme/base '(define (f) later)
                                 '(define later 5)))
> (base-module-eval '(f))
5
> (define base-top-eval
    ; non-module code can have free variables:
    (make-evaluator '(begin) '(define (f) later)))
> (base-top-eval '(+ 1 2))
3
> (base-top-eval '(define later 5))
```

```
> (base-top-eval '(f))
5
```

The make-module-evaluator function is essentially a restriction of make-evaluator, where the program must be a module, and all imports are part of the program:

```
(define base-module-eval2
  ; equivalent to base-module-eval:
  (make-module-evaluator '(module m scheme/base
                             (define (f) later)
                             (define later 5))))
```

In all cases, the evaluator operates in an isolated and limited environment:

- It uses a new custodian and namespace. When gui? is true, it is also runs in its own eventspace.
- The evaluator works under the sandbox-security-guard, which restricts file system and network access.
- Each evaluation is wrapped in a call-with-limits; see also sandbox-eval-limits and set-eval-limits.

Evaluation can also be instrumented to track coverage information when sandbox-coverage-enabled is set. Exceptions (both syntax and run-time) are propagated as usual to the caller of the evaluation function (i.e., catch it with with-handlers). However, note that a sandboxed evaluator is convenient for testing, since all exceptions happen in the same way, so you don't need special code to catch syntax errors.

Finally, the fact that a sandboxed evaluator accept syntax objects makes it usable as the value for "current-eval", which means that you can easily start a sandboxed read-eval-print-loop. For example, here is a quick implementation of a networked REPL:

```
(define e
  (make-module-evaluator '(module m scheme/base)))
(let-values ([(i o) (tcp-accept (tcp-listen 9999))])
  (parameterize ([current-input-port i]
                 [current-output-port o]
                 [current-error-port o]
                 [current-eval a])
    (read-eval-print-loop)
    (fprintf o "\nBye...\n")
    (close-output-port o)))
```

13.10.1 Customizing Evaluators

The evaluators that `make-evaluator` creates can be customized via several parameters. These parameters affect newly created evaluators; changing them has no effect on already-running evaluators.

```
(sandbox-init-hook) → (-> any)
(sandbox-init-hook thunk) → void?
  thunk : (-> any)
```

A parameter that determines a thunk to be called for initializing a new evaluator. The hook is called just before the program is evaluated in a newly-created evaluator context. It can be used to setup environment parameters related to reading, writing, evaluation, and so on. Certain languages (`'(special r5rs)` and the teaching languages) have initializations specific to the language; the hook is used after that initialization, so it can override settings.

```
(sandbox-reader) → (any/c . -> . any)
(sandbox-reader proc) → void?
  proc : (any/c . -> . any)
```

A parameter that determines a function to reads all expressions from `(current-input-port)`. The function is used to read program source for an evaluator when a string. byte string, or port is supplies. The reader function receives a value to be used as input source (i.e., the first argument to `read-syntax`), and it should return a list of syntax objects. The default reader calls `read-syntax`, accumulating results in a list until it receives `eof`.

```
(sandbox-input) → (or/c false/c
                        string? bytes?
                        input-port?
                        (one-of/c 'pipe)
                        (-> input-port?))
(sandbox-input in) → void?
  in : (or/c false/c
             string? bytes?
             input-port?
             (one-of/c 'pipe)
             (-> input-port?))
```

A parameter that determines the initial `current-input-port` setting for a newly created evaluator. It defaults to `#f`, which creates an empty port. The following other values are allowed:

- a string or byte string, which is converted to a port using `open-input-string` or open-input-bytes;

- an input port;
- the symbol 'pipe, which triggers the creation of a pipe, where put-input can return the output end of the pipe or write directly to it;
- a thunk, which is called to obtain a port (e.g., using current-input-port means that the evaluator input is the same as the calling context's input).

```
(sandbox-output) → (or/c false/c
                         output-port?
                         (one-of/c 'pipe 'bytes 'string)
                         (-> output-port?))
(sandbox-output in) → void?
  in : (or/c false/c
             output-port?
             (one-of/c 'pipe 'bytes 'string)
             (-> output-port?))
```

A parameter that determines the initial current-output-port setting for a newly created evaluator. It defaults to #f, which creates a port that discrds all data. The following other values are allowed:

- an output port, which is used as-is;
- the symbol 'bytes, which causes get-output to return the complete output as a byte string;
- the symbol 'string, which is similar to 'bytes, but makes get-output produce a string;
- the symbol 'pipe, which triggers the creation of a pipe, where get-output returns the input end of the pipe;
- a thunk, which is called to obtain a port (e.g., using current-output-port means that the evaluator output is not diverted).

```
(sandbox-error-output) → (or/c false/c
                               output-port?
                               (one-of/c 'pipe 'bytes 'string)
                               (-> output-port?))
(sandbox-error-output in) → void?
```

```
in : (or/c false/c
           output-port?
           (one-of/c 'pipe 'bytes 'string)
           (-> output-port?))
```

Like `sandbox-output`, but for the initial `current-error-port` value. An evaluator's error output is set after its output, so using `current-output-port` for this parameter value means that the error port is the same as the evaluator's initial output port.

The default is `current-error-port`, which means that the error output of the generated evaluator goes to the calling context's error port.

```
(sandbox-coverage-enabled) → boolean?
(sandbox-coverage-enabled enabled?) → void?
  enabled? : any/c
```

A parameter that controls whether syntactic coverage information is collected by sandbox evaluators. Use `get-uncovered-expressions` to retrieve coverage information.

```
(sandbox-propagate-breaks) → boolean?
(sandbox-propagate-breaks propagate?) → void?
  propagate? : any/c
```

When this boolean parameter is true, breaking while an evaluator is running evaluator propagates the break signal to the sandboxed context. This makes the sandboxed evaluator break, typically, but beware that sandboxed evaluation can capture and avoid the breaks (so if safe execution of code is your goal, make sure you use it with a time limit). The default is `#t`.

```
(sandbox-namespace-specs) → (cons/c (-> namespace?)
                                    (listof module-path?))
(sandbox-namespace-specs spec) → void?
  spec : (cons/c (-> namespace?)
                 (listof module-path?))
```

A parameter that holds a list of values that specify how to create a namespace for evaluation in `make-evaluator` or `make-module-evaluator`. The first item in the list is a thunk that creates the namespace, and the rest are module paths for modules that to be attached to the created namespace using `namespace-attach-module`.

The default is `(list make-base-namespace)` if `gui?` is `#f`, `(list make-gui-namespace)` if `gui?` is `#t`.

The module paths are needed for sharing module instantiations between the sandbox and the caller. For example, sandbox code that returns `posn` values (from the `lang/posn` module) will not be recognized as such by your own code by default, since the sandbox will have its own instance of `lang/posn` and thus its own struct type for `posn`s. To be able to use such values, include `'lang/posn` in the list of module paths.

When testing code that uses a teaching language, the following piece of code can be helpful:

```
(sandbox-namespace-specs
 (let ([specs (sandbox-namespace-specs)])
   `(,(car specs)
     ,@(cdr specs)
     lang/posn
     ,@(if gui? '(mrlib/cache-image-snip) '()))))
```

```
(sandbox-override-collection-paths) → (listof path-string?)
(sandbox-override-collection-paths paths) → void?
  paths : (listof path-string?)
```

A parameter that determines a list of collection directories to prefix `current-library-collection-paths` in an evaluator. This parameter is useful for cases when you want to test code using an alternate, test-friendly version of a collection, for example, testing code that uses GUI (like the `htdp/world` teachpack) can be done using a fake library that provides the same interface but no actual interaction. The default is `null`.

```
(sandbox-security-guard) → security-guard?
(sandbox-security-guard guard) → void?
  guard : security-guard?
```

A parameter that determines the initial `(current-security-guard)` for sandboxed evaluations. The default forbids all filesystem I/O except for things in `sandbox-path-permissions`, and it uses `sandbox-network-guard` for network connections.

```
(sandbox-path-permissions)
 → (listof (list/c (one-of/c 'execute 'write 'delete
                             'read 'exists)
                   (or/c byte-regexp? bytes? string? path?)))
(sandbox-path-permissions perms) → void?
  perms : (listof (list/c (one-of/c 'execute 'write 'delete
                                    'read 'exists)
                          (or/c byte-regexp? bytes? string? path?)))
```

A parameter that configures the behavior of the default sandbox security guard by listing paths and access modes that are allowed for them. The contents of this parameter is a list of specifications, each is an access mode and a byte-regexp for paths that are granted this access.

The access mode symbol is one of: `'execute`, `'write`, `'delete`, `'read`, or `'exists`. These symbols are in decreasing order: each implies access for the following modes too (e.g., `'read` allows reading or checking for existence).

The path regexp is used to identify paths that are granted access. It can also be given as a path (or a string or a byte string), which is (made into a complete path, cleansed, simplified, and then) converted to a regexp that allows the path and sub-directories; e.g., `"/foo/bar"` applies to `"/foo/bar/baz"`.

The default value is null, but when an evaluator is created, it is augmented by `'read` permissions that make it possible to use collection libraries (including `sandbox-override-collection-paths`). See `make-evalautor` for more information.

```
(sandbox-network-guard)
 → (symbol?
     (or/c (and/c string? immutable?) false/c)
     (or/c (integer-in 1 65535) false/c)
     (one-of/c 'server 'client)
     . -> . any)
(sandbox-network-guard proc) → void?
  proc : (symbol?
          (or/c (and/c string? immutable?) false/c)
          (or/c (integer-in 1 65535) false/c)
          (one-of/c 'server 'client)
          . -> . any)
```

A parameter that specifieds a procedure to be used (as is) by the default `sandbox-security-guard`. The default forbids all network connection.

```
(sandbox-eval-limits)
 → (or/c (list/c (or/c exact-nonnegative-integer? false/c)
                 (or/c exact-nonnegative-integer? false/c))
         false/c)
(sandbox-eval-limits limits) → void?
  limits : (or/c (list/c (or/c exact-nonnegative-integer? false/c)
                         (or/c exact-nonnegative-integer? false/c))
                 false/c)
```

A parameter that determines the default limits on *each* use of a `make-evaluator` function, including the initial evaluation of the input program. Its value should be a list of two numbers, the first is a timeout value in seconds, and the second is a memory limit in megabytes. Either one can be `#f` for disabling the corresponding limit; alternately, the parameter can be set to `#f` to disable all limits (in case more are available in future versions). The default is `(list 30 20)`.

When limits are set, `call-with-limits` (see below) is wrapped around each use of the evaluator, so consuming too much time or memory results in an exception. Change the limits of a running evaluator using `set-eval-limits`.

```
(sandbox-make-inspector) → (-> inspector?)
(sandbox-make-inspector make) → void?
  make : (-> inspector?)
```

A parameter that determines the procedure used to create the inspector for sandboxed evaluation. The procedure is called when initializing an evaluator, and the default parameter value is `make-inspector`.

13.10.2 Interacting with Evaluators

The following functions are used to interact with a sandboxed evaluator in addition to using it to evaluate code.

```
(kill-evaluator evaluator) → void?
  evaluator : (any/c . -> . any)
```

Releases the resources that are held by *evaluator* by shutting down the evaluator's custodian. Attempting to use an evaluator after killing raises an exception, and attempts to kill a dead evaluator are ignored.

Killing an evaluator is similar to sending an `eof` value to the evaluator, except that an `eof` value will raise an error immediately.

```
(break-evaluator evaluator) → void?
  evaluator : (any/c . -> . any)
```

Sends a break to the running evaluator. The effect of this is as if Ctrl-C was typed when the evaluator is currently executing, which propagates the break to the evaluator's context.

```
(set-eval-limits evaluator secs mb) → void?
  evaluator : (any/c . -> . any)
  secs : (or/c exact-nonnegative-integer? false/c)
  mb : (or/c exact-nonnegative-integer? false/c)
```

Changes the per-expression limits that *evaluator* uses to `sec` seconds and *mb* megabytes (either one can be `#f`, indicating no limit).

This procedure should be used to modify an existing evaluator limits, because changing the sandbox-eval-limits parameter does not affect existing evaluators. See also call-with-limits.

```
(put-input evaluator) → output-port?
  evaluator : (any/c . -> . any)
(put-input evaluator i/o) → void?
  evaluator : (any/c . -> . any)
  i/o : (or/c bytes? string? eof-object?)
```

If (sandbox-input) is 'pipe when an evaluator is created, then this procedure can be used to retrieve the output port end of the pipe (when used with no arguments), or to add a string or a byte string into the pipe. It can also be used with eof, which closes the pipe.

```
(get-output evaluator) → (or/c input-port? bytes? string?)
  evaluator : (any/c . -> . any)
(get-error-output evaluator)
 → (or/c input-port? bytes? string?)
  evaluator : (any/c . -> . any)
```

Returns the output or error-output of the *evaluator*, in a way that depends on the setting of (sandbox-output) or (sandbox-error-output) when the evaluator was created:

- if it was 'pipe, then get-output returns the input port end of the created pipe;
- if it was 'bytes or 'string, then the result is the accumulated output, and the output is directed to a new output string or byte string (so each call returns a different piece of the evaluator's output);
- otherwise, it returns #f.

```
(get-uncovered-expressions evaluator
                          [prog?
                           src])     → (listof syntax?)
  evaluator : (any/c . -> . any)
  prog? : any/c = #t
  src : any/c = 'program
```

Retrieves uncovered expression from an evaluator, as longs as the sandbox-coverage-enabled parameter had a true value when the evaluator was created. Otherwise, and exception is raised to indicate that no coverage information is available.

The *prog?* argument specifies whether to obtain expressions that were uncovered after only the original input program was evaluated (#t) or after all later uses of the evaluator (#f).

Using #t retrieves a list that is saved after the input program is evaluated, and before the evaluator is used, so the result is always the same.

A #t value of *prog?* is useful for testing student programs to find out whether a submission has sufficient test coverage built in. A #f value is useful for writing test suites for a program to ensure that your tests cover the whole code.

The second optional argument, *src*, specifies that the result should be filtered to hold only syntax objects whose source matches *src*. The default, 'program, is the source associated with the input program by the default sandbox-reader—which provides only syntax objects from the input program (and not from required modules or expressions that were passed to the evaluator). A #f avoids filtering.

The resulting list of syntax objects has at most one expression for each position and span. Thus, the contents may be unreliable, but the position information is reliable (i.e., it always indicates source code that would be painted red in DrScheme when coverage information is used).

Note that if the input program is a sequence of syntax values, either make sure that they have 'program as the source field, or use the *src* argument. Using a sequence of S-expressions (not syntax objects) for an input program leads to unreliable coverage results, since each expression may be assigned a single source location.

13.10.3 Miscellaneous

```
gui? : boolean?
```

True if the scheme/gui module can be used, #f otherwise; see gui-available?.

Various aspects of the scheme/sandbox library change when the GUI library is available, such as using a new eventspace for each evaluator.

```
(call-with-limits secs mb thunk) → any
  secs : (or/c exact-nonnegative-integer? false/c)
  mb : (or/c exact-nonnegative-integer? false/c)
  thunk : (-> any)
```

Executes the given *thunk* with memory and time restrictions: if execution consumes more than *mb* megabytes or more than sec seconds, then the computation is aborted and the exn:fail:resource exception is raised. Otherwise the result of the thunk is returned as usual (a value, multiple values, or an exception). Each of the two limits can be #f to indicate the absence of a limit. See also custodian-limit-memory for information on memory limits.

Sandboxed evaluators use `call-with-limits`, according to the `sandbox-eval-limits` setting and uses of `set-eval-limits`: each expression evaluation is protected from time-outs and memory problems. Use `call-with-limits` directly only to limit a whole testing session, instead of each expression.

```
(with-limits mb-expr body-expr body ...)
```

A macro version of `call-with-limits`.

```
(exn:fail:resource? v) → boolean?
  v : any/c
(exn:fail:resource-resource exn) → (one-of/c 'time 'memory)
  exn : exn:fail:resource?
```

A predicate and accessor for exceptions that are raised by `call-with-limits`. The `resource` field holds a symbol, either `'time` or `'memory`.

14 Operating System

14.1 Paths

When a Scheme procedure takes a filesystem path as an argument, the path can be provided either as a string or as an instance of the *path* datatype. If a string is provided, it is converted to a path using `string->path`. A Scheme procedure that generates a filesystem path always generates a path value.

By default, paths are created and manipulated for the current platform, but procedures that merely manipulate paths (without using the filesystem) can manipulate paths using conventions for other supported platforms. The `bytes->path` procedure accepts an optional argument that indicates the platform for the path, either `'unix` or `'windows`. For other functions, such as `build-path` or `simplify-path`, the behavior is sensitive to the kind of path that is supplied. Unless otherwise specified, a procedure that requires a path accepts only paths for the current platform.

Two path values are `equal?` when they are use the same convention type and when their byte-string representations are `equal?`. A path string (or byte string) cannot be empty, and it cannot contain a nul character or byte. When an empty string or a string containing nul is provided as a path to any procedure except `absolute-path?`, `relative-path?`, or `complete-path?`, the `exn:fail:contract` exception is raised.

Most Scheme primitives that accept paths first *cleanse* the path before using it. Procedures that build paths or merely check the form of a path do not cleanse paths, with the exceptions of `cleanse-path`, `expand-user-path`, and `simplify-path`. For more information about path cleansing and other platform-specific details, see §14.1.3 "Unix and Mac OS X Paths" for Unix and Mac OS X paths and §14.1.4 "Windows Path Conventions" for Windows paths.

14.1.1 Manipulating Paths

```
(path? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a path value for the current platform (not a string, and not a path for a different platform), `#f` otherwise.

```
(path-string? v) → boolean?
  v : any/c
```

Return `#t` if *v* is either a path value for the current platform or a non-empty string without nul characters, `#f` otherwise.

```
(path-for-some-system? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a path value for some platform (not a string), `#f` otherwise.

```
(string->path str) → path?
  str : string?
```

Produces a path whose byte-string name is `(string->bytes/locale string (char->integer #\?))`.

Beware that the current locale might not encode every string, in which case `string->path` can produce the same path for different *str*s. See also `string->path-element`, which should be used instead of `string->path` when a string represents a single path element.

```
(bytes->path bstr [type]) → path?
  bstr : bytes?
  type : (one-of/c 'unix 'windows)
       = (system-path-convention-type)
```

Produces a path (for some platform) whose byte-string name is *bstr*. The optional *type* specifies the convention to use for the path.

For converting relative path elements from literals, use instead `bytes->path-element`, which applies a suitable encoding for individual elements.

```
(path->string path) → string?
  path : path?
```

Produces a string that represents *path* by decoding *path*'s byte-string name using the current locale's encoding; `?` is used in the result string where encoding fails, and if the encoding result is the empty string, then the result is `"?"`.

The resulting string is suitable for displaying to a user, string-ordering comparisons, etc., but it is not suitable for re-creating a path (possibly modified) via `string->path`, since decoding and re-encoding the path's byte string may lose information.

Furthermore, for display and sorting based on individual path elements (such as pathless file names), use `path-element->string`, instead, to avoid special encodings use to represent some relative paths. See §14.1.4 "Windows Path Conventions" for specific information about the conversion of Windows paths.

```
(path->bytes path) → bytes?
```

path : `path?`

Produces *path*'s byte string representation. No information is lost in this translation, so that `(bytes->path (path->bytes path) (path-convention-type path))` always produces a path is that is `equal?` to *path*. The *path* argument can be a path for any platform.

Conversion to and from byte values is useful for marshaling and unmarshaling paths, but manipulating the byte form of a path is generally a mistake. In particular, the byte string may start with a `\\?\REL` encoding for Windows paths. Instead of `path->bytes`, use `split-path` and `path-element->bytes` to manipulate individual path elements.

`(string->path-element str)` → `path?`
  str : `string?`

Like `string->path`, except that *str* corresponds to a single relative element in a path, and it is encoded as necessary to convert it to a path. See §14.1.3 "Unix and Mac OS X Paths" for more information on the conversion for Unix and Mac OS X paths, and see §14.1.4 "Windows Path Conventions" for more information on the conversion for Windows paths.

If *str* does not correspond to any path element (e.g., it is an absolute path, or it can be split), or if it corresponds to an up-directory or same-directory indicator under Unix and Mac OS X, then `exn:fail:contract` exception is raised.

As for `path->string`, information can be lost from *str* in the locale-specific conversion to a path.

`(bytes->path-element bstr [type])` → `path?`
  bstr : `bytes?`
  type : `(one-of/c 'unix 'windows)`
       = `(system-path-convention-type)`

Like `bytes->path`, except that *bstr* corresponds to a single relative element in a path. In terms of conversions and restrictions on *bstr*, `bytes->path-element` is like `string->path-element`.

The `bytes->path-element` procedure is generally the best choice for reconstructing a path based on another path (where the other path is deconstructed with `split-path` and `path-element->bytes`) when ASCII-level manipulation of path elements is necessary.

`(path-element->string path)` → `string?`
  path : `path?`

Like `path->string`, except any encoding prefix is removed. See §14.1.3 "Unix and Mac

OS X Paths" for more information on the conversion for Unix and Mac OS X paths, and see §14.1.4 "Windows Path Conventions" for more information on the conversion for Windows paths. In addition, trailing path separators are removed, as by `split-path`.

The *path* argument must be such that `split-path` applied to *path* would return `'relative` as its first result and a path as its second result, otherwise the `exn:fail:contract` exception is raised.

The `path-element->string` procedure is generally the best choice for presenting a pathless file or directory name to a user.

```
(path-element->bytes path) → bytes?
  path : path-string?
```

Like `path->bytes`, except that any encoding prefix is removed, etc., as for `path-element->string`.

For any reasonable locale, consecutive ASCII characters in the printed form of *path* are mapped to consecutive byte values that match each character's code-point value, and a leading or trailing ASCII character is mapped to a leading or trailing byte, respectively. The *path* argument can be a path for any platform.

The `path-element->bytes` procedure is generally the right choice (in combination with `split-path`) for extracting the content of a path to manipulate it at the ASCII level (then reassembling the result with `bytes->path-element` and `build-path`).

```
(path-convention-type path) → (one-of 'unix 'windows)
  path : path?
```

Accepts a path value (not a string) and returns its convention type.

```
(system-path-convention-type) → (one-of 'unix 'windows)
```

Returns the path convention type of the current platform: `'unix` for Unix and Mac OS X, `'windows` for Windows.

```
(build-path base sub ...) → path?
  base : (or/c path-string?
               (one-of/c 'up 'same))
  sub : (or/c (and/c path-string?
                     (not/c complete-path?))
              (one-of/c 'up 'same))
```

Creates a path given a base path and any number of sub-path extensions. If *base* is an absolute path, the result is an absolute path, otherwise the result is a relative path.

The *base* and each *sub* must be either a relative path, the symbol 'up (indicating the relative parent directory), or the symbol 'same (indicating the relative current directory). For Windows paths, if *base* is a drive specification (with or without a trailing slash) the first *sub* can be an absolute (driveless) path. For all platforms, the last *sub* can be a filename.

The *base* and sub-paths arguments can be paths for any platform. The platform for the resulting path is inferred from the *base* and *sub* arguments, where string arguments imply a path for the current platform. If different arguments are for different platforms, the exn:fail:contract exception is raised. If no argument implies a platform (i.e., all are 'up or 'same), the generated path is for the current platform.

Each *sub* and *base* can optionally end in a directory separator. If the last *sub* ends in a separator, it is included in the resulting path.

If *base* or *sub* is an illegal path string (because it is empty or contains a nul character), the exn:fail:contract exception is raised.

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

See §14.1.3 "Unix and Mac OS X Paths" for more information on the construction of Unix and Mac OS X paths, and see §14.1.4 "Windows Path Conventions" for more information on the construction of Windows paths.

The following examples assume that the current directory is \File{/home/joeuser} for Unix examples and \File{C:\Joe's Files} for Windows examples.

```
(define p1 (build-path (current-directory) "src" "scheme"))
 ; Unix: p1 is "/home/joeuser/src/scheme"
 ; Windows: p1 is "C:\\Joe's Files\\src\\scheme"
(define p2 (build-path 'up 'up "docs" "Scheme"))
 ; Unix: p2 is "../../docs/Scheme"
 ; Windows: p2 is "..\\..\\docs\\Scheme"
(build-path p2 p1)
 ; Unix and Windows: raises exn:fail:contract; p1 is absolute
(build-path p1 p2)
 ; Unix: is "/home/joeuser/src/scheme/../../docs/Scheme"
 ; Windows: is "C:\\Joe's Files\\src\\scheme\\..\\..\\docs\\Scheme"
```

```
(build-path/convention-type type
                            base
                            sub ...) → path?
  type : (one-of/c 'unix 'windows)
  base : path-string?
```

```
sub : (or/c path-string?
            (one-of/c 'up 'same))
```

Like `build-path`, except a path convention type is specified explicitly.

```
(absolute-path? path) → boolean?
  path : path-string?
```

Returns `#t` if *`path`* is an absolute path, `#f` otherwise. The *`path`* argument can be a path for any platform. If *`path`* is not a legal path string (e.g., it contains a nul character), `#f` is returned. This procedure does not access the filesystem.

```
(relative-path? path) → boolean?
  path : path-string?
```

Returns `#t` if *`path`* is a relative path, `#f` otherwise. The *`path`* argument can be a path for any platform. If *`path`* is not a legal path string (e.g., it contains a nul character), `#f` is returned. This procedure does not access the filesystem.

```
(complete-path? path) → boolean?
  path : path-string?
```

Returns `#t` if *`path`* is a completely determined path (*not* relative to a directory or drive), `#f` otherwise. The *`path`* argument can be a path for any platform. Note that for Windows paths, an absolute path can omit the drive specification, in which case the path is neither relative nor complete. If *`path`* is not a legal path string (e.g., it contains a nul character), `#f` is returned.

This procedure does not access the filesystem.

```
(path->complete-path path [base]) → path?
  path : path-string?
  base : path-string? = (current-directory)
```

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`*. If *`base`* is not a complete path, the `exn:fail:contract` exception is raised.

The *`path`* and *`base`* arguments can paths for any platform; if they are for different platforms, the `exn:fail:contract` exception is raised.

This procedure does not access the filesystem.

```
(path->directory-path path) → path?
  path : path-string?
```

Returns *path* if *path* syntactically refers to a directory and ends in a separator, otherwise it returns an extended version of *path* that specifies a directory and ends with a separator. For example, under Unix and Mac OS X, the path `"x/y/"` syntactically refers to a directory and ends in a separator, but `"x/y"` would be extended to `"x/y/"`, and `"x/.."` would be extended to `"x/../"`. The *path* argument can be a path for any platform, and the result will be for the same platform.

This procedure does not access the filesystem.

```
(resolve-path path) → path?
  path : path-string?
```

Cleanses *path* and returns a path that references the same file or directory as *path*. Under Unix and Mac OS X, if *path* is a soft link to another path, then the referenced path is returned (this may be a relative path with respect to the directory owning *path*), otherwise *path* is returned (after expansion).

```
(cleanse-path path [expand-tilde?]) → path
  path : path-string?
  expand-tilde? : any/c = #f
```

Cleanses *path* (as described at the beginning of this section). The filesystem might be accessed, but the source or expanded path might be a non-existent path.

```
(expand-user-path path) → path
  path : path-string?
```

Cleanses *path*. In addition, under Unix and Mac OS X, a leading `~` is treated as user's home directory and expanded; the username follows the `~` (before a `/` or the end of the path), where `~` by itself indicates the home directory of the current user.

```
(simplify-path path [use-filesystem?]) → path?
  path : path-string?
  use-filesystem? : boolean? = #t
```

Eliminates redundant path separators (except for a single trailing separator), up-directory `..`, and same-directory `.` indicators in *path*, and changes `/` separators to `\` separators in Windows paths, such that the result accesses the same file or directory (if it exists) as *path*.

In general, the pathname is normalized as much as possible — without consulting the filesystem if *use-filesystem?* is `#f`, and (under Windows) without changing the case of letters within the path. If *path* syntactically refers to a directory, the result ends with a directory separator.

When *path* is simplified and *use-filesystem?* is true (the default), a complete path is returned; if *path* is relative, it is resolved with respect to the current directory, and up-directory indicators are removed taking into account soft links (so that the resulting path refers to the same directory as before).

When *use-filesystem?* is `#f`, up-directory indicators are removed by deleting a preceding path element, and the result can be a relative path with up-directory indicators remaining at the beginning of the path or, for Unix and Mac OS X paths; otherwise, up-directory indicators are dropped when they refer to the parent of a root directory. Similarly, the result can be the same as `(build-path 'same)` (but with a trailing separator) if eliminating up-directory indicators leaves only same-directory indicators.

The *path* argument can be a path for any platform when *use-filesystem?* is `#f`, and the resulting path is for the same platform.

The filesystem might be accessed when *use-filesystem?* is true, but the source or simplified path might be a non-existent path. If *path* cannot be simplified due to a cycle of links, the `exn:fail:filesystem` exception is raised (but a successfully simplified path may still involve a cycle of links if the cycle did not inhibit the simplification).

See §14.1.3 “Unix and Mac OS X Paths” for more information on simplifying Unix and Mac OS X paths, and see §14.1.4 “Windows Path Conventions” for more information on simplifying Windows paths.

```
(normal-case-path path) → path?
  path : path-string?
```

Returns *path* with “normalized” case letters. For Unix and Mac OS X paths, this procedure always returns the input path, because filesystems for these platforms can be case-sensitive. For Windows paths, if *path* does not start `\\?\`, the resulting string uses only lowercase letters, based on the current locale. In addition, for Windows paths when the path does not start `\\?\`, all `/`s are converted to `\`s, and trailing spaces and `.`s are removed.

The *path* argument can be a path for any platform, but beware that local-sensitive decoding and conversion of the path may be different on the current platform than for the path’s platform.

This procedure does not access the filesystem.

```
(split-path path) → (or/c path?
                          (one-of/c 'relative #f))
                    (or/c path?
                          (one-of/c 'up 'same))
                    boolean?
  path : path-string?
```

Deconstructs *path* into a smaller path and an immediate directory or file name. Three values are returned:

- `base` is either
  - a path,
  - `'relative` if *path* is an immediate relative directory or filename, or
  - `#f` if *path* is a root directory.
- `name` is either
  - a directory-name path,
  - a filename,
  - `'up` if the last part of *path* specifies the parent directory of the preceding path (e.g., `..` under Unix), or
  - `'same` if the last part of *path* specifies the same directory as the preceding path (e.g., `.` under Unix).
- `must-be-dir?` is `#t` if *path* explicitly specifies a directory (e.g., with a trailing separator), `#f` otherwise. Note that `must-be-dir?` does not specify whether `name` is actually a directory or not, but whether *path* syntactically specifies a directory.

Compared to *path*, redundant separators (if any) are removed in the result `base` and `name`. If `base` is `#f`, then `name` cannot be `'up` or `'same`. The *path* argument can be a path for any platform, and resulting paths for the same platform.

This procedure does not access the filesystem.

See §14.1.3 "Unix and Mac OS X Paths" for more information on splitting Unix and Mac OS X paths, and see §14.1.4 "Windows Path Conventions" for more information on splitting Windows paths.

```
(path-replace-suffix path suffix) → path?
  path : path-string?
  suffix : (or/c string? bytes?)
```

Returns a path that is the same as *path*, except that the suffix for the last element of the path is changed to *suffix*. If the last element of *path* has no suffix, then *suffix* is added to the path. A suffix is defined as a `.` followed by any number of non-`.` characters/bytes at the end of the path element, as long as the path element is not `".."` or `"."`. The *path* argument can be a path for any platform, and the result is for the same platform. If *path* represents a root, the `exn:fail:contract` exception is raised.

```
(path-add-suffix path suffix) → path?
  path : path-string?
  suffix : (or/c string? bytes?)
```

Similar to `path-replace-suffix`, but any existing suffix on *path* is preserved by replacing every `.` in the last path element with `_`, and then the *suffix* is added to the end.

14.1.2 More Path Utilities

```
(require scheme/path)
```

The bindings documented in this section are provided by the `scheme/path` and `scheme` libraries, but not `scheme/base`.

```
(explode-path path)
 → (listof (or/c path? (one-of/c 'up 'same)))
  path : path-string?
```

Returns the list of path element that constitute *path*. If *path* is simplified in the sense of `simple-form-path`, then the result is always a list of paths, and the first element of the list is a root.

```
(file-name-from-path path) → (or/c path? false/c)
  path : path-string?
```

Returns the last element of *path*. If *path* syntactically a directory path (see `split-path`), then then result is `#f`.

```
(filename-extension path) → (or/c bytes? false/c)
  path : path-string?
```

Returns a byte string that is the extension part of the filename in *path* without the `.` separator. If *path* is syntactically a directory (see `split-path`) or if the path has no extension, `#f` is returned.

```
(find-relative-path base path) → path?
  base : path-string?
  path : path-string?
```

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

```
(normalize-path path [wrt]) → path?
  path : path-string?
  wrt : (and/c path-string? complete-path?)
      = (current-directory)
```

Returns a normalized, complete version of *`path`*, expanding the path and resolving all soft links. If *`path`* is relative, then *`wrt`* is used as the base path.

Letter case is *not* normalized by `normalize-path`. For this and other reasons, such as whether the path is syntactically a directory, the result of `normalize-path` is not suitable for comparisons that determine whether two paths refer to the same file or directory (i.e., the comparison may produce false negatives).

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

```
(path-only path) → (or/c path? false/c)
  path : path-string?
```

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

```
(simple-form-path path) → path?
  path : path-string?
```

Returns `(simplify-path (path->complete-path path))`, which ensures that the result is a complete path containing no up- or same-directory indicators.

14.1.3 Unix and Mac OS X Paths

In Unix and Mac OS X paths, a `/` separates elements of the path, `.` as a path element always means the directory indicated by preceding path, and `..` as a path element always means the parent of the directory indicated by the preceding path. A leading `~` in a path is not treated specially, but `expand-user-path` can be used to convert a leading `~` element to a user-specific directory. No other character or byte has a special meaning within a path. Multiple adjacent `/` are equivalent to a single `/` (i.e., they act as a single path separator).

A path root is always `/`. A path starting with `/` is an absolute, complete path, and a path starting with any other character is a relative path.

Any pathname that ends with a `/` syntactically refers to a directory, as does any path whose last element is `.` or `..`.

A Unix and Mac OS X path is cleansed by replacing multiple adjacent `/`s with a single `/`.

For `(bytes->path-element bstr)`, `bstr` must not contain any `/`, otherwise the `exn:fail:contract` exception is raised. The result of `(path-element->bytes path)` or `(path-element->string path)` is always the same as the result of `(path->bytes path)` and `(path->string path)`. Since that is not the case for other platforms, however, `path-element->bytes` and `path-element->string` should be used when converting individual path elements.

Under Mac OS X, Finder aliases are zero-length files.

14.1.4 Windows Path Conventions

In general, a Windows pathname consists of an optional drive specifier and a drive-specific path. A Windows path can be *absolute* but still relative to the current drive; such paths start with a `/` or `\` separator and are not UNC paths or paths that start with `\\?\`.

A path that starts with a drive specification is *complete*. Roughly, a drive specification is either a Roman letter followed by a colon, a UNC path of the form `\\`⟨*machine*⟩`\`⟨*volume*⟩, or a `\\?\` form followed by something other than `REL\`⟨*element*⟩, or `RED\`⟨*element*⟩. (Variants of `\\?\` paths are described further below.)

Scheme fails to implement the usual Windows path syntax in one way. Outside of Scheme, a pathname `"C:rant.txt"` can be a drive-specific relative path. That is, it names a file `"rant.txt"` on drive `"C:"`, but the complete path to the file is determined by the current working directory for drive `"C:"`. Scheme does not support drive-specific working directories (only a working directory across all drives, as reflected by the `current-directory` parameter). Consequently, Scheme implicitly converts a path like `"C:rant.txt"` into `"C:\rant.txt"`.

- *Scheme-specific:* Whenever a path starts with a drive specifier ⟨*letter*⟩`:` that is not followed by a `/` or `\`, a `\` is inserted as the path is cleansed.

Otherwise, Scheme follows standard Windows path conventions, but also adds `\\?\REL` and `\\?\RED` conventions to deal with paths inexpressible in the standard conventsion, plus conventions to deal with excessive `\`s in `\\?\` paths.

In the following, ⟨*letter*⟩ stands for a Roman letter (case does not matter), ⟨*machine*⟩ stands for any sequence of characters that does not include `\` or `/` and is not `?`, ⟨*volume*⟩ stands for any sequence of characters that does not include `\` or `/` , and ⟨*element*⟩ stands for any sequence of characters that does not include `\`.

- Trailing spaces and `.` in a path element are ignored when the element is the last one in the path, unless the path starts with `\\?\` or the element consists of only spaces and `.`s.

- The following special "files", which access devices, exist in all directories, case-insensitively, and with all possible endings after a period or colon, except in pathnames that start with `\\?\`: `"NUL"`, `"CON"`, `"PRN"`, `"AUX"`, `"COM1"`, `"COM2"`, `"COM3"`, `"COM4"`, `"COM5"`, `"COM6"`, `"COM7"`, `"COM8"`, `"COM9"`, `"LPT1"`, `"LPT2"`, `"LPT3"`, `"LPT4"`, `"LPT5"`, `"LPT6"`, `"LPT7"`, `"LPT8"`, `"LPT9"`.

- Except for `\\?\` paths, `/`s are equivalent to `\`s. Except for `\\?\` paths and the start of UNC paths, multiple adjacent `/`s and `\`s count as a single `\`. In a path that starts `\\?\` paths, elements can be separated by either a single or double `\`.

- A directory can be accessed with or without a trailing separator. In the case of a non-`\\?\` path, the trailing separator can be any number of `/`s and `\`s; in the case of a `\\?\` path, a trailing separator must be a single `\`, except that two `\`s can follow `\\?\`⟨*letter*⟩`:`.

- Except for `\\?\` paths, a single `.` as a path element means "the current directory," and a `..` as a path element means "the parent directory." Up-directory path elements (i.e., `..`) immediately after a drive are ignored.

- A pathname that starts `\\`⟨*machine*⟩`\`⟨*volume*⟩ (where a `/` can replace any `\`) is a UNC path, and the starting `\\`⟨*machine*⟩`\`⟨*volume*⟩ counts as the drive specifier.

- Normally, a path element cannot contain any of the following characters:

`< > : " / \ |`

Except for `\`, path elements containing these characters can be accessed using a `\\?\` path (assuming that the underlying filesystem allows the characters).

- In a pathname that starts `\\?\`⟨*letter*⟩`:\`, the `\\?\`⟨*letter*⟩`:\` prefix counts as the path's drive, as long as the path does not both contain non-drive elements and end with two consecutive `\`s, and as long as the path contains no sequence of three or more `\`s. Two `\`s can appear in place of the `\` before ⟨*letter*⟩. A `/` cannot be used in place of a `\` (but `/`s can be used in element names, though the result typically does not name an actual directory or file).

- In a pathname that starts `\\?\UNC\`⟨*machine*⟩`\`⟨*volume*⟩, the

`\\?\UNC\`⟨*machine*⟩`\`⟨*volume*⟩ prefix counts as the path's drive, as long as the path does not end with two consecutive `\`s, and as long as the path contains no sequence of three or more `\`s. Two `\`s can appear in place of the `\` before `UNC`, the `\`s after `UNC`, and/or the `\`s after⟨*machine*⟩. The letters in the `UNC` part can be uppercase or lowercase, and `/` cannot be used in place of `\`s (but `/` can be used in element names).

- *Scheme-specific:* A pathname that starts `\\?\REL\`⟨*element*⟩ or `\\?\REL\\`⟨*element*⟩ is a relative path, as long as the path does not end with two consecutive `\`s, and as long as the path contains no sequence of three or more `\`s. This Scheme-specific path form supports relative paths with elements that are not normally expressible in Windows paths (e.g., a final element that ends in a space). The `REL` part must be exactly the three uppercase letters, and `/`s cannot be used in place of `\`s. If the path starts `\\?\REL\..` then for as long as the path continues with repetitions of `\..`, each element counts as an up-directory element; a single `\` must be used to separate the up-directory elements. As soon as a second `\` is used to separate the elements, or as soon as a non-`..` element is encountered, the remaining elements are all literals (never up-directory elements). When a `\\?\REL` path value is converted to a string (or when the path value is written or displayed), the string does not contain the starting `\\?\REL` or the immediately following `\`s; converting a path value to a byte string preserves the `\\?\REL` prefix.
- *Scheme-specific:* A pathname that starts `\\?\RED\`⟨*element*⟩ or `\\?\RED\\`⟨*element*⟩ is a drive-relative path, as long as the path does not end with two consecutive `\`s, and as long as the path contains no sequence of three or more `\`s. This Scheme-specific path form supports drive-relative paths (i.e., absolute given a drive) with elements that are not normally expressible in Windows paths. The `RED` part must be exactly the three uppercase letters, and `/`s cannot be used in place of `\`s. Unlike `\\?\REL` paths, a `..` element is always a literal path element. When a `\\?\RED` path value is converted to a string (or when the path value is written or displayed), the string does not contain the starting `\\?\RED` and it contains a single starting `\`; converting a path value to a byte string preserves the `\\?\RED` prefix.

Three additional Scheme-specific rules provide meanings to character sequences that are otherwise ill-formed as Windows paths:

- *Scheme-specific:* In a pathname of the form `\\?\`⟨*any*⟩`\\` where ⟨*any*⟩ is any non-empty sequence of characters other than ⟨*letter*⟩`:` or `\`⟨*letter*⟩`:`, the entire path counts as the path's (non-existent) drive.
- *Scheme-specific:* In a pathname of the form `\\?\`⟨*any*⟩`\\\`⟨*elements*⟩, where ⟨*any*⟩ is any non-empty sequence of characters and ⟨*elements*⟩ is any sequence that does not start with a `\`, does not end with two `\`s, and does not contain a sequence of three `\`s, then `\\?\`⟨*any*⟩`\\` counts as the path's (non-existent) drive.
- *Scheme-specific:* In a pathname that starts `\\?\` and does not match any of the patterns from the preceding bullets, `\\?\` counts as the path's (non-existent) drive.

Outside of Scheme, except for `\\?\` paths, pathnames are typically limited to 259 characters. Scheme internally converts pathnames to `\\?\` form as needed to avoid this limit. The operating system cannot access files through `\\?\` paths that are longer than 32,000 characters or so.

Where the above descriptions says "character," substitute "byte" for interpreting byte strings as paths. The encoding of Windows paths into bytes preserves ASCII characters, and all special characters mentioned above are ASCII, so all of the rules are the same.

Beware that the `\` path separator is an escape character in Scheme strings. Thus, the path `\\?\REL\..\\..` as a string must be written `"\\\\?\\REL\\..\\\\.."`.

A path that ends with a directory separator syntactically refers to a directory. In addition, a path syntactcially refers to a directory if its last element is a same-directory or up-directory indicator (not quoted by a `\\?\` form), or if it refers to a root.

Windows paths are cleansed as follows: In paths that start `\\?\`, redundant `\`s are removed, an extra `\` is added in a `\\?\REL` if an extra one is not already present to separate up-directory indicators from literal path elements, and an extra `\` is similarly added after `\\?\RED` if an extra one is not already present. When `\\?\` acts as the root and the path contains, to additional `/`s (which might otherwise be redundant) are included after the root. For other paths, multiple `/`s are converted to single `/`s (except at the beginning of a shared folder name), a `/` is inserted after the colon in a drive specification if it is missing.

For `(bytes->path-element bstr)`, `/`s, colons, trailing dots, trailing whitespace, and special device names (e.g., "aux") in *bstr* are encoded as a literal part of the path element by using a `\\?\REL` prefix. The `bstr` argument must not contain a `\`, otherwise the `exn:fail:contract` exception is raised.

For `(path-element->bytes path)` or `(path-element->string path)`, if the byte-string form of *path* starts with a `\\?\REL`, the prefix is not included in the result.

For `(build-path base-path sub-path ...)`, trailing spaces and periods are removed from the last element of *base-path* and all but the last *sub-path* (unless the element consists of only spaces and peroids), except for those that start with `\\?\`. If *base-path* starts `\\?\`, then after each non-`\\?\REL\` and non-`\\?\RED\` *sub-path* is added, all `/`s in the addition are converted to `\`s, multiple consecutive `\`s are converted to a single `\`, added `.` elements are removed, and added `..` elements are removed along with the preceding element; these conversions are not performed on the original *base-path* part of the result or on any `\\?\REL\` or `\\?\RED\` or *sub-path*. If a `\\?\REL\` or `\\?\RED\` *sub-path* is added to a non-`\\?\` *base-path*, the the *base-path* (with any additions up to the `\\?\REL\` or `\\?\RED\` *sub-path*) is simplified and converted to a `\\?\` path. In other cases, a `\` may be added or removed before combining paths to avoid changing the root meaning of the path (e.g., combining `//x` and `y` produces `/x/y`, because `//x/y` would be a UNC path instead of a drive-relative path).

For `(simplify-path path use-filesystem?)`, *path* is expanded, and if *path* does not start with `\\?\`, trailing spaces and periods are removed, a `/` is inserted after the colon in a drive specification if it is missing, and a `\` is inserted after `\\?\` as a root if there are elements and no extra `\` already. Otherwise, if no indicators or redundant separators are in *path*, then *path* is returned.

For `(split-path path)` producing *base*, *name*, and *must-be-dir?*, splitting a path that does not start with `\\?\` can produce parts that start with `\\?\`. For example, splitting `C:/x~/aux/` produces `\\?\C:\x~\` and `\\?\REL\\aux`; the `\\?\` is needed in these cases to preserve a trailing space after `x` and to avoid referring to the AUX device instead of an `"aux"` file.

14.2 Filesystem

14.2.1 Locating Paths

```
(find-system-path kind) → path?
  kind : symbol?
```

Returns a machine-specific path for a standard type of path specified by *kind*, which must be one of the following:

- `'home-dir` — the current user's home directory.

  Under Unix and Mac OS X, this directory is determined by expanding the path `"~"`, which is expanded by first checking for a `HOME` environment variable. If none is defined, the `USER` and `LOGNAME` environment variables are consulted (in that order) to find a user name, and then system files are consulted to locate the user's home directory.

  Under Windows, the user's home directory is the user-specific profile directory as determined by the Windows registry. If the registry cannot provide a directory for some reason, the value of the `USERPROFILE` environment variable is used instead, as long as it refers to a directory that exists. If `USERPROFILE` also fails, the directory is the one specified by the `HOMEDRIVE` and `HOMEPATH` environment variables. If those environment variables are not defined, or if the indicated directory still does not exist, the directory containing the current executable is used as the home directory.

- `'pref-dir` — the standard directory for storing the current user's preferences. Under Unix, the directory is `".plt-scheme"` in the user's home directory. Under Windows, it is `"PLT Scheme"` in the user's application-data folder as specified by the Windows registry; the application-data folder is usually `"Application Data"` in the user's profile directory. Under Mac OS X, it is `"Library/Preferences"` in the user's home directory. This directory might not exist.

- 'pref-file — a file that contains a symbol-keyed association list of preference values. The file's directory path always matches the result returned for 'pref-dir. The file name is "plt-prefs.ss" under Unix and Windows, and it is "org.plt-scheme.prefs.ss" under Mac OS X. The file's directory might not exist. See also get-preference.
- 'temp-dir — the standard directory for storing temporary files. Under Unix and Mac OS X, this is the directory specified by the TMPDIR environment variable, if it is defined, otherwise it is the first path that exists among "/var/tmp", "/usr/tmp", and "/tmp". Under Windows, the result is the directory specified by the TMP or TEMP environment variable, if it is defined, otherwise it is the current directory.
- 'init-dir — the directory containing the initialization file used by stand-alone mzscheme executable. It is the same as the current user's home directory.
- 'init-file — the file loaded at start-up by the stand-alone mzscheme executable. The directory part of the path is the same path as returned for 'init-dir. The file name is platform-specific:
  - Unix and Mac OS X: ".mzschemerc"
  - Windows: "mzschemerc.ss"
- 'addon-dir — a directory for installing PLT Scheme extensions. It's the same as 'pref-dir, except under Mac OS X, where it is "Library/PLT Scheme" in the user's home directory. This directory might not exist.
- 'doc-dir — the standard directory for storing the current user's documents. Under Unix, it's the same as 'home-dir. Under Mac OS X, it's the "Documents" directory in the user's home directory. Under Windows, it is the user's documents folder as specified by the Windows registry; the documents folder is usually "My Documents" in the user's home directory.
- 'desk-dir — the directory for the current user's desktop. Under Unix, it's the same as 'home-dir. Under Windows, it is the user's desktop folder as specified by the Windows registry; the documents folder is usually "Desktop" in the user's home directory. Under Mac OS X, it is the desktop directory, which is specifically "~/Desktop" under Mac OS X.
- 'sys-dir — the directory containing the operating system for Windows. Under Unix and Mac OS X, the result is "/".
- 'exec-file — the path of the mzscheme executable as provided by the operating system for the current invocation.

For MrEd, the executable path is the name of a MrEd executable.

- 'run-file — the path of the current executable; this may be different from result for 'exec-file because an alternate path was provided through a --name or -N command-line flag to the mzscheme (or mred) executable, or because an embedding executable installed an alternate path. In particular a "launcher" script created by make-mzscheme-launcher sets this path to the script's path. In the mzscheme executable, this path is also bound initially to program.

- `'collects-dir` — a path to the main collection of libraries (see §16.2 "Libraries and Collections"). If this path is relative, it's relative to the directory of `(find-system-path 'exec-file)`. This path is normally embedded in the `mzscheme` executable, but it can be overridden by the `--collects` or `-X` command-line flag.

- `'orig-dir` — the current directory at start-up, which can be useful in converting a relative-path result from `(find-system-path 'exec-file)` or `(find-system-path 'run-file)` to a complete path.

```
(path-list-string->path-list str
                             default-path-list)
 → (listof path?)
  str : (or/c string? bytes?)
  default-path-list : (listof path?)
```

Parses a string or byte string containing a list of paths, and returns a list of path strings. Under Unix and Mac OS X, paths in a path list are separated by a `:`; under Windows, paths are separated by a `;`. Whenever the path list contains an empty path, the list *`default-path-list`* is spliced into the returned list of paths. Parts of *`str`* that do not form a valid path are not included in the returned list.

```
(find-executable-path  program-sub
                       related-sub
                      [deepest?])   → (or/c path? false/c)
  program-sub : path-string?
  related-sub : path-string?
  deepest? : any/c = #f
```

Finds a path for the executable *`program-sub`*, returning `#f` if the path cannot be found.

If *`related-sub`* is not `#f`, then it must be a relative path string, and the path found for *`program-sub`* must be such that the file or directory *`related-sub`* exists in the same directory as the executable. The result is then the full path for the found *`related-sub`*, instead of the path for the executable.

This procedure is used by the `mzscheme` executable to find the standard library collection directory (see §16.2 "Libraries and Collections"). In this case, `program` is the name used to start MzScheme and `related` is `"collects"`. The *`related-sub`* argument is used because, under Unix and Mac OS X, *`program-sub`* may involve to a sequence of soft links; in this case, *`related-sub`* determines which link in the chain is relevant.

If *`related-sub`* is not `#f`, then when `find-executable-path` does not finds a *`program-sub`* that is a link to another file path, the search can continue with the destination of the link. Further links are inspected until *`related-sub`* is found or the end of the chain of links is reached. If *deepest?* is #f (the default), then the result corresponds to the first path in a chain of links for which *related-sub* is found (and further links are not actually explored); otherwise, the result corresponds to the last link in the chain for which *related-sub* is found.

If *program-sub* is a pathless name, find-executable-path gets the value of the PATH environment variable; if this environment variable is defined, find-executable-path tries each path in PATH as a prefix for *program-sub* using the search algorithm described above for path-containing *program-sub*s. If the PATH environment variable is not defined, *program-sub* 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.)

14.2.2 Files

```
(file-exists? path) → boolean?
  path : path-string?
```

Returns #t if a file (not a directory) *path* exists, #f otherwise.

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

```
(link-exists? path) → boolean?
  path : path-string?
```

Returns #t if a link *path* exists (Unix and Mac OS X), #f otherwise.

The predicates file-exists? or directory-exists? work on the final destination of a link or series of links, while link-exists? only follows links to resolve the base part of *path* (i.e., everything except the last name in the path).

This procedure never raises the exn:fail:filesystem exception.

```
(delete-file path) → void?
  path : path-string?
```

Feletes the file with path *path* if it exists, otherwise the exn:fail:filesystem exception is raised. If *path* is a link, the link is deleted rather than the destination of the link.

```
(rename-file-or-directory old
                          new
                          [exists-ok?]) → void?
  old : path-string?
  new : path-string?
  exists-ok? : any/c = #f
```

Renames the file or directory with path *old*—if it exists—to the path *new*. If the file or directory is not renamed successfully, the `exn:fail:filesystem` exception is raised.

This procedure can be used to move a file/directory to a different directory (on the same disk) as well as rename a file/directory within a directory. Unless *exists-ok?* is provided as a true value, *new* cannot refer to an existing file or directory. Even if *exists-ok?* is true, *new* cannot refer to an existing file when *old* is a directory, and vice versa. (If *new* exists and is replaced, the replacement is atomic in the filesystem, except under Windows 95, 98, or Me. However, the check for existence is not included in the atomic action, which means that race conditions are possible when *exists-ok?* is false or not supplied.)

If *old* is a link, the link is renamed rather than the destination of the link, and it counts as a file for replacing any existing *new*.

```
(file-or-directory-modify-seconds path
                                  [secs-n
                                   fail-thunk]) → any
  path : path-string?
  secs-n : (or/c exact-integer? false/c) = #f
  fail-thunk : (-> any)
             = (lambda () (raise (make-exn:fail:filesystem ....)))
```

Returns the file or directory's last modification date as platform-specific seconds (see also §14.5 "Time") when *secs-n* is not provided or is `#f`. (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).)

If *secs-n* is provided and not `#f`, the access and modification times of *path* are set to the given time.

On error (e.g., if no such file exists), *fail-thunk* is called, and the default *fail-thunk* raises `exn:fail:filesystem`.

```
(file-or-directory-permissions path) → (listof symbol?)
  path : path-string?
```

Returns a list containing `'read`, `'write`, and/or `'execute` for the given file or directory path. On error (e.g., if no such file exists), the `exn:fail:filesystem` exception is raised.

Under Unix and Mac OS X, permissions are checked for the current effective user instead of the real user.

```
(file-size path) → nonnegative-exact-integer?
  path : path-string?
```

Returns the (logical) size of the specified file in bytes. Under Mac OS X, this size excludes the resource-fork size. On error (e.g., if no such file exists), the `exn:fail:filesystem` exception is raised.

```
(copy-file src dest) → void?
  src : path-string?
  dest : path-string?
```

Creates the file *dest* as a copy of *src*. If the file is not successfully copied, the `exn:fail:filesystem` exception is raised. If *dest* already exists, the copy will fail. File permissions are preserved in the copy. Under Mac OS X, the resource fork is also preserved in the copy. If *src* refers to a link, the target of the link is copied, rather than the link itself.

```
(make-file-or-directory-link to path) → void?
  to : path-string?
  path : path-string?
```

Creates a link *path* to *to* under Unix and Mac OS X. The creation will fail if *path* already exists. The *to* need not refer to an existing file or directory, and *to* is not expanded before writing the link. If the link is not created successfully,the `exn:fail:filesystem` exception is raised. Under Windows, the `exn:fail:unsupported` exception is raised always.

14.2.3 Directories

See also: `rename-file-or-directory`, `file-or-directory-modify-seconds`, `file-or-directory-permissions`.

```
(current-directory) → path-string?
(current-directory path) → void?
  path : path-string?
```

A parameter that determines the current directory for resolving relative paths.

When the parameter procedure is called to set the current directory, the path argument is cleansed using `cleanse-path`, simplified using `simplify-path`, and then converted to a directory path with `path->directory-path`; cleansing and simplification raise an exception if the path is ill-formed. Thus, the current value of `current-directory` is always a cleansed, simplified, complete, directory path.

The path is not checked for existence when the parameter is set.

```
(current-drive) → path?
```

Returns the current drive name Windows. For other platforms, the `exn:fail:unsupported` exception is raised. The current drive is always the drive of the current directory.

```
(directory-exists? path) → boolean?
  path : path-string?
```

Returns `#t` if *path* refers to a directory, `#f` otherwise.

```
(make-directory path) → void?
  path : path-string?
```

Creates a new directory with the path *path*. If the directory is not created successfully, the `exn:fail:filesystem` exception is raised.

```
(delete-directory path) → void?
  path : path-string?
```

Deletes an existing directory with the path *path*. If the directory is not deleted successfully, the `exn:fail:filesystem` exception is raised.

```
(directory-list [path]) → (listof path?)
  path : path-string? = (current-directory)
```

Returns a list of all files and directories in the directory specified by *path*. If *path* is omitted, a list of files and directories in the current directory is returned. Under Unix and Mac OS X, an element of the list can start with `./~` if it would otherwise start with `~`. Under Windows, an element of the list may start with `\\?\REL\\`.

```
(filesystem-root-list) → (listof path?)
```

Returns a list of all current root directories. Obtaining this list can be particularly slow under Windows.

14.2.4 Declaring Paths Needed at Run Time

```
(require scheme/runtime-path)
```

The bindings documented in this section are provided by the `scheme/runtime-path` library, not `scheme/base` or `scheme`.

The `scheme/runtime-path` library provides forms for accessing files and directories at run time using a path that are usually relative to an enclosing source file. Unlike using `collection-path`, `define-runtime-path` exposes each run-time path to tools like the executable and distribution creators, so that files and directories needed at run time are carried along in a distribution.

```
(define-runtime-path id expr)
```

Uses *expr* as both a compile-time (i.e., phase 1) expression and a run-time (i.e., phase 0) expression. In either context, *expr* should produce a path, a string that represents a path, a list of the form `(list 'lib str ...+)`, or a list of the form `(list 'so str)`.

For run time, *id* is bound to a path that is based on the result of *expr*. The path is normally computed by taking a relative path result from *expr* and adding it to a path for the enclosing file (which is computed as described below). However, tools like the executable creator can also arrange (by colluding with `scheme/runtime-path`) to have a different base path substituted in a generated executable. If *expr* produces an absolute path, it is normally returned directly, but again may be replaced by an executable creator. In all cases, the executable creator preserves the relative locations of all paths. When *expr* produces a relative or absolute path, then the path bound to *id* is always an absolute path.

If *expr* produces a list of the form `(list 'lib str ...+)`, the value bound to *id* is an absolute path. The path refers to a collection-based file similar to using the value as a module path.

If *expr* produces a list of the form `(list 'so str)`, the value bound to *id* can be either *str* or an absolute path; it is an absolute path when adding the platform-specific shared-library extension — as produced by `(system-type 'so-suffix)` — and then searching in the PLT-specific shared-object library directories (as determined by `find-dll-dirs`) locates the path. In this way, shared-object libraries that are installed specifically for PLT Scheme get carried along in distributions.

For compile-time, the *expr* result is used by an executable creator—but not the result when the containing module is compiled. Instead, *expr* is preserved in the module as a compile-time expression (in the sense of `begin-for-syntax`). Later, at the time that an executable is created, the compile-time portion of the module is executed (again), and the result of *expr* is the file to be included with the executable. The reason for the extra compile-time execution is that the result of *expr* might be platform-dependent, so the result should not be stored in the (platform-independent) bytecode form of the module; the platform at executable-creation time, however, is the same as at run time for the executable. Note that *expr* is still evaluated at run-time; consequently, avoid procedures like `collection-path`, which depends on the source installation, and instead use relative paths and forms like `(list 'lib str ...+)`.

If a path is needed only on some platforms and not on others, use `define-runtime-path-list` with an *expr* that produces an empty list on platforms where the path is not needed.

The enclosing path for a `define-runtime-path` is determined as follows from the `define-runtime-path` syntactic form:

- If the form has a source module according to `syntax-source-module`, then the source location is determined by preserving the original expression as a syntax object, extracting its source module path at run time (again using `syntax-source-module`), and then resolving the resulting module path index.
- If the expression has no source module, the `syntax-source` location associated with the form is used, if is a string or path.
- If no source module is available, and `syntax-source` produces no path, then `current-load-relative-directory` is used if it is not `#f`. Finally, `current-directory` is used if all else fails.

In the latter two cases, the path is normally preserved in (platform-specific) byte form. If it is is within the result of `find-collects-dir`, however, it the path is recorded relative to `(find-collects-dir)`, and it is reconstructed using `(find-collects-dir)` at run time.

Examples:

```
; Access a file "data.txt" at run-time that is originally
; located in the same directory as the module source file:
(define-runtime-path data-file "data.txt")
(define (read-data)
  (with-input-from-file data-file
    (lambda ()
      (read-bytes (file-size data-file)))))

; Load a platform-specific shared object (using ffi-lib)
; that is located in a platform-specific sub-directory of the
; module's source directory:
(define-runtime-path libfit-path
  (build-path "compiled" "native" (system-library-subpath #f)
              (path-replace-suffix "libfit"
                                   (system-type 'so-suffix))))
(define libfit (ffi-lib libfit-path))

; Load a platform-specific shared object that might be installed
```

```
; as part of the operating system, or might be installed
; specifically for PLT Scheme:
(define-runtime-path libssl-so
  (case (system-type)
    [(windows) '(so "ssleay32")]
    [else '(so "libssl")]))
(define libssl (ffi-lib libssl-so))
```

`(define-runtime-paths (id ...) expr)`

Like `define-runtime-path`, but declares and binds multiple paths at once. The *expr* should produce as many values as *id*s.

`(define-runtime-path-list id expr)`

Like `define-runtime-path`, but *expr* should produce a list of paths.

`(runtime-paths module-path)`

This form is mainly for use by tools such as executable builders. It expands to a quoted list containing the run-time paths declared by *module-path*, returning the compile-time results of the declaration `expr`s, except that paths are converted to byte strings. The enclosing module must require (directly or indirectly) the module specified by *module-path*, which is an unquoted module path.

14.2.5 More File and Directory Utilities

`(require scheme/file)`

The bindings documented in this section are provided by the `scheme/file` and `scheme` libraries, but not `scheme/base`.

```
(copy-directory/files src dest) → void?
  src : path-string?
  dest : path-string?
```

Copies the file or directory *src* to *dest*, raising `exn:fail:filesystem` if the file or directory cannot be copied, possibly because *dest* exists already. If *src* is a directory, the copy applies recursively to the directory's content. If a source is a link, the target of the link is copied rather than the link itself.

```
(delete-directory/files path) → void?
  path : path-string?
```

Deletes the file or directory specified by *path*, raising `exn:fail:filesystem` if the file or directory cannot be deleted. If *path* is a directory, then `delete-directory/files` is first applied to each file and directory in *path* before the directory is deleted.

```
(find-files predicate [start-path]) → (listof path?)
  predicate : (path? . -> . any/c)
  start-path : (or/c path-string? false/c) = #f
```

Traverses the filesystem starting at *start-path* and creates a list of all files and directories for which *predicate* returns true. If *start-path* is `#f`, then the traversal starts from `(current-directory)`. In the resulting list, each directory precedes its content.

The *predicate* procedure is called with a single argument for each file or directory. If *start-path* is `#f`, the argument is a pathname string that is relative to the current directory. Otherwise, it is a path building on *start-path*. Consequently, supplying `(current-directory)` for *start-path* is different from supplying `#f`, because *predicate* receives complete paths in the former case and relative paths in the latter. Another difference is that *predicate* is not called for the current directory when *start-path* is `#f`.

The `find-files` traversal follows soft links. To avoid following links, use the more general `fold-files` procedure.

If *start-path* does not refer to an existing file or directory, then *predicate* will be called exactly once with *start-path* as the argument.

```
(pathlist-closure path-list) → (listof path?)
  path-list : (listof path-string?)
```

Given a list of paths, either absolute or relative to the current directory, returns a list such that

- if a nested path is given, all of its ancestors are also included in the result (but the same ancestor is not added twice);
- if a path refers to directory, all of its descendants are also included in the result;
- ancestor directories appear before their descendants in the result list.

```
(fold-files proc
            init-val
           [start-path
            follow-links?]) → any
  proc : (and/c (path? (one-of/c 'file 'dir 'link) any/c
                . -> . any/c)
               (or/c procedure?
                     ((path? (one-of/c 'dir) any/c)
                      . ->* . (any/c any/c))))
  init-val : any/c
  start-path : (or/c path-string? false/c) = #f
  follow-links? : any/c = #t
```

Traverses the filesystem starting at *start-path*, calling *proc* on each discovered file, directory, and link. If *start-path* is `#f`, then the traversal starts from `(current-directory)`.

The *proc* procedure is called with three arguments for each file, directory, or link:

- If *start-path* is `#f`, the first argument is a pathname string that is relative to the current directory. Otherwise, the first argument is a pathname that starts with *start-path*. Consequently, supplying `(current-directory)` for *start-path* is different from supplying `#f`, because *proc* receives complete paths in the former case and relative paths in the latter. Another difference is that *proc* is not called for the current directory when *start-path* is `#f`.
- The second argument is a symbol, either `'file`, `'dir`, or `'link`. The second argument can be `'link` when *follow-links?* is `#f`, in which case the filesystem traversal does not follow links. If *follow-links?* is `#t`, then *proc* will only get a `'link` as a second argument when it encounters a dangling symbolic link (one that does not resolve to an existing file or directory).
- The third argument is the accumulated result. For the first call to *proc*, the third argument is *init-val*. For the second call to *proc* (if any), the third argument is the result from the first call, and so on. The result of the last call to *proc* is the result of `fold-files`.

The *proc* argument is used in an analogous way to the procedure argument of `foldl`, where its result is used as the new accumulated result. There is an exception for the case of a directory (when the second argument is `'dir`): in this case the procedure may return two values, the second indicating whether the recursive scan should include the given directory or not. If it returns a single value, the directory is scanned.

An error is signaled if the *start-path* is provided but no such path exists, or if paths disappear during the scan.

```
(make-directory* path) → void?
  path : path-string?
```

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

```
(make-temporary-file [template
                      copy-from-filename
                      directory])        → path?
  template : string? = "mztmp~a"
  copy-from-filename : (or/c path-string? false/c (one-of/c 'directory))
                     = #f
  directory : (or/c path-string? false/c) = #f
```

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.

The `template` argument must be a format string suitable for use with `format` and one additional string argument (where the string contains only digits). If the resulting string is a relative path, it is combined with the result of `(find-system-path 'temp-dir)`, unless `directory` is provided and non-`#f`, in which case the file name generated from `template` is combined with `directory` to obtain a full path.

If `copy-from-filename` is provided as path, the temporary file is created as a copy of the named file (using `copy-file`). If `copy-from-filename` is `#f`, the temporary file is created as empty. If `copy-from-filename` is `'directory`, then the temporary "file" is created as a directory.

When a temporary file is created, it 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 `open-output-file`) and to delete it when it is no longer needed.

```
(get-preference  name
                [failure-thunk
                 flush-mode
                 filename])    → any
  name : symbol?
  failure-thunk : (-> any) = (lambda () #f)
  flush-mode : any/c = 'timestamp
  filename : (or/c string-path? false/c) = #f
```

Extracts a preference value from the file designated by `(find-system-path 'pref-file)`, or by *filename* if it is provided and is not `#f`. In the former case, if the preference file doesn't exist, `get-preferences` attempts to read a `"plt-prefs.ss"` file in the "defaults" collection, instead. If neither file exists, the preference set is empty.

The preference file should contain a symbol-keyed association list (written to the file with the default parameter settings). Keys starting with `mzscheme:`, `mred:`, and `plt:` in any letter case are reserved for use by PLT.

The result of `get-preference` is the value associated with *name* if it exists in the association list, or the result of calling *failure-thunk* otherwise.

Preference settings are cached (weakly) across calls to `get-preference`, using `(path->complete-path filename)` as a cache key. If *flush-mode* is provided as `#f`, the cache is used instead of the re-consulting the preferences file. If *flush-mode* is provided as `'timestamp` (the default), then the cache is used only if the file has a timestamp that is the same as the last time the file was read. Otherwise, the file is re-consulted.

See also `put-preferences`. For a more elaborate preference system, see `preferences:get`.

```
(put-preferences  names
                  vals
                 [locked-proc
                  filename])  → void?
  names : (listof symbol?)
  vals : list?
  locked-proc : (path? . -> . any) = (lambda (p) (error ....))
  filename : (or/c false/c path-string?) = #f
```

Installs a set of preference values and writes all current values to the preference file designated by `(find-system-path 'pref-file)`, or *filename* if it is supplied and not `#f`.

The *names* argument supplies the preference names, and *vals* must have the same length as *names*. Each element of *vals* must be an instance of a built-in data type whose `write` output is `read`able (i.e., the `print-unreadable` parameter is set to `#f` while writing preferences).

Current preference values are read from the preference file before updating, and an update "lock" is held starting before the file read, and lasting until after the preferences file is updated. The lock is implemented by the existence of a file in the same directory as the preference file. If the directory of the preferences file does not already exist, it is created.

If the update lock is already held (i.e., the lock file exists), then `locked` is called with a single argument: the path of the lock file. The default `locked` reports an error; an alternative thunk might wait a while and try again, or give the user the choice to delete the lock file (in case a previous update attempt encountered disaster).

If *filename* is `#f` or not supplied, and the preference file does not already exist, then values read from the "defaults" collection (if any) are written for preferences that are not mentioned in *names*.

14.3 Networking

14.3.1 TCP

```
(require scheme/tcp)
```

The bindings documented in this section are provided by the `scheme/tcp` and `scheme` libraries, but not `scheme/base`.

For information about TCP in general, see *TCP/IP Illustrated, Volume 1* by W. Richard Stevens.

```
(tcp-listen  port-no
            [max-allow-wait
             reuse?
             hostname])      → tcp-listener?
  port-no : (and/c nonnegative-exact-integer?
                   (integer-in 1 65535))
  max-allow-wait : nonnegative-exact-integer? = 4
  reuse? : any/c = #f
  hostname : (or/c string? false/c) = #f
```

Creates a "listening" server on the local machine at the port number specified by `port-no`. The `max-allow-wait` argument determines the maximum number of client connections that can be waiting for acceptance. (When `max-allow-wait` clients are waiting acceptance, no new client connections can be made.)

If the `reuse?` argument is true, then `tcp-listen` will create a listener even if the port is involved in a `TIME_WAIT` state. Such a use of `reuse?` defeats certain guarantees of the TCP protocol; see Stevens's book for details. Furthermore, on many modern platforms, a true value for `reuse?` overrides `TIME_WAIT` only if the listener was previously created with a true value for `reuse?`.

If `hostname` is `#f` (the default), then the listener accepts connections to all of the listening machine's addresses. Otherwise, the listener accepts connections only at the interface(s) associated with the given hostname. For example, providing `"127.0.0.1"` as `hostname` creates a listener that accepts only connections to `"127.0.0.1"` (the loopback interface) from the local machine.

(Scheme implements a listener with multiple sockets, if necessary, to accomodate multiple addresses with different protocol families. Under Linux, if `hostname` maps to both IPv4 and

IPv6 addresses, then the behavior depends on whether IPv6 is supported and IPv6 sockets can be configured to listen to only IPv6 connections: if IPv6 is not supported or IPv6 sockets are not configurable, then the IPv6 addresses are ignored; otherwise, each IPv6 listener accepts only IPv6 connections.)

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 §13.6 "Custodians").

If the server cannot be started by `tcp-listen`, the `exn:fail:network` exception is raised.}

```
(tcp-connect  hostname
              port-no
             [local-hostname
              local-port-no]) → input-port? output-port?
  hostname : string?
  port-no : (and/c nonnegative-exact-integer?
                   (integer-in 1 65535))
  local-hostname : (or/c string? false/c) = #f
  local-port-no : (or/c (and/c nonnegative-exact-integer? = #f
                               (integer-in 1 65535))
                        false/c)
```

Attempts to connect as a client to a listening server. The *hostname* argument is the server host's Internet address name, and *port-no* is the port number where the server is listening.

(If *hostname* is associated with multiple addresses, they are tried one at a time until a connection succeeds. The name `"localhost"` generally specifies the local machine.)

The optional *local-hostname* and *local-port-no* specify the client's address and port. If both are `#f` (the default), the client's address and port are selected automatically. If *local-hostname* is not `#f`, then *local-port-no* must be non-`#f`. If *local-port-no* is non-`#f` and *local-hostname* is `#f`, then the given port is used but the address is selected automatically.

Two values 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 §13.6 "Custodians").

Initially, the returned input port is block-buffered, and the returned output port is block-buffered. Change the buffer mode using `file-stream-buffer-mode`.

Both of the returned ports must be closed to terminate the TCP connection. When both ports are still open, closing the output port with `close-output-port` sends a TCP close to the server (which is seen as an end-of-file if the server reads the connection through a port). In contrast, `tcp-abandon-port` (see below) closes the output port, but does not send a TCP close until the input port is also closed.

Note that the TCP protocol does not support a state where one end is willing to send but not read, nor does it include an automatic message when one end of a connection is fully closed. Instead, the other end of a connection discovers that one end is fully closed only as a response to sending data; in particular, some number of writes on the still-open end may appear to succeed, though writes will eventually produce an error.

If a connection cannot be established by `tcp-connect`, the `exn:fail:network` exception is raised.

```
(tcp-connect/enable-break  hostname
                           port-no
                          [local-hostname]
                           local-port-no)
 → input-port? output-port?
  hostname : string?
  port-no : (and/c nonnegative-exact-integer?
                  (integer-in 1 65535))
  local-hostname : (or/c string? false/c) = #f
  local-port-no : (or/c (and/c nonnegative-exact-integer?
                               (integer-in 1 65535))
                        false/c)
```

Like `tcp-connect`, but breaking is enabled (see §9.6 "Breaks") while trying to connect. If breaking is disabled when `tcp-connect/enable-break` is called, then either ports are returned or the `exn:break` exception is raised, but not both.

```
(tcp-accept listener) → input-port? output-port?
  listener : tcp-listener?
```

Accepts a client connection for the server associated with *listener*, which 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?`.)

Two values 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 §13.6 "Custodians").

In terms of buffering and connection states, the ports act the same as ports from `tcp-connect`.

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

```
(tcp-accept/enable-break listener) → input-port? output-port?
  listener : tcp-listener?
```

Like `tcp-accept`, but breaking is enabled (see §9.6 "Breaks") while trying to accept a connection. If breaking is disabled when `tcp-accept/enable-break` is called, then either ports are returned or the `exn:break` exception is raised, but not both.

```
(tcp-accept-ready? listener) → boolean?
  listener : tcp-listener?
```

Tests whether an unaccepted client has connected to the server associated with *listener*. The *listener* argument is a TCP listener value returned by `tcp-listen`. If a client is waiting, the return value is `#t`, otherwise it is `#f`. A client is accepted with the `tcp-accept` procedure, which returns ports for communicating with the client and removes the client from the list of unaccepted clients.

If the listener has been closed, the `exn:fail:network` exception is raised.

```
(tcp-close listener) → void?
  listener : tcp-listener?
```

Shuts down the server associated with *listener*. The *listener* argument is a TCP listener value returned by `tcp-listen`. All unaccepted clients receive an end-of-file from the server; connections to accepted clients are unaffected.

If the listener has already been closed, the `exn:fail:network` exception is raised.

The listener's port number may not become immediately available for new listeners (with the default `reuse?` argument of `tcp-listen`). For further information, see Stevens's explanation of the `TIME\_WAIT` TCP state.

```
(tcp-listener? v) → boolean?
  v : any/c
```

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

```
(tcp-accept-evt listener) → evt?
  listener : tcp-listener?
```

Returns a synchronizable event (see §10.2.1 "Events") that is in a blocking state when `tcp-accept` on *listener* would block. If the event is chosen in a synchronization, the result is a list of two items, which correspond to the two results of `tcp-accept`. (If the event is not chosen, no connections are accepted.)

```
(tcp-abandon-port tcp-port) → void?
  tcp-port : tcp-port?
```

Like `close-output-port` or `close-input-port` (depending on whether *tcp-port* is an input or output port), but if *tcp-port* is an output port and its associated input port is not yet closed, then the other end of the TCP connection does not receive a TCP close message until the input port is also closed.

The TCP protocol does not include a "no longer reading" state on connections, so `tcp-abandon-port` is equivalent to `close-input-port` on input TCP ports.

```
(tcp-addresses tcp-port [port-numbers?])
 → (or/c (values string? string?)
         (values string? (integer-in 1 65535)
                 string? (integer-in 1 65535)))
  tcp-port : tcp-port?
  port-numbers? : any/c = #f
```

Returns two strings when *port-numbers?* is `#f` (the default). The first string is the Internet address for the local machine a viewed by the given TCP port's connection. (For most machines, the answer corresponds to the current machine's only Internet address, but when a machine serves multiple addresses, the result is connection-specific.) The second string is the Internet address for the other end of the connection.

If *port-numbers?* is true, then four results are returned: a string for the local machine's address, an exact integer between `1` and `65535` for the local machine's port number, a string for the remote machine's address, and an exact integer between `1` and `65535` for the remote machine's port number.

If the given port has been closed, the `exn:fail:network` exception is raised.

```
(tcp-port? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a port returned by `tcp-accept`, `tcp-connect`, `tcp-accept/enable-break`, or `tcp-connect/enable-break`, `#f` otherwise.

14.3.2 UDP

```
(require scheme/udp)
```

The bindings documented in this section are provided by the `scheme/udp` and `scheme` libraries, but not `scheme/base`.

For information about UDP in general, see *TCP/IP Illustrated, Volume 1* by W. Richard Stevens.

```
(udp-open-socket [family-hostname
                  family-port-no]) → udp?
  family-hostname : (or/c string? false/c) = #f
  family-port-no : (or/c string? false/c) = #f
```

Creates and returns a UDP socket to send and receive datagrams (broadcasting is allowed). Initially, the socket is not bound or connected to any address or port.

If *`family-hostname`* or *`family-port-no`* is not `#f`, then the socket's protocol family is determined from these arguments. The socket is *not* bound to the hostname or port number. For example, the arguments might be the hostname and port to which messages will be sent through the socket, which ensures that the socket's protocol family is consistent with the destination. Alternately, the arguments might be the same as for a future call to `udp-bind!`, which ensures that the socket's protocol family is consistent with the binding. If neither *`family-hostname`* nor *`family-port-no`* is non-`#f`, then the socket's protocol family is IPv4.

```
(udp-bind! udp-socket
           hostname-string
           port-no)        → void?
  udp-socket : udp?
  hostname-string : (or/c string? false/c)
  port-no : (and/c nonnegative-exact-integer?
                   (integer-in 1 65535))
```

Binds an unbound *`udp-socket`* to the local port number *`port-no`*.

If *`hostname-string`* is `#f`, then the socket accepts connections to all of the listening machine's IP addresses at *`port-no`*. Otherwise, the socket accepts connections only at the IP address associated with the given name. For example, providing `"127.0.0.1"` as *`hostname-string`* typically creates a listener that accepts only connections to `"127.0.0.1"` from the local machine.

A socket cannot receive datagrams until it is bound to a local address and port. If a socket is not bound before it is used with a sending procedure `udp-send`, `udp-send-to`, etc., the sending procedure binds the socket to a random local port. Similarly, if an event from `udp-send-evt` or `udp-send-to-evt` is chosen for a synchronization (see §10.2.1 "Events"), the socket is bound; if the event is not chosen, the socket may or may not become bound. The binding of a bound socket cannot be changed.

If *udp-socket* is already bound or closed, the `exn:fail:network` exception is raised.

```
(udp-connect! udp-socket
              hostname-string
              port-no)        → void?
  udp-socket : udp?
  hostname-string : (or/c string? false/c)
  port-no : (or/c (and/c nonnegative-exact-integer?
                         (integer-in 1 65535))
                  false/c)
```

Connects the socket to the indicated remote address and port if *hostname-string* is a string and *port-no* is an exact integer.

If *hostname-string* is `#f`, then *port-no* also must be `#f`, and the port is disconnected (if connected). If one of *hostname-string* or *port-no* is `#f` and the other is not, the `exn:fail:contract` exception is raised.

A connected socket can be used with `udp-send` (not `udp-send-to`), and it accepts datagrams only from the connected address and port. A socket need not be connected to receive datagrams. A socket can be connected, re-connected, and disconnected any number of times.

If *udp-socket* is closed, the `exn:fail:network` exception is raised.

```
(udp-send-to udp-socket
             hostname
             port-no
             bstr
            [start-pos
             end-pos])   → void
  udp-socket : udp?
  hostname : string?
  port-no : (and/c nonnegative-exact-integer?
                   (integer-in 1 65535))
  bstr : bytes?
  start-pos : nonnegative-exact-integer? = 0
  end-pos : nonnegative-exact-integer? = (bytes-length bstr)
```

Sends `(subbytes bytes start-pos end-pos)` as a datagram from the unconnected *udp-socket* to the socket at the remote machine `hostname-address` on the port *port-*

no. The *udp-socket* need not be bound or connected; if it is not bound, udp-send-to binds it to a random local port. If the socket's outgoing datagram queue is too full to support the send, udp-send-to blocks until the datagram can be queued.

If *start-pos* is greater than the length of *bstr*, or if *end-pos* is less than *start-pos* or greater than the length of *bstr*, the exn:fail:contract exception is raised.

If *udp-socket* is closed or connected, the exn:fail:network exception is raised.

```
(udp-send udp-socket bstr [start-pos end-pos]) → void
  udp-socket : udp?
  bstr : bytes?
  start-pos : nonnegative-exact-integer? = 0
  end-pos : nonnegative-exact-integer? = (bytes-length bstr)
```

Like udp-send-to, except that *udp-socket* must be connected, and the datagram goes to the connection target. If *udp-socket* is closed or unconnected, the exn:fail:network exception is raised.

```
(udp-send-to* udp-socket
              hostname
              port-no
              bstr
             [start-pos
              end-pos])  → boolean?
  udp-socket : udp?
  hostname : string?
  port-no : (and/c nonnegative-exact-integer?
                   (integer-in 1 65535))
  bstr : bytes?
  start-pos : nonnegative-exact-integer? = 0
  end-pos : nonnegative-exact-integer? = (bytes-length bstr)
```

Like udp-send-to, but never blocks; if the socket's outgoing queue is too full to support the send, #f is returned, otherwise the datagram is queued and the result is #t.

```
(udp-send* udp-socket bstr [start-pos end-pos]) → boolean?
  udp-socket : udp?
  bstr : bytes?
  start-pos : nonnegative-exact-integer? = 0
  end-pos : nonnegative-exact-integer? = (bytes-length bstr)
```

Like udp-send, except that (like udp-send-to) it never blocks and returns #f or #t.

```
(udp-send-to/enable-break udp-socket
                          hostname
                          port-no
                          bstr
                         [start-pos
                          end-pos])  → void
  udp-socket : udp?
  hostname : string?
  port-no : (and/c nonnegative-exact-integer?
                   (integer-in 1 65535))
  bstr : bytes?
  start-pos : nonnegative-exact-integer? = 0
  end-pos : nonnegative-exact-integer? = (bytes-length bstr)
```

Like `udp-send-to`, but breaking is enabled (see §9.6 “Breaks”) while trying to send the datagram. If breaking is disabled when `udp-send-to/enable-break` is called, then either the datagram is sent or the `exn:break` exception is raised, but not both.

```
(udp-send/enable-break udp-socket
                       bstr
                      [start-pos
                       end-pos])  → void
  udp-socket : udp?
  bstr : bytes?
  start-pos : nonnegative-exact-integer? = 0
  end-pos : nonnegative-exact-integer? = (bytes-length bstr)
```

Like `udp-send`, except that breaks are enabled like `udp-send-to/enable-break`.

```
(udp-receive! udp-socket
              bstr
             [start-pos
              end-pos])  → nonnegative-exact-integer?
                           string?
                           (integer-in 1 65535)
  udp-socket : udp?
  bstr : (and/c bytes? (not immutable?))
  start-pos : nonnegative-exact-integer? = 0
  end-pos : nonnegative-exact-integer? = (bytes-length bstr)
```

Accepts up to *end-pos*-*start-pos* bytes of *udp-socket*’s next incoming datagram into *bstr*, writing the datagram bytes starting at position *start-pos* within *bstr*. The *udp-socket* must be bound to a local address and port (but need not be connected). If no incoming datagram is immediately available, `udp-receive!` blocks until one is available.

Three values are returned: the number of received bytes (between `0` and *`end-pos`*-*`start-pos`*, a hostname string indicating the source address of the datagram, and an integer indicating the source port of the datagram. If the received datagram is longer than *`end-pos`*-*`start-pos`* bytes, the remainder is discarded.

If *`start-pos`* is greater than the length of *`bstr`*, or if *`end-pos`* is less than *`start-pos`* or greater than the length of *`bstr`*, the `exn:fail:contract` exception is raised.

```
(udp-receive!* udp-socket
               bstr
              [start-pos
               end-pos])
 → (or/c nonnegative-exact-integer? false/c)
   (or/c string? false/c)
   (or/c (integer-in 1 65535) false/c)
  udp-socket : udp?
  bstr : (and/c bytes? (not immutable?))
  start-pos : nonnegative-exact-integer? = 0
  end-pos : nonnegative-exact-integer? = (bytes-length bstr)
```

Like `udp-receive!`, except that it never blocks. If no datagram is available, the three result values are all `#f`.

```
(udp-receive!/enable-break udp-socket
                           bstr
                          [start-pos
                           end-pos])
 → nonnegative-exact-integer?
   string?
   (integer-in 1 65535)
  udp-socket : udp?
  bstr : (and/c bytes? (not immutable?))
  start-pos : nonnegative-exact-integer? = 0
  end-pos : nonnegative-exact-integer? = (bytes-length bstr)
```

Like `udp-receive!`, but breaking is enabled (see §9.6 "Breaks") while trying to receive the datagram. If breaking is disabled when `udp-receive!/enable-break` is called, then either a datagram is received or the `exn:break` exception is raised, but not both.

```
(udp-close udp-socket) → void?
  udp-socket : udp?
```

Closes *udp-socket*, discarding unreceived datagrams. If the socket is already closed, the `exn:fail:network` exception is raised.

```
(udp? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a socket created by `udp-open-socket`, `#f` otherwise.

```
(udp-bound? udp-socket) → boolean?
  udp-socket : udp?
```

Returns `#t` if *udp-socket* is bound to a local address and port, `#f` otherwise.

```
(udp-connected? udp-socket) → boolean?
  udp-socket : udp?
```

Returns `#t` if *udp-socket* is connected to a remote address and port, `#f` otherwise.

```
(udp-send-ready-evt udp-socket) → evt?
  udp-socket : udp?
```

Returns a synchronizable event (see §10.2.1 "Events") that is in a blocking state when `udp-send-to` on *udp-socket* would block.

```
(udp-receive-ready-evt udp-socket) → evt?
  udp-socket : udp?
```

Returns a synchronizable event (see §10.2.1 "Events") that is in a blocking state when `udp-receive!` on *udp-socket* would block.

```
(udp-send-to-evt  udp-socket
                  hostname
                  port-no
                  bstr
                 [start-pos
                  end-pos])   → evt?
  udp-socket : udp?
  hostname : string?
  port-no : (and/c nonnegative-exact-integer?
                   (integer-in 1 65535))
  bstr : bytes?
```

```
  start-pos : nonnegative-exact-integer? = 0
  end-pos : nonnegative-exact-integer? = (bytes-length bstr)
```

Returns a synchronizable event. The event is in a blocking state when `udp-send-to` on *`udp-socket`* would block. Otherwise, if the event is chosen in a synchronization, data is sent as for `(udp-send-to udp-socket hostname-address port-no bstr start-pos end-pos)`, and the synchronization result is `#<void>`. (No bytes are sent if the event is not chosen.)

```
(udp-send-evt  udp-socket
               bstr
              [start-pos
               end-pos])   → evt?
  udp-socket : udp?
  bstr : bytes?
  start-pos : nonnegative-exact-integer? = 0
  end-pos : nonnegative-exact-integer? = (bytes-length bstr)
```

Returns a synchronizable event. The event is in a blocking state when `udp-send` on *`udp-socket`* would block. Otherwise, if the event is chosen in a synchronization, data is sent as for `(udp-send-to udp-socket bstr start-pos end-pos)`, and the synchronization result is `#<void>`. (No bytes are sent if the event is not chosen.) If *`udp-socket`* is closed or unconnected, the `exn:fail:network` exception is raised during a synchronization attempt.

```
(udp-receive!-evt  udp-socket
                   bstr
                  [start-pos
                   end-pos])   → evt?
  udp-socket : udp?
  bstr : (and/c bytes? (not immutable?))
  start-pos : nonnegative-exact-integer? = 0
  end-pos : nonnegative-exact-integer? = (bytes-length bstr)
```

Returns a synchronizable event. The event is in a blocking state when `udp-receive` on *`udp-socket`* would block. Otherwise, if the event is chosen in a synchronization, data is received into *`bstr`* as for `(udp-receive! udp-socket bytes start-pos end-pos)`, and the synchronization result is a list of three values, corresponding to the three results from `udp-receive!`. (No bytes are received and the *`bstr`* content is not modified if the event is not chosen.)

14.4 Processes

```
(subprocess stdout
            stdin
            stderr
            command
            arg ...) → subprocess?
                       (or/c input-port? false/c)
                       (or/c output-port? false/c)
                       (or/c input-port? false/c)
  stdout : (or/c output-port? false/c)
  stdin : (or/c input-port? false/c)
  stderr : (or/c output-port? false/c)
  command : path-string?
  arg : string?
(subprocess stdout
            stdin
            stderr
            command
            exact
            arg)    → subprocess?
                      (or/c input-port? false/c)
                      (or/c output-port? false/c)
                      (or/c input-port? false/c)
  stdout : (or/c output-port? false/c)
  stdin : (or/c input-port? false/c)
  stderr : (or/c output-port? false/c)
  command : path-string?
  exact : (one-of/c 'exact)
  arg : string?
```

Creates a new process in the underlying operating system to execute *command* asynchronously. See also `system` and `process` from `scheme/system`.

The *command* argument is a path to a program executable, and the *arg*s are command-line arguments for the program. Under Unix and Mac OS X, command-line arguments are passed as byte strings using the current locale's encoding (see §12.1.1 "Encodings and Locales").

Under Windows, the first *arg* can be replaced with `'exact`, which triggers a Windows-specific behavior: the sole *arg* is used exactly as the command-line for the subprocess. Otherwise, under Windows, a command-line string is constructed from *command* and *arg* so that a typical Windows console application can parse it back to an array of arguments. If `'exact` is provided on a non-Windows platform, the `exn:fail:contract` exception is raised.

For information on the Windows command-line conventions, search for "command line parsing" at `http://msdn.microsoft.com/`.

Unless it is `#f`, *stdout* is used for the launched process's standard output, *stdin* is used for the process's standard input, and *stderr* is used for the process's standard error. All provided ports must be file-stream ports. Any of the ports can be `#f`, in which case a system pipe is created and returned by `subprocess`. For each port that is provided, no pipe is created and the corresponding returned value is `#f`.

The `subprocess` procedure returns four values:

- a subprocess value representing the created process;
- an input port piped from the process's standard output, or `#f` if `stdout-output-port` was a port;
- an output port piped to the process standard input, or `#f` if `stdin-input-port` was a port;
- an input port piped from the process's standard error, or `#f` if `stderr-output-port` was a port.

Important: All ports returned from `subprocess` must be explicitly closed with `close-input-port` or `close-output-port`.

The returned ports are file-stream ports (see §12.1.5 "File Ports"), and they are placed into the management of the current custodian (see §13.6 "Custodians"). The `exn:fail` exception is raised when a low-level error prevents the spawning of a process or the creation of operating system pipes for process communication.

```
(subprocess-wait subproc) → void?
  subproc : subprocess?
```

Blocks until the process represented by *`subproc`* terminates.

```
(subprocess-status subproc) → (or/c (one-of/c 'running)
                                    nonnegative-exact-integer?)
  subproc : subprocess?
```

Returns `'running` if the process represented by *`subproc`* is still running, or its exit code otherwise. The exit code is an exact integer, and `0` typically indicates success. If the process terminated due to a fault or signal, the exit code is non-zero.

```
(subprocess-kill subproc force?) → void?
  subproc : subprocess?
  force? : any/c
```

Terminates the subprocess represented by *`subproc`* if *`force?`* is true and if the process still running. If an error occurs during termination, the `exn:fail` exception is raised.

If *force?* is `#f` under Unix and Mac OS X, the subprocess is sent an interrupt signal instead of a kill signal (and the subprocess might handle the signal without terminating). Under Windows, no action is taken when *force?* is `#f`.

```
(subprocess-pid subproce) → nonnegative-exact-integer?
  subproce : subprocess?
```

Returns the operating system's numerical ID (if any) for the process represented by `subproc`, valid only as long as the process is running.

```
(subprocess? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a subprocess value, `#f` otherwise.

```
(shell-execute verb
               target
               parameters
               dir
               show-mode) → false/c
  verb : (or/c string? false/c)
  target : string?
  parameters : string?
  dir : path-string?
  show-mode : symbol?
```

Performs the action specified by `verb` on `target` in Windows. For platforms other than Windows, the `exn:fail:unsupported` exception is raised.

For example,

```
(shell-execute #f "http://www.plt-scheme.org" ""
                (current-directory) 'sw_shownormal)
```

Opens the PLT Scheme home page in a browser window.

The `verb` can be `#f`, in which case the operating system will use a default verb. Common verbs include `"open"`, `"edit"`, `"find"`, `"explore"`, and `"print"`.

The `target` is the target for the action, usually a filename path. The file could be executable, or it could be a file with a recognized extension that can be handled by an installed application.

The `parameters` argument is passed on to the system to perform the action. For example, in the case of opening an executable, the `parameters` is used as the command line (after the executable name).

The `dir` is used as the current directory when performing the action.

The `show-mode` sets the display mode for a Window affected by the action. It must be one of the following symbols; the description of each symbol's meaning is taken from the Windows API documentation.

- `'sw_hide` or `'SW_HIDE` — Hides the window and activates another window.
- `'sw_maximize` or `'SW_MAXIMIZE` — Maximizes the window.
- `'sw_minimize` or `'SW_MINIMIZE` — Minimizes the window and activates the next top-level window in the z-order.
- `'sw_restore` or `'SW_RESTORE` — Activates and displays the window. If the window is minimized or maximized, Windows restores it to its original size and position.
- `'sw_show` or `'SW_SHOW` — Activates the window and displays it in its current size and position.
- `'sw_showdefault` or `'SW_SHOWDEFAULT` — Uses a default.
- `'sw_showmaximized` or `'SW_SHOWMAXIMIZED` — Activates the window and displays it as a maximized window.
- `'sw_showminimized` or `'SW_SHOWMINIMIZED` — Activates the window and displays it as a minimized window.
- `'sw_showminnoactive` or `'SW_SHOWMINNOACTIVE` — Displays the window as a minimized window. The active window remains active.
- `'sw_showna` or `'SW_SHOWNA` — Displays the window in its current state. The active window remains active.
- `'sw_shownoactivate` or `'SW_SHOWNOACTIVATE` — Displays a window in its most recent size and position. The active window remains active.
- `'sw_shownormal` or `'SW_SHOWNORMAL` — Activates and displays a window. If the window is minimized or maximized, Windows restores it to its original size and position.

If the action fails, the `exn:fail` exception is raised. If the action succeeds, the result is `#f`.

In future versions of Scheme, the result may be a subprocess value if the operating system did returns a process handle (but if a subprocess value is returned, its process ID will be `0` instead of the real process ID).

14.4.1 Simple Subprocesses

```
(require scheme/system)
```

The bindings documented in this section are provided by the `scheme/system` library, not `scheme/base` or `scheme`.

```
(system command) → boolean?
  command : string?
```

Executes a Unix, Mac OS X, or Windows shell command synchronously (i.e., the call to `system` does not return until the subprocess has ended). The *command* argument is a string containing no nul characters. If the command succeeds, the return value is `#t`, `#f` otherwise.

```
(system* command arg ...) → boolean?
  command : path-string?
  arg : string?
(system* command exact arg) → boolean?
  command : path-string?
  exact : (one-of/c 'exact)
  arg : string?
```

Like `system`, except that *command* is a filename that is executed directly (instead of through a shell command), and the *arg*s are the arguments. The executed file is passed the specified string arguments (which must contain no nul characters).

Under Windows, the first argument after *command* can be `'exact`, and the final *arg* is a complete command line. See `subprocess` for details.

```
(system/exit-code command) → (integer-in 0 255)
  command : string?
```

Like `system`, except that the result is the exit code returned by the subprocess. A `0` result normally indicates success.

```
(system*/exit-code command arg ...) → (integer-in 0 255)
  command : path-string?
  arg : string?
(system*/exit-code command exact arg) → (integer-in 0 255)
  command : path-string?
  exact : (one-of/c 'exact)
  arg : string?
```

Like system*, but returns the exit code like system/exit-code.

```
(process command)
 → (list input-port?
         output-port?
         nonnegative-exact-integer?
         input-port?
         ((one-of/c 'status 'wait 'interrupt 'kill) . -> . any))
  command : string?
```

Executes a shell command asynchronously. 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, and
- a procedure of one argument, either 'status, 'wait, 'interrupt, or 'kill:
  - 'status returns the status of the subprocess as one of 'running, 'done-ok, or 'done-error.
  - 'exit-code returns the integer exit code of the subprocess or #f if it is still running.
  - 'wait blocks execution in the current thread until the subprocess has completed.
  - 'interrupt sends the subprocess an interrupt signal under Unix and Mac OS X, and takes no action under Windows. The result is #<void>.
  - 'kill terminates the subprocess and returns #<void>.

Important: All three ports returned from process must be explicitly closed with close-input-port or close-output-port.

```
(process* command arg ...) → list?
  command : path-string?
  arg : string?
(process* command exact arg) → list?
  command : path-string?
  exact : (one-of/c 'exact)
  arg : string?
```

Like process, except that *command* is a filename that is executed directly, and the *arg*s are the arguments. Under Windows, as for `system*`, the first *arg* can be replaced with `'exact`.

```
(process/ports out in error-out command) → list?
  out : (or/c false/c output-port?)
  in : (or/c false/c input-port?)
  error-out : (or/c false/c output-port?)
  command : string?
```

Like `process`, except that *out* is used for the process's standard output, *in* is used for the process's standard input, and *error-out* is used for the process's standard error. Any of the ports can be `#f`, in which case a system pipe is created and returned, as in `process`. For each port that is provided, no pipe is created, and the corresponding value in the returned list is `#f`.

```
(process*/ports out
                in
                error-out
                command
                arg ...)   → list?
  out : (or/c false/c output-port?)
  in : (or/c false/c input-port?)
  error-out : (or/c false/c output-port?)
  command : path-string?
  arg : string?
(process*/ports out
                in
                error-out
                command
                exact
                arg)       → list?
  out : (or/c false/c output-port?)
  in : (or/c false/c input-port?)
  error-out : (or/c false/c output-port?)
  command : path-string?
  exact : (one-of/c 'exact)
  arg : string?
```

Like `process*`, but with the port handling of `process/ports`.

14.5 Time

```
(current-seconds) → exact-integer?
```

Returns the current time in seconds. This time is always an exact integer based on a platform-specific starting date (with a platform-specific minimum and maximum value).

The value of `(current-seconds)` increases as time passes (increasing by 1 for each second that passes). The current time in seconds can be compared with a time returned by `file-or-directory-modify-seconds`.

```
(seconds->date secs-n) → date?
  secs-n : exact-integer?
```

Takes *secs-n*, a platform-specific time in seconds returned by `current-seconds` or `file-or-directory-modify-seconds`, and returns an instance of the `date` structure type. If *secs-n* is too small or large, the `exn:fail` exception is raised.

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

```
(struct date (second
              minute
              hour
              day
              month
              year
              week-day
              year-day
              dst?
              time-zone-offset)
         #:transparent)
  second : (integer-in 0 61)
  minute : (integer-in 0 59)
  hour : (integer-in 0 23)
  day : (integer-in 1 31)
  month : (integer-in 1 12)
  year : exact-nonnegative-integer?
  week-day : (integer-in 0 6)
  year-day : (integer-in 0 365)
  dst? : boolean?
  time-zone-offset : exact-integer?
```

Represents a date. For the `second` field, values of `60` and `61` are for unusual, but possible for leap-seconds. The `year-day` field reaches `365` only in leap years.

The `time-zone-offset` field reports the number of seconds east of GMT for the current time zone (e.g., Pacific Standard Time is `-28800`), an exact integer.

The value produced for the `time-zone-offset` field tends to be sensitive to the value of the `TZ` environment variable, especially on Unix platforms; consult the system documentation (usually under `tzset`) for details.

See also the `scheme/date` library.

```
(current-milliseconds) → exact-integer?
```

Returns the current "time" in fixnum milliseconds (possibly negative). This time is based on a platform-specific starting date or on the machine's startup time. Since the result is a fixnum, the value increases only over a limited (though reasonably long) time.

```
(current-inexact-milliseconds) → real?
```

Like `current-milliseconds`, but the result never decreases (until the machine is turned off).

```
(current-process-milliseconds) → exact-integer?
```

Returns the amount of processor time in fixnum milliseconds that has been consumed by the Scheme process on the underlying operating system. (Under Unix and Mac OS X, this includes both user and system time.) The precision of the result is platform-specific, and since the result is a fixnum, the value increases only over a limited (though reasonably long) time.

```
(current-gc-milliseconds) → exact-integer?
```

Returns the amount of processor time in fixnum milliseconds that has been consumed by Scheme's garbage collection so far. This time is a portion of the time reported by `(current-process-milliseconds)`, and is similarly limited.

```
(time-apply proc arg ...) → list?
                            exact-integer?
                            exact-integer?
                            exact-integer?
  proc : procedure?
  arg : any/c
```

Collects timing information for a procedure application.

Four values are returned: a list containing the result(s) of applying *proc*, the number of milliseconds of CPU time required to obtain this result, the number of "real" milliseconds required for the result, and the number of milliseconds of CPU time (included in the first result) spent on garbage collection.

The reliability of the timing numbers depends on the platform. If multiple MzScheme threads are running, then the reported time may include work performed by other threads.

```
(time expr)
```

Reports `time-apply`-style timing information for the evaluation of *`expr`* directly to the current output port. The result is the result of *`expr`*.

14.5.1 Date Utilities

```
(require scheme/date)
```

```
(date->string date [time?]) → string?
  date : date?
  time? : any/c = #f
```

Converts a date to a string. The returned string contains the time of day only if *`time?`*. See also `date-display-format`.

```
(date-display-format) → (one-of/c 'american
                                  'chinese
                                  'german
                                  'indian
                                  'irish
                                  'iso-8601
                                  'rfc2822
                                  'julian)
(date-display-format format) → void?
  format : (one-of/c 'american
                     'chinese
                     'german
                     'indian
                     'irish
                     'iso-8601
                     'rfc2822
                     'julian)
```

Parameter that determines the date string format. The initial format is `'american`.

```
(find-seconds second
              minute
              hour
              day
              month
              year)  → exact-integer?
  second : (integer-in 0 61)
  minute : (integer-in 0 59)
  hour : (integer-in 0 23)
  day : (integer-in 1 31)
  month : (integer-in 1 12)
  year : exact-nonnegative-integer?
```

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 signaled, otherwise an integer is returned.

```
(date->julian/scalinger date) → exact-integer?
  date : date?
```

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-number) → string?
  date-number : exact-integer?
```

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

14.6 Environment and Runtime Information

```
(getenv name) → (or/c string? false/c)
  name : string?
```

Gets the value of an operating system environment variable. The *name* argument cannot contain a null character; if an environment variable named by *name* exists, its value is returned (as a string); otherwise, `#f` is returned.

```
(putenv name value) → boolean?
  name : string?
```

`value : string?`

Sets the value of an operating system environment variable. The `name` and `value` arguments are strings that cannot contain a null character; the environment variable named by `name` is set to `value`. The return value is `#t` if the assignment succeeds, `#f` otherwise.

```
(system-type [mode]) → (or/c symbol? string? bytes?)
  mode : (one-of 'os 'gc 'link 'so-suffix 'machine) = 'os
```

Returns information about the operating system, build mode, or machine for a running Scheme.

In `'os` mode, the possible symbol results are:

- `'unix`
- `'windows`
- `'macosx`

In `'gc` mode, the possible symbol results are:

- `'cgc`
- `'3m`

In `'link` mode, the possible symbol results are:

- `'static` (Unix)
- `'shared` (Unix)
- `'dll` (Windows)
- `'framework` (Mac OS X)

Future ports of Scheme may expand the list of `'os`, `'gc`, and `'link` results.

In `'so-suffix` mode, then the result is a byte string that represents the file extension used for shared objects on the current platform. The byte string starts with a period, so it is suitable as a second argument to `path-replace-suffix`.

In `'machine` mode, then the result is a string, which contains further details about the current machine in a platform-specific format.

`(system-language+country) → string?`

Returns a string to identify the current user's language and country.

Under Unix and Mac OS X, the string is five characters: two lowercase ASCII letters for the language, an underscore, and two uppercase ASCII letters for the country. Under Windows, the string can be arbitrarily long, but the language and country are in English (all ASCII letters or spaces) separated by an underscore.

Under Unix, the result is determined by checking the `LC_ALL`, `LC_TYPE`, and `LANG` environment variables, in that order (and the result is used if the environment variable's value starts with two lowercase ASCII letters, an underscore, and two uppercase ASCII letters, followed by either nothing or a period). Under Windows and Mac OS X, the result is determined by system calls.

```
(system-library-subpath [mode]) → path?
  mode : (one-of 'cgc '3m #f) = (system-type 'gc)
```

Returns a relative directory path. This string can be used to build paths to system-specific files. For example, when Scheme is running under Solaris on a Sparc architecture, the subpath starts `"sparc-solaris"`, while the subpath for Windows on an i386 architecture starts `"win32\\i386"`.

The optional *mode* argument specifies the relevant garbage-collection variant, which one of the possible results of `(system-type 'gc)`: `'cgc` or `'3m`. It can also be `#f`, in which case the result is independent of the garbage-collection variant.

`(version) → (and/c string? immutable?)`

Returns an string indicating the currently executing version of Scheme.

`(banner) → (and/c string? immutable?)`

Returns an immutable string for Scheme's start-up banner text (or the banner text for an embedding program, such as MrEd). The banner string ends with a newline.

```
(current-command-line-arguments)
 → (vectorof (and/c string? immutable?))
(current-command-line-arguments argv) → void?
  argv : (vectorof (and/c string? immutable?))
```

A parameter that is initialized with command-line arguments when Scheme starts (not including any command-line arguments that were treated as flags for the system).

```
(current-thread-initial-stack-size) → exact-positive-integer?
(current-thread-initial-stack-size size) → void?
  size : exact-positive-integer?
```

A parameter that provides a hint about how much space to reserve for a newly created thread's local variables. The actual space used by a computation is affected by just-in-time (JIT) compilation, but it is otherwise platform-independent.

```
(vector-set-performance-stats! results [thd]) → void?
  results : (and/c vector?
                   (not/c immutable?))
  thd : (or/c thread? false/c) = #f
```

Sets elements in `results` to report current performance statistics. If `thd` is not `#f`, a particular set of thread-specific statistics are reported, otherwise a different set of global statics are reported.

For global statistics, up to 10 elements are set in the vector, starting from the beginning. (In future versions of Scheme, additional elements will be set.) If `results` has n elements where $n < 8$, then the n elements are set to the first n performance-statistics values. The reported statistics values are as follows, in the order that they are set within `results`:

- `0`: The same value as returned by `current-process-milliseconds`.
- `1`: The same value as returned by `current-milliseconds`.
- `2`: The same value as returned by `current-gc-milliseconds`.
- `3`: The number of garbage collections performed since start-up.
- `4`: The number of thread context switches performed since start-up.
- `5`: The number of internal stack overflows handled since start-up.
- `6`: The number of threads currently scheduled for execution (i.e., threads that are running, not suspended, and not unscheduled due to a synchronization).
- `7`: The number of syntax objects read from compiled code since start-up.
- `8`: The number of hash-table searches performed.
- `9`: The number of additional hash slots searched to complete hash searches (using double hashing).
- `10`: The number of bytes allocated for machine code that is not reported by `current-memory-use`.

For thread-specific statistics, up to 4 elements are set in the vector:

- `0`: `#t` if the thread is running, `#f` otherwise (same result as `thread-running?`).
- `1`: `#t` if the thread has terminated, `#f` otherwise (same result as `thread-dead?`).
- `2`: `#t` if the thread is currently blocked on a synchronizable event (or sleeping for some number of milliseconds), `#f` otherwise.
- `3`: The number of bytes currently in use for the thread's continuation.

14.7 Command-Line Parsing

```
(require scheme/cmdline)
```

The bindings documented in this section are provided by the `scheme/cmdline` and `scheme` libraries, but not `scheme/base`.

```
(command-line optional-name-expr optional-argv-expr
              flag-clause ...
              finish-clause)
```

```
optional-name-expr =
                   | #:name name-expr

optional-argv-expr =
                   | #:argv argv-expr

       flag-clause = #:multi flag-spec ...
                   | #:once-each flag-spec ...
                   | #:once-any flag-spec ...
                   | #:final flag-spec ...
                   | #:help-labels string ...

         flag-spec = (flags id ... help-spec body ...+)
                   | (flags => handler-expr help-expr)

             flags = flag-string
                   | (flag-string ...+)

         help-spec = string
                   | (string-expr ...+)

     finish-clause =
                   | #:args arg-formals body ...+
                   | #:handlers handlers-exprs

       arg-formals = id
                   | (id ...)
                   | (id ...+ . id)

    handlers-exprs = finish-expr arg-strings-expr
                   | finish-expr arg-strings-expr help-expr
                   | finish-expr arg-strings-expr help-expr
                     unknown-expr
```

Parses a command line according to the specification in the `flag-clause`s.

The `name-expr`, if provided, should produce a path or string to be used as the program name for reporting errors when the command-line is ill-formed. It defaults to `(find-system-path 'run-file)`. When a path is provided, only the last element of the path is used to report an error.

The `argv-expr`, if provided, must evaluate to a list or a vector of strings. It defaults to `(current-command-line-arguments)`.

The command-line is disassembled into flags, each 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.

A `#:multi`, `#:once-each`, `#:once-any`, or `#:final` 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:fail` 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:fail` exception is raised.
- `#:final` — Like `#:multi`, except that no argument after the flag is treated as a flag. Note that multiple `#:final` flags can be specified if they have short names; for example, if `-a` is a `#:final` flag, then `-aa` combines two instances of `-a` in a single command-line argument.

A normal flag specification has four parts:

- *`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.
- *`id`*s — identifier that are bound to the flag's arguments. The number of identifiers determines how many arguments can be provided on the command line with the flag, and the names of these identifiers will appear in the help message describing the flag. The *`id`*s are bound to string values in the *`body`*s for handling the flag.
- *`help-spec`* — a string or sequence of strings that describes the flag. This string is used in the help message generated by the handler for the built-in `-h` (or `--help`) flag. A single literal string can be provided, or any number of expressions that produce strings; in the latter case, strings after the first one are displayed on subsequent lines.

- *bodys* — 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 *bodys* is evaluated as the corresponding flag is encountered. When the *bodys* are evaluated, the preceding *ids* 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 used as in the *table* argument of `parse-command-line`.

A `#:help-labels` clause inserts text lines into the help table of command-line flags. Each string in the clause provides a separate line of text.

After the flag clauses, a final clause handles command-line arguments that are not parsed as flags:

- Supplying no finish clause is the same as suppling `#:args () (void)`.
- For an `#:args` finish clause, identifiers in *arg-formals* are bound to the leftover command-line strings in the same way that identifiers are bound for a `lambda` expression. Thus, specifying a single *id* (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 identifiers 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 identifiers in *arg-formals*, the `exn:fail` exception is raised. Otherwise, `args` clause's *bodys* are evaluated to handle the leftover arguments, and the result of the last *body* is the result of the `command-line` expression.
- A `#:handlers` finish clause escapes to a more general method of handling the leftover arguments. In this case, the values of the expressions are used like the last two to four arguments `parse-command-line`.

Example:

```
(define verbose-mode (make-parameter #f))
(define profiling-on (make-parameter #f))
(define optimize-level (make-parameter 0))
(define link-flags (make-parameter null))

(define file-to-compile
  (command-line
   #:program "compiler"
   #: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"        (; show help on separate lines
                        "Compile with optimization level 2,"
                        "which includes all of level 1")
                       (optimize-level 2)]
#:multi
[("-l" "--link-flags") lf ; flag takes one argument
                       "Add a flag <lf> for the linker"
                       (link-flags (cons lf (link-flags)))]
#:args (filename) ; expect one command-line argument: <filename>
; return the argument as a filename to compile
filename))
```

```
(parse-command-line  name
                     argv
                     table
                     finish-proc
                     arg-help-strs
                    [help-proc
                     unknown-proc]) → any
  name : (or/c string? path?)
  argv : (or/c (listof string?) (vectorof string?))
  table : (listof (list/c symbol? list?))
  finish-proc : ((list?) list? . ->* . any)
  arg-help-strs : (listof string?)
  help-proc : (string? . -> . any) = (lambda (str) ....)
  unknown-proc : (string? . -> . any) = (lambda (str) ...)
```

Parses a command-line using the specification in *table*. For an overview of command-line parsing, see the `command-line` form, which provides a more convenient notation for most purposes.

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-strs*. 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, *finish-proc* 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 *finish-proc* 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 `procedure-arity`) except `0` or a list of `0`s (i.e., the procedure must at least accept one or more arguments).

The *arg-help-strs* 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-strs* 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:fail` 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 either help strings or flag specifications. A mode symbol is one of `'once-each`, `'once-any`, `'multi`, `'final`, or `'help-labels`, with the same meanings as the corresponding clause tags in `command-line`. For the `'help-labels` mode, a list of help string is provided. For the other modes, a list of flag specifications is provided, where each specification maps a number of flags to a single handler procedure. A specification is a list of three items:

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

  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.
- A non-empty list for constructing help information for the spec. The first element of the list describes the flag; it can be a string or a non-empty list of strings, and in the latter case, each string is shown on its own line. Additional elements of the main list must be strings to 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 core example for `command-line`, translated to the procedural form:

```
(parse-command-line "compile" (current-command-line-arguments)
  `((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")]
     [("--optimize-2")
      ,(lambda (flag) (optimize-level 2))
      (("Compile with optimization level 2,"
        "which implies all optimizations of level 1"))])
    (multi
     [("-l" "--link-flags")
      ,(lambda (flag lf) (link-flags (cons lf (link-flags))))
      ("Add a flag <lf> for the linker" "lf")]))
   (lambda (flag-accum file) file)
   '("filename"))
```

15 Memory Management

15.1 Weak Boxes

A *weak box* is similar to a normal box (see §3.12 "Boxes"), but when the garbage collector (see §1.1.7 "Garbage Collection") can prove that the content value of a weak box is only reachable via weak references, the content of the weak box is replaced with `#f`. A *weak reference* is a reference through a weak box, through a key reference in a weak hash table (see §3.13 "Hash Tables"), through a value in an ephemeron where the value can be replaced by `#f` (see §15.2 "Ephemerons"), or through a custodian (see §13.6 "Custodians").

```
(make-weak-box v) → weak-box?
  v : any/c
```

Returns a new weak box that initially contains `v`.

```
(weak-box-value weak-box) → any
  weak-box : weak-box?
```

Returns the value contained in `weak-box`. If the garbage collector has proven that the previous content value of `weak-box` was reachable only through a weak reference, then `#f` is returned.

```
(weak-box? v) → boolean?
  v : any/c
```

Returns `#t` if `v` is a weak box, `#f` otherwise.

15.2 Ephemerons

An *ephemeron* is similar to a weak box (see §15.1 "Weak Boxes"), except that

- an ephemeron contains a key and a value; the value can be extracted from the ephemeron, but the value is replaced by `#f` when the automatic memory manager can prove that either the ephemeron or the key is reachable only through weak references (see §15.1 "Weak Boxes"); and
- nothing reachable from the value in an ephemeron counts toward the reachability of an ephemeron key (whether for the same ephemeron or another), unless the same value is reachable through a non-weak reference, or unless the value's ephemeron key is

reachable through a non-weak reference (see §15.1 "Weak Boxes" for information on weak references).

In particular, an ephemeron can be combined with a weak hash table (see §3.13 "Hash Tables") to produce a mapping where the memory manager can reclaim key–value pairs even when the value refers to the key.

```
(make-ephemeron key v) → ephemeron?
  key : any/c
  v : any/c
```

Returns a new ephemeron whose key is `key` and whose value is initially `v`.

```
(ephemeron-value ephemeron) → any
  ephemeron : ephemeron?
```

Returns the value contained in `ephemeron`. If the garbage collector has proven that the key for `ephemeron` is only weakly reachable, then the result is `#f`.

```
(ephemeron? v) → boolean?
  v : any/c
```

Returns `#t` if `v` is an ephemeron, `#f` otherwise.

15.3 Wills and Executors

A *will executor* manages a collection of values and associated *will* procedures. The will procedure for each value is ready to be executed when the value has been proven (by the garbage collector) to be unreachable, except through weak references (see §15.1 "Weak Boxes") or as the registrant for other will executors. A will is useful for triggering clean-up actions on data associated with an unreachable value, such as closing a port embedded in an object when the object is no longer used.

Calling the `will-execute` or `will-try-execute` procedure executes a will that is ready in the specified will executor. Wills are not executed automatically, because certain programs need control to avoid race conditions. However, a program can create a thread whose sole job is to execute wills for a particular executor.

If a value is registered with multiple wills (in one or multiple executors), the wills are readied in the reverse order of registration. Since readying a will procedure makes the value reachable again, the will must be executed and the value must be proven again unreachable through only weak references before another of the wills is readied or executed. However, wills for distinct unreachable values are readied at the same time, regardless of whether the values are reachable from each other.

A will executor's register is held non-weakly until after the corresponding will procedure is executed. Thus, if the content value of a weak box (see §15.1 "Weak Boxes") is registered with a will executor, the weak box's content is not changed to `#f` until all wills have been executed for the value and the value has been proven again reachable through only weak references.

`(make-will-executor)` → `will-executor?`

Returns a new will executor with no managed values.

`(will-executor? v)` → `boolean?`
  `v : any/c`

Returns `#t` if *v* is a will executor, `#f` otherwise.

`(will-register executor v proc)` → `void?`
  `executor : will-executor?`
  `v : any/c`
  `proc : (any/c . -> . any)`

Registers the value *v* with the will procedure *proc* in the will executor *executor*. When *v* is proven unreachable, then the procedure *proc* is ready to be called with *v* as its argument via `will-execute` or `will-try-execute`. The *proc* argument is strongly referenced until the will procedure is executed.

`(will-execute executor)` → `any`
  `executor : will-executor?`

Invokes the will procedure for a single "unreachable" value registered with the executor `executable`. The values returned by the will procedure are 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)` → `any`
  `executor : any/c`

Like `will-execute` if a will is ready for immediate execution. Otherwise, `#f` is returned.

15.4 Garbage Collection

```
(collect-garbage) → void?
```

Forces an immediate garbage collection. Some effectively unreachable data may remain uncollected, because the collector cannot prove that it is unreachable.

The `collect-garbage` procedure provides some control over the timing of collections, but garbage will obviously be collected even if this procedure is never called.

```
(current-memory-use [cust]) → nonnegative-exact-integer?
  cust : custodian? = #f
```

Returns an estimate of the number of bytes of memory occupied by reachable data from *cust*. (The estimate is calculated *without* performing an immediate garbage collection; performing a collection generally decreases the number returned by `current-memory-use`.) If *cust* is not provided, the estimate is a total reachable from any custodians.

When PLT Scheme is compiled without support for memory accounting, the estimate is the same (i.e., all memory) for any individual custodian; see also `custodian-memory-accounting-available?`.

```
(dump-memory-stats) → any
```

Dumps information about memory usage to the (low-level) standard output port.

16 Running PLT Scheme

16.1 Starting MzScheme or MrEd

The core PLT Scheme run-time system is available in two main variants:

- MzScheme, which provides the primitives libraries on which `scheme/base` is implemented. Under Unix and Mac OS X, the executable is called `mzscheme`. Under Windows, the executable is called `MzScheme.exe`.
- MrEd, which extends `mzscheme` with GUI primitives on which `scheme/gui/base` is implemented. Under Unix, the executable is called `mred`. Under Windows, the executable is called `MrEd.exe`. Under Mac OS X, the `mred` script launches `MrEd.app`.

16.1.1 Initialization

On startup, the top-level environment contains no bindings—not even for function application. Primitive modules with names that start with `#%` are defined, but they are not meant for direct use, and the set of such modules can change. For example, the `'#%kernel` module is eventually used to bootstrap the implemetation of `scheme/base`, and `'#%mred-kernel` is used for `scheme/gui/base`.

The first action of MzScheme or MrEd is to initialize `current-library-collection-paths` to the result of `(find-library-collection-paths extras)`, where *`extras`* are extra directory paths provided in order in the command line with `-S`/`--search`. An executable created from the MzScheme or MrEd executable can embed additional paths that are appended to *`extras`*.

MzScheme and MrEd next `require` `scheme/init` and `scheme/gui/init`, respectively, but only if the command line does not specify a `require` flag (`-t`/`--require`, `-l`/`--lib`, or `-u`/`--require-script`) before any `eval`, `load`, or read-eval-print-loop flag (`-e`/`--eval`, `-f`/`--load`, `-r`/`--script`, `-m`/`--main`, `-i`/`--repl`, or `-z`/`--text-repl`). The initialization library can be changed with the `-I` configuration option.

After potentially loading the initialization module, expression `eval`s, files `load`s, and module `require`s are executed in the order that they are provided on the command line. If any raises an uncaught exception, then the remaining `eval`s, `load`s, and `require`s are skipped.

After running all command-line expressions, files, and modules, MzScheme or MrEd then starts a read-eval-print loop for interactive evaluation if no command line flags are provided other than configuration options. If any command-line argument is provided that is not a configuration option, then the read-eval-print-loop is not started, unless the `-i`/`--repl` or `-z`/`--text-repl` flag is provided on the command line to specifically re-enable it. In addition, just before the command line is started, MzScheme loads the file `(find-system-path 'init-file)` and MrEd loads the file `(find-graphical-system-path 'init-file)` is loaded, unless the `-q`/`--no-init-file` flag is specified on the command line.

Finally, before MrEd exists, it waits for all frames to class, all timers to stop, etc. in the main eventspace by evaluating `(scheme 'yield)`. This waiting step can be suppressed with the `-V`/`--no-yield` command-line flag.

The exit status for the MzScheme or MrEd process indicates an error if an error occurs during a command-line `eval`, `load`, or `require` when no read-eval-print loop is started. Otherwise, the exit status is `0` or determined by a call to `exit`.

16.1.2 Init Libraries

```
(require scheme/init)
```

The `scheme/init` library is the default start-up library for MzScheme. It re-exports the `scheme` and `scheme/help` libraries, and it sets `current-print` to use `pretty-print`.

```
(require scheme/gui/init)
```

The `scheme/gui/init` library is the default start-up library for MrEd. It re-exports the `scheme/init` and `scheme/gui/base` libraries, and it sets `current-load` to use `text-editor-load-handler`.

16.1.3 mz-cmdlineCommand Line

The MzScheme and MrEd executables recognize the following command-line flags:

- File and expression options:
  - `-e` ⟨*expr*⟩ or `--eval` ⟨*expr*⟩ : `eval`s ⟨*expr*⟩. The results of the evaluation are printed via `current-print`.
  - `-f` ⟨*file*⟩ or `--load` ⟨*file*⟩ : `load`s ⟨*file*⟩.
  - `-t` ⟨*file*⟩ or `--require` ⟨*file*⟩ : `require`s ⟨*file*⟩.
  - `-l` ⟨*path*⟩ or `--lib` ⟨*path*⟩ : `require`s `(lib "`⟨*path*⟩`")`.
  - `-p` ⟨*file*⟩ ⟨*u*⟩ ⟨*path*⟩ : `require`s `(planet "`⟨*file*⟩`" "`⟨*user*⟩`" "`⟨*pkg*⟩`")`.
  - `-r` ⟨*file*⟩ or `--script` ⟨*file*⟩ : `load`s ⟨*file*⟩ as a script. This flag is like `-t` ⟨*file*⟩ plus `-N` ⟨*file*⟩ to set the program name and `--` to cause all further command-line elements to be treated as non-flag arguments.

- `-u` ⟨*file*⟩ or `--require-script` ⟨*file*⟩ : `requires` ⟨*file*⟩ as a script; This flag is like `-t` ⟨*file*⟩ plus `-N` ⟨*file*⟩ to set the program name and `--` to cause all further command-line elements to be treated as non-flag arguments.
- `-k` ⟨*n*⟩ ⟨*m*⟩ : Loads code embedded in the executable from file position ⟨*n*⟩ to ⟨*m*⟩. This option is normally embedded in a stand-alone binary that also embeds Scheme code.
- `-m` or `--main` : Evaluates a call to `main` in the top-level environment. All of the command-line arguments that are not processed as options (i.e., the arguments put into `current-command-line-arguments`) are passed as arguments to `main`. The results of the call are printed via `current-print`.

• Interaction options:

- `-i` or `--repl` : Runs interactive read-eval-print loop, using either `read-eval-print-loop` (MzScheme) or `graphical-read-eval-print-loop` (MrEd) after showing `(banner)` and loading `(find-system-path 'init-file)`.
- `-z` or `--text-repl` : MrEd only; like `-i`/`--repl`, but uses `textual-read-eval-print-loop` instead of `graphical-read-eval-print-loop`.
- `-n` or `--no-lib` : Skips requiring the initialization library (i.e., `scheme/init` or `scheme/gui/init`, unless it is changed with the `-I` flag) when not otherwise disabled.
- `-v` or `--version` : Shows `(banner)`.
- `-K` or `--back` : MrEd, Mac OS X only; leave application in the background.
- `-V` `--no-yield` : Skips final `(yield 'wait)` action, which normally waits until all frames are closed, etc. in the main eventspace before exiting.

• *Configuration options*:

- `-c` or `--no-compiled` : Disables loading of compiled byte-code `".zo"` files, by initializing `current-compiled-file-paths` to `null`.
- `-q` or `--no-init-file` : Skips loading `(find-system-path 'init-file)` for `-i`/`--repl` or `-z`/`--text-repl`.
- `-I` ⟨*path*⟩ : Sets `(lib "`⟨*path*⟩`")` as the path to `require` to initialize the namespace, unless namespace initialization is disabled.
- `-X` ⟨*dir*⟩ or `--collects` ⟨*dir*⟩ : Sets ⟨*dir*⟩ as the path to the main collection of libraries by making `(find-system-path 'collects-dir)` produce ⟨*dir*⟩.
- `-S` ⟨*dir*⟩ or `--search` ⟨*dir*⟩ : Adds ⟨*dir*⟩ to the library collection search path. The `"dir"` is added after a user-specific directory, if any, and before the main collection directory.
- `-U` or `--no-user-path` : Omits user-psecific paths in the search for collections, C libraries, etc. by initializing the `use-user-specific-search-paths` parameter to `#f`.

- `-N` ⟨*file*⟩ or `--name` ⟨*file*⟩ : sets the name of the executable as reported by `(find-system-path 'run-file)` to ⟨*file*⟩.
- `-j` or `--no-jit` : Disables the native-code just-in-time compiler by setting the `eval-jit-enabled` parameter to `#f`.
- `-d` or `--no-delay` : Disables on-demand parsing of compiled code and syntax objects by setting the `read-on-demand-source` parameter to `#f`.
- `-b` or `--binary` : Requests binary mode, instead of text mode, for the process's input, out, and error ports. This flag currently has no effect, because binary mode is always used.

- Meta options:
  - `--` : No argument following this flag is itself used as a flag.
  - `-h` or `--help` : Shows information about the command-line flags and start-up process and exits, ignoring all other flags.

If at least one command-line argument is provided, and if the first one after any configuration option is not a flag, then a `-u`/`----require-script` flag is implicitly added before the first non-flag argument.

If no command-line arguments are supplied other than configuration options, then the `-i`/`----repl` flag is effectively added.

For MrEd under X11, the follow flags are recognized when they appear at the beginning of the command line, and they count as configuration options (i.e., they do not disable the read-eval-print loop or prevent the insertion of `-u`/`--require-script`):

- `-display` ⟨*display*⟩ : Sets the X11 display to use.
- `-geometry` ⟨*arg*⟩, `-bg` ⟨*arg*⟩, `-background` ⟨*arg*⟩, `-fg` ⟨*arg*⟩, `-foreground` ⟨*arg*⟩, `-fn` ⟨*arg*⟩, `-font` ⟨*arg*⟩, `-iconic`, `-name` ⟨*arg*⟩, `-rv`, `-reverse`, `+rv`, `-selectionTimeout` ⟨*arg*⟩, `-synchronous`, `-title` ⟨*arg*⟩, `-xnllanguage` ⟨*arg*⟩, or `-xrm` ⟨*arg*⟩ : Standard X11 arguments that are mostly ignored but accepted for compatibility with other X11 programs. The `-synchronous` and `-xrm` flags behave in the usual way.
- `-singleInstance` : If an existing MrEd is already running on the same X11 display, if it was started on a machine with the same hostname, and if it was started with the same name as reported by `(find-system-path 'run-file)`—possibly set with the `-N`/`--name` command-line argument—then all non-option command-line arguments are treated as filenames and sent to the existing MrEd instance via the application file handler (see `application-file-handler`).

Similarly, under Mac OS X, a leading switch starting with `-psn_` is treated as a special configuration option. It indicates that Finder started the application, so the current input, output, and error output are redirected to a GUI window.

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

`-ifve` ⟨*file*⟩ ⟨*expr*⟩

and

`-i -f` ⟨*file*⟩ `-v -e` ⟨*expr*⟩

are equivalent. If a collapsed `--` appears before other collapsed switches in the same collapsed set, it is implicitly moved to the end of the collapsed set.

Extra arguments following the last option are available from the `current-command-line-arguments` parameter.

16.2 Libraries and Collections

A *library* is `module` declaration for use by multiple programs. Scheme further groups libraries into *collections* that can be easily distributed and easily added to a local MzScheme installation.

Some collections are distributed via PLaneT. Such collections are referenced through a `planet` module path (see `require`) and are downloaded by Scheme on demand.

Other collections are distributed with PLT Scheme, in which case each collection is a directory that is located in a `"collects"` directory relative to the `mzscheme`. A collection can also be installed in a user-specific directory. More generally, the search path for installed collections can be configured through the `current-library-collection-paths` parameter. In all of these cases, the collections are referenced through `lib` paths (see `require`).

For example, the following module uses the `"getinfo.ss"` library module from the `"setup"` collection, and the `"cards.ss"` library module from the `"games"` collection's `"cards"` subcollection:

```
#lang scheme
(require (lib "setup/getinfo.ss")
         (lib "games/cards/cards.ss"))
....
```

In general, the *`rel-string`* in `(lib` *`rel-string`*`)` consists of one or more path elements that name collections, and then a final path element that names a library file; the path elements are separated by `/`. If the final element has no file suffix, then `/main.ss` is implicitly appended to the path.

The translation of a `"planet"` or `"lib"` path to a `module` declaration is determined by the module name resolver, as specified by the `current-module-name-resolver` parameter.

For the default module name resolver, The search path for collections is determined by the `current-library-collection-paths` parameter. The list of paths in `current-library-collection-paths` is searched from first to last to locate the first collection in a *`rel-string`*. To find a sub-collection, the enclosing collection is first found; if the sub-collection is not present in the found enclosing collection, then the search continues by looking for another instance of the enclosing collection, and so on. In other words, the directory tree for each element in the search path is spliced together with the directory trees of other path elements. (The "splicing" of tress applies only to directories; a file within a collection is found only within the first instance of the collection.)

The value of the `current-library-collection-paths` parameter is initialized in `mzscheme` to the result of `(find-library-collection-paths)`.

```
(find-library-collection-paths [extras]) → (listof path?)
  extras : (listof path-string?) = null
```

Produces a list of paths as follows:

- The path produced by `(build-path (find-system-path 'addon-dir) (version) "collects")` is the first element of the default collection path list, unless the value of the `use-user-specific-search-paths` parameter is `#f`.
- Extra directories provided in *`extras`* are included next, converted to complete paths relative to the executable.
- If the directory specified by `(find-system-path 'collects-dir)` is absolute, or if it is relative (to the executable) and it exists, then it is added to the end of the default collection path list.
- If the `PLTCOLLECTS` environment variable is defined, it is combined with the default list using `path-list-string->path-list`. If it is not defined, the default collection path list (as constructed by the first three bullets above) is used directly.

```
(collection-path collection ...+) → path?
  collection : string?
```

Returns the path to a directory containing the libraries of the collection indicated by *`col-`*

`lections`, where the second `collection` (if any) names a sub-collection, and so on. If the collection is not found, the `exn:fail:filesystem` exception is raised.

```
(current-library-collection-paths)
 → (listof (and/c path? complete-path?))
(current-library-collection-paths paths) → void?
  paths : (listof (and/c path? complete-path?))
```

Parameter that determines a list of complete directory paths for library collections used by `require`. See §16.2 "Libraries and Collections" for more information.

```
(use-user-specific-search-paths) → boolean?
(use-user-specific-search-paths on?) → void?
  on? : any/c
```

Parameter that determines whether user-specific paths, which are in the directory produced by `(find-system-path 'addon-dir)`, are included in search paths for collections and other files. For example, `find-library-collection-paths` omits the user-specific collection directory when this parameter's value is `#f`.

16.3 Interactive Help

```
(require scheme/help)
```

The `help` form documented in this section is provided by the `scheme/help` and `scheme/init` libraries, which means that it is available when `mzscheme` is started with no command-line arguments. It is not provided by `scheme` or `scheme/base`.

```
help
(help id)
(help id #:from module-path)
(help #:search datum ...)
```

Searches the documentation, and opens a web browser (using the user's selected browser) to display the results. See `net/sendurl` for information on how the user's browser is launched.

A simple `help` or `(help)` form opens this page.

A `(help id)` form looks for documentation specific to the current binding of `id`. For example,

```
(require net/url)
(help url->string)
```

opens a web browser to show the documentation for `url->string` from the `net/url` library.

For the purposes of `help`, a `for-label` require introduces a binding without actually executing the `net/url` library—for cases when you want to check documentation, but cannot or do not want to run the providing module.

```
(require scheme/gui) ; does not work in mzscheme
(require (for-label scheme/gui)) ; ok in mzscheme
(help frame%)
```

If *id* has no for-label and normal binding, then `help` lists all libraries that are known to export a binding for *id*.

The `(help id #:from module-path)` variant is similar to `(help id)`, but using only the exports of *module-path*. (The *module-path* module is required `for-label` in a temporary namespace.)

```
(help frame% #:from scheme/gui) ; equivalent to the above
```

The `(help #:search datum ...)` form performs a general search. Searching uses strings; each string *datum* is used as-is, and any other form of *datum* is converted to a string using `display`. No *datum* is evaluated as an expression.

For example,

```
(help #:search "web browser" firefox)
```

searches the documentation index for references that include the phrase “web browser” or “firefox.”

16.4 Interactive Module Loading

```
(require scheme/enter)
```

The bindings documented in this section are provided by the `scheme/enter` and `scheme` libraries, but not `scheme/base`.

```
(enter! module-path)
(enter! #f)
```

Intended for use in a REPL, such as when `mzscheme` is started in interactive mode. When a *module-path* is provided (in the same sense as for `require`), the corresponding module is loaded or invoked, and the current namespace is changed to the body of the module via `module->namespace`. When `#f` is provided, then the current namespace is restored to the original one.

If invoking *module-path* requires loading any files, then modification dates of the files are recorded. If the file is modified, then a later `enter!` re-loads the module from source; see also §1.1.10.2 "Module Re-declarations". Similarly if a later `enter!` transitively `requires` a modified module, then the required module is re-loaded. Re-loading support works only for modules that are first loaded (either directly or indirectly through transitive `requires`) via `enter!`.

After switching namespaces to the designated module, `enter!` automatically requires `scheme/enter` into the namespace, so that `enter!` can be used to switch namespaces again.

When it loads or re-loads a module from a file, `enter!` prints a message to `(current-error-port)`.

Bibliography

[Danvy90] Olivier Danvy and Andre Filinski, “Abstracting Control,” LISP and Functional Programming, 1990.
[Felleisen88] Matthias Felleisen, Mitch Wand, Dan Friedman, and Bruce Duba, “Abstract Continuations: A Mathematica
[Friedman95] Daniel P. Friedman, C. T. Haynes, and R. Kent Dybvig, “Exception system proposal,” web page, 1995. ht
[Gasbichler02] Martin Gasbichler and Michael Sperber, “Processes vs. User-Level Threads in Scsh,” Workshop on Schem
[Gunter95] Carl Gunter, Didier Remy, and Jon Rieke, “A Generalization of Exceptions and Control in ML-like Langua
[Hieb90] Robert Hieb and R. Kent Dybvig, “Continuations and Concurrency,” Principles and Practice of Parallel Pro
[L’Ecuyer02] Pierre L’Ecuyer, Richard Simard, E. Jack Chen, and W. David Kelton, “An Object-Oriented Random-Num
[Queinnec91] Queinnec and Serpette, “A Dynamic Extent Control Operator for Partial Continuations,” Principles of Prog
[Shan04] Ken Shan, “Shift to Control,” Workshop on Scheme and Functional Programming, 2004.
[Sperber07] Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten (editors), “The Revised^6 Report
[Sitaram90] Dorai Sitaram, “Control Delimiters and Their Hierarchies,” *Lisp and Symbolic Computation*, 1990.
[Sitaram93] Dorai Sitaram, “Handling Control,” Programming Language Design and Implementation, 1993.
[SRFI-42] Sebastian Egner, “SRFI-42: Eager Comprehensions,” SRFI, 2003. `http://srfi.schemers.org/srfi-`

Index