PLT MzLib: Libraries Manual

PLT (scheme@plt-scheme.org)

350
Released June 2006

Copyright notice

Send us your Web links

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

Thanks

Contributors to MzLib include Dorai Sitaram, Bruce Hauman, Jens Axel Søgaard, Gann Bierner, and Kurt Howard (working from Steve Moshier's Cephes library). Publicly available packages have been assimilated from others, including Andrew Wright (`match`) and Marc Feeley (original pretty-printing implementation).

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

This manual was typeset on June 18, 2006.

Contents

1. MzLib

The MzLib collection consists of several libraries, each of which provides a set of procedures and syntax.

To use a MzLib library, either at the top-level or within a module, import it with

```
(require (lib libname))
```

For example, to use the **list.ss** library:

```
(require (lib "list.ss"))
```

The MzLib collection provides the following libraries:

- **async-channel.ss** — buffered channels
- **awk.ss** — AWK-like syntax
- **class.ss** — object system
- **cm.ss** — compilation manager
- **cm-accomplice.ss** — compilation support hook syntax transformers
- **cmdline.ss** — command-line parsing
- **cml.ss** — Concurrent ML compatibility
- **compat.ss** — compatibility procedures and syntax
- **compile.ss** — bytecode compilation
- **contract.ss** — programming by contract
- **date.ss** — date-processing procedures
- **deflate.ss** — `gzip`
- **defmacro.ss** — `define-macro` and `defmacro`
- **etc.ss** — semi-standard procedures and syntax
- **file.ss** — file-processing procedures
- **include.ss** — textual source inclusion
- **inflate.ss** — `gunzip`
- **integer-set.ss** — sets of exact integers
- **kw.ss** — keyword argument procedures
- **list.ss** — list-processing procedures
- **match.ss** — pattern matching (backwards compatible library)
- **math.ss** — arithmetic procedures and constants
- **md5.ss** — MD5 message-digest algorithm
- **os.ss** — system utilities
- **package.ss** — local-definition scope control
- **pconvert.ss** — print values as expressions
- **pconvert-prop.ss** — property to adjust printed form
- **plt-match.ss** — pattern matching (improved syntax for patterns)
- **port.ss** — port utilities
- **pregexp.ss** – Perl-style regular expressions
- **pretty.ss** — pretty-printer

- **restart.ss** — stand-alone MzScheme emulator
- **sendevent.ss** — AppleEvents
- **serialize.ss** — serialization of data
- **shared.ss** — graph constructor syntax
- **string.ss** — string-processing procedures
- **struct.ss** — structure unilities
- **stxparam.ss** — support for parameter-like syntax bindings
- **surrogate.ss** — a proxy-like design pattern
- **tar.ss** — create `tar` files
- **thread.ss** — thread utilities
- **trace.ss** — procedure tracing
- **traceld.ss** — file-load tracing
- **transcr.ss** — transcripts
- **unit.ss** — component system
- **unitsig.ss** — component system with signatures
- **zip.ss** — create `zip` files

2. async-channel.ss: Buffered Asynchronous Channels

To load: `(require (lib "async-channel.ss"))`

This library implemented buffered asynchronous channels to complement MzScheme's synchronous channels (see §7.5 in *PLT MzScheme: Language Manual*).

`(make-async-channel [`*`limit-k`*`])` PROCEDURE

Returns an asynchronous channel with a buffer limit of *`limit-k`* items. A get operation blocks when the channel is empty, and a put operation blocks when the channel has *`limit-k`* items already.

If *`limit-k`* is `#f` (the default), the channel buffer has no limited (so a put never blocks). Otherwise, *`limit-k`* must be a positive exact integer.

The asynchronous channel value can be used directly with `sync` (see §7.7 in *PLT MzScheme: Language Manual*). The channel blocks until `async-channel-get` would return a value, and the unblock result is the received value.

`(async-channel-get `*`async-channel`*`)` PROCEDURE

Blocks until at least one value is available in *`async-channel`*, and then returns the first of the values that was put into *`async-channel`*.

`(async-channel-try-get `*`async-channel`*`)` PROCEDURE

If at least one value is immediately available in *`async-channel`*, returns the first of the values that was put into *`async-channel`*. If *`async-channel`* is empty, the result is `#f`.

`(async-channel-put `*`async-channel v`*`)` PROCEDURE

Puts *`v`* into *`async-channel`*, blocking if *`async-channel`*'s buffer is full until space is available. The result is void.

`(async-channel-put-evt `*`async-channel v`*`)` PROCEDURE

Returns a synchronizable event that is blocked while `(async-channel-put `*`async-channel v`*`)` would block. The unblock result is the event itself. See also §7.7 in *PLT MzScheme: Language Manual*.

3. awk.ss: Awk-like Syntax

To load: `(require (lib "awk.ss"))`

This library defines the `awk` macro from Scsh:

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

counter-variable/optional is either empty or
  variable

continue-variable/optional is either empty or
  variable

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

test is one of
  integer
  regexp-str
  expr

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

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

4. class.ss: Classes and Objects

To load: `(require (lib "class.ss"))`

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.

An *object* is a collection of bindings for fields that are instantiated according to a class description.

The object 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 first-class Scheme values. However, a MzScheme class or interface is not a MzScheme object (i.e., there are no “meta-classes” or “meta-interfaces”).

4.1 Object Example

The following example conveys the object system's basic style.

```
(define stack<%> (interface () push! pop! none?))

(define stack%
  (class* object% (stack<%>)
    ; Declare public methods that can be overridden:
    (public push! pop! none?)
    ; Declare a public method that can be augmented, only:
    (pubment print-name)

     (define stack null)        ; A private field
     (init-field (name 'stack)) ; A public field

     ; Method implementations:
     (define (push! v)
       (set! stack (cons v stack)))
     (define (pop!)
       (let ([v (car stack)])
         (set! stack (cdr stack))
         v))
     (define (none?)
       (null? stack))
     (define (print-name)
       (display name)
       (inner (void) print-name) ; Let subclass print more
       (newline))

     ; Call superclass initializer:
     (super-new)))

(define fancy-stack%
  (class stack%
    ; Declare override
    (override push!)
    ; Implement override:
    (define (push! v)
      (super push! (cons 'fancy v)))

     ; Add inherited field to local environment
     (inherit-field name)

     ; Declare augment
     (augment print-name)
     ; Implement augment
     (define (print-name)
       (when (equal? name 'Bob)
         (display ", Esq."))
       (inner (void) print-name))

     (super-new)))
```

```
(define double-stack%
  (class stack%
    (inherit push!)

     (public double-push!)
     (define (double-push! v) (push! v) (push! v))

     ; Always supply name
     (super-new (name 'double-stack))))

(define-values (make-safe-stack-class is-safe-stack?)
  (let ([safe-stack<%> (interface (stack<%>))])
    (values
     (lambda (super%)
       (class* super% (safe-stack<%>)
         (inherit none?)
         (override pop!)
         (define (pop!)
           (if (none?)
               #f
               (super pop!)))
         (super-new)))
     (lambda (obj)
       (is-a? obj safe-stack<%>)))))

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

The interface *stack<%>*[1] defines the ever-popular stack interface with the methods *push!*, *pop!*, and *none?*. Since it has no superinterfaces, the only derivation requirement of *stack<%>* is that its classes are derived from the built-in empty class, `object%`. The class *stack%*[2] is derived from `object%` and implements the *stack<%>* interface. Three additional classes are derived from the basic *stack%* implementation:

- The class *fancy-stack%* defines a stack that overrides *push!* to tag each item as fancy. It also augments *print-name* to add an "Esq." suffix if the stack's name is `'Bob`.
- The class *double-stack%* extends the functionality *stack%* with a new method, `double-push!`. It also supplies a specific `name` to *stack%*.
- The class *safe-stack%* overrides the *pop!* method of *stack%*, ensuring that `#f` is returned whenever the stack is empty.

In each derived class, the `(super-new ...)` form causes the superclass portion of the object to be initialized, including the initialization of its fields.

The creation of *safe-stack%* illustrates the use of classes as first-class values. Applying *make-safe-stack-class* to *fancy-stack%* or *double-stack%* — indeed, *any* class with *push*, *pop!*, and *none?* methods — creates a "safe" version of the class. A stack object can be recognized as a safe stack by testing it with *is-safe-stack?*; this predicate returns `#t` only for instances of a class created with *make-safe-stack-class* (because only those classes implement the *safe-stack<%>* interface).

In each of the example classes, the field *name* contains the name of the class. The *name* instance variable is introduced as a new instance variable in *stack%*, and it is declared there with the `init-field` keyword, which means that

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

an instantiation of the class can specify the initial value, but it defaults to `'stack`. The *double-stack%* class provides *name* when initializing the *stack%* part of the object, so a name cannot be supplied when instantiating *double-stack%*. When the *print-name* method of an object from *double-stack%* is invoked, the name printed to the screen is always "double-stack".

While all of *fancy-stack%*, *double-stack%*, and *safe-stack%* inherit the *push!* method of *stack%*, it is declared with `inherit` only in *double-stack%*; new declarations in *fancy-stack%* and *safe-stack%* do not need to refer to *push!*, so the inheritance does not need to be declared. Similarly, only *safe-stack%* needs to declare `(inherit none?)`.

The *fancy-stack%* class overrides *pop!* to extend the implementation of *pop!*. The new definition of *pop!* must accesses the original *pop!* method that is defined in *stack%* through the `super` form.

The *stack%* class declares its *print-name* method using `pubment`, which means that the method is public, but it can only be augmented in subclasses, and not overridden. The implementation of *print-name* uses `inner` to execute a subclass-supplied augmenting method. If no such augmenting method is available, the `(void)` expression is evaluated, instead. The *fancy-stack%* classes uses `augment` to declare an augmentation of *print-name*, and also uses `inner` to allow further augmenting in later subclasses.

The `instantiate` form, the `new` form, and the `make-object` procedure all create an object from a class. The `instantiate` form supports initialization arguments by both position and name, the `new` form only supports by name initialization arguments, and `make-object` supports initialization arguments by position only. The following examples create objects using the classes above:

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

The `send` form calls a method on an object, finding the method by name. The following example uses the objects created above:

```
(send stack push! fred)
(send stack push! double-stack)
(let loop ()
  (if (not (send stack none?))
    (begin
      (send (send stack pop!) print-name)
      (loop))))
```

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

4.2 Creating Interfaces

The `interface` form creates a new interface:

```
(interface (super-interface-expr ···) identifier ···)
```

All of the *identifier*s must be 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 *`identifier`*s, plus all identifiers from the superinterfaces. Duplicate identifier names among the superinterfaces are ignored, but if a superinterface contains one of the *`identifier`*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.

4.3 Creating Classes

The built-in class `object%` 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%`.

The `class*` form creates a new class:

```
(class* superclass-expr (interface-expr ···)
  class-clause
  ···)

class-clause is one of
  (inspect inspector-expr)
  (init init-declaration ···)
  (init-field init-declaration ···)
  (field field-declaration ···)
  (inherit-field optionally-renamed-id ···)
  (init-rest id)
  (init-rest)
  (public optionally-renamed-id ···)
  (pubment optionally-renamed-id ···)
  (public-final optionally-renamed-id ···)
  (override optionally-renamed-id ···)
  (overment optionally-renamed-id ···)
  (override-final optionally-renamed-id ···)
  (augment optionally-renamed-id ···)
  (augride optionally-renamed-id ···)
  (augment-final optionally-renamed-id ···)
  (private id ···)
  (inherit optionally-renamed-id ···)
  (rename-super renamed-id ···)
  (rename-inner renamed-id ···)
  method-definition
  definition
  expr
  (begin class-clause ···)

init-declaration is one of
  identifier
  (optionally-renamed-id)
```

```
  (optionally-renamed-id default-value-expr)

field-declaration is
  (optionally-renamed-id default-value-expr)

optionally-renamed-id is one of
  identifier
  renamed-id

renamed-id is
  (internal-id external-id)

method-definition is
  (define-values (identifier) method-procedure)

method-procedure is
  (lambda formals expr ···¹)
  (case-lambda (formals expr ···¹) ···)
  (let-values (((identifier) method-procedure) ···) method-procedure)
  (letrec-values (((identifier) method-procedure) ···) method-procedure)
  (let-values (((identifier) method-procedure) ···¹) identifier)
  (letrec-values (((identifier) method-procedure) ···¹) identifier)
```

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 §4.5 in *PLT MzScheme: Language Manual*) 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 §4.5 in *PLT MzScheme: Language Manual*). 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 *identifier* or *optionally-renamed-id* 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 §4.4).

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 §4.4.

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 §4.4); `super` is available for calling superclass methods (see §4.3.3.1); and `inner` is available for calling subclass augmentations of methods (see §4.3.3.1).

The `public`, `override`, `augment`, `pubment`, `overment`, `augride`, `public-final`, `override-final`, `augment-final`, `private`, `inherit`, `rename-super`, `rename-inner this`, `super`, `inner`, `super-instantiate`, `super-make-object`, and `super-new` keywords are all exported by **class.ss** as syntactic forms that raise an error when used outside of a class declaration.

The `class` form is like `class*`, but omits the *interface-expr*s, for the case that none are needed:

```
(class superclass-expr
 class-clause
 ···)
```

The `public*`, `pubment*`, `public-final*`, `override*`, `overment*`, `override-final*`, `augment*`, `augride*`, `augment-final*`, and `private*` forms abbreviate a `public`, etc. declaration and a sequence of definitions:

```
 (public* (name expr) ···)
=expands=>
 (begin
  (public name ···)
  (define name expr) ···)

etc.
```

The `define/public`, `define/pubment`, `define/public-final`, `define/override`, `define/overment`, `define/override-final`, `define/augment`, `define/augride`, `define/augment-final`, and `define/private` forms similarly abbreviate a `public`, etc. declaration with a definition:

```
 (define/public name expr)
=expands=>
 (begin
  (public name)
  (define name expr))

 (define/public (header . formals) expr)
=expands=>
 (begin
  (public name)
  (define (header . formals) expr))

etc.
```

4.3.1 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 §4.4.

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 §4.4.)

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 §4.4. 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 §4.4 for further details.

See also §4.3.3.3 for information about internal and external names.

4.3.2 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 §4.5). 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 §4.3.1 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 §3.1 in *PLT MzScheme: Language Manual*). The fields of a class are initialized at the same time that the class's initialization expressions are evaluated; see §4.4 for more information.

See also §4.3.3.3 for information about internal and external names.

4.3.3 Methods

4.3.3.1 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 §4.3, 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 *`identifier`* must be one of the *`identifier`*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:

```
(super identifier arg-expr ···)
```

Such a `super` call always accesses the superclass method, independent of whether the method is overridden again in subclasses.

When a method is declared with `pubment`, `augment`, or `overment`, then a subclass augmenting method can be called using the `inner` form:

```
(inner default-expr identifier arg-expr ···)
```

If the object's class does not supply an augmenting method, then *`default-expr`* is evaluated, and the *`arg-expr`*s are not evaluated. Otherwise, the augmenting method is called with the *`arg-expr`* 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. (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.)

4.3.3.2 Inherited and Superclass Methods

Each `inherit`, `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 superclass and augmenting methods are typically accessed through `super` and `inner` in a class that also declares the methods.

Methods declared with `inherit` access overriding declarations, if any, at run time. Methods 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

```
(identifier (lambda () default-expr) arg-expr ···)
```

so that a *`default-expr`* is available to evaluate when no augmenting method is available. In such a form, `lambda` is a keyword to separate the *`default-expr`* from the *`arg-expr`*. When an augmenting method is available, it receives the results of the *`arg-expr`*s as arguments.

Methods that are present in the superclass but not declared with `inherit` 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`, `rename-super`, or `rename-inner` is not present in the superclass, the `exn:fail:object` exception is raised when the class expression is evaluated.

4.3.3.3 Internal and External Names

Each method declared with `public`, `override`, `augment`, `pubment`, `overment`, `augride`, `public-final`, `override-final`, `augment-final`, `inherit`, `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 §4.5). If a single *`identifier`* 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` form introduces a set of scoped external names:

```
(define-local-member-name identifier ···)
```

Unless it appears as the top-level definition, this form binds each *`identifier`* so that, within the scope of the definition, each use of each *`identifier`* 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 *`identifier`*s are accessible only in the scope of the `define-local-member-name` declaration. As a top-level definition, `define-local-member-name` binds *`identifier`* to its symbolic form.

The binding introduced by `define-local-member-name` is a syntax binding that can be exported and imported with modules (see §5 in *PLT MzScheme: Language Manual*). 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 (see §4.8) does not expose hidden names.

Example:

```
(define o (let ()
            (define-local-member-name m)
            (define c% (class object%
                         (define/public (m) 10)
                         (super-new)))
            (define o (new c%))

           (send o m) ; ⇒ 10
           o))

(send o m) ; ⇒ error: no method m
```

The `define-local-name` form maps a single external name to an external name that is determined by an expression:

```
(define-member-name identifier key-expr)
```

The value of *`key-expr`* must be the result of either a `member-name-key` expression,

```
(member-name-key identifier)
```

or `(generate-member-key)`. The latter produces a hidden name, just like the binding for `define-local-member-name`. The `(member-name-key identifier)` form produces a representation of the external name for *`identifier`* in the environment of the `member-name-key` expression.

Example:

```
(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) ; ⇒ error: no method m
(send (new (make-c% (member-name-key p))) p) ; ⇒ 10

(define (fresh-c%)
  (let ([key (generate-member-name)])
    (values (make-c% key) key)))

(define-values (fc% key) (fresh-c%))
(send (new fc%) m) ; ⇒ error: no method m
(let ()
  (define-member-name p key)
  (send (new fc%) p)) ; ⇒ 10
```

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.

4.4 Creating Objects

The `make-object` procedure creates a new object with by-position initialization arguments:

```
(make-object class init-v ···)
```

An instance of *`class`* is created, and the *`init-v`*s are passed as initialization arguments, bound to the initialization variables of *`class`* for the newly created object as described in §4.3.1. If *`class`* is not a class, the `exn:fail:contract` exception is raised.

The `new` form creates a new object with by-name initialization arguments:

```
(new class-expr (identifier by-name-expr) ···)
```

An instance of the value of *`class-expr`* is created, and the value of each *`by-name-expr`* is provided as a by-name argument for the corresponding *`identifier`*.

The `instantiate` form creates a new object with both by-position and by-name initialization arguments:

```
(instantiate class-expr (by-pos-expr ···) (identifier by-name-expr) ···)
```

An instance of the value of *`class-expr`* is created, 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 *`identifier`*.

All fields in the newly created object are initially bound to the special undefined value (see §3.1 in *PLT MzScheme: Language Manual*). 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-instantiate` form:

```
(super-instantiate (by-position-super-init-expr ···) (identifier by-name-super-init-expr
```

or by using the procedure produced by the `super-make-object` form:

```
(super-make-object super-init-v ···)
```

or by using `super-new` form:

```
(super-new (identifier by-name-super-init-expr ···) ···)
```

The *`by-position-super-init-expr`*s, *`by-name-super-init-exp`*s, and *`super-init-v`*s are mapped to initialization variables in the same way as for `instantiate`, `make-object`, and `new`.

By-name initialization arguments to a class that have no matching initialization variable are implicitly added as by-name arguments to a `super-instantiate`, `super-make-object`, or `super-new` 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.

Fields inherited from a superclass will not be 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++).

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

4.5 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!`.

4.5.1 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 form

```
(method-id arg-expr ··· . arg-list-expr)
(super method-id arg-expr ··· . arg-list-expr)
(inner default-expr method-id arg-expr ··· . arg-list-expr)
```

which calls the method in a way analogous to `(apply method-id arg-expr ··· arg-list-expr)`. The *arg-list-expr* must not be a parenthesized expression, otherwise the dot and the parentheses will cancel each other.

Methods are called from outside a class with the `send` and `send/apply` forms:

```
(send obj-expr method-name arg-expr ···)
(send obj-expr method-name arg-expr ··· . arg-list-expr)
(send/apply obj-expr method-name arg-expr ··· arg-list-expr)
```

where the last two forms apply the method to a list of argument values; in the second form, *arg-list-expr* cannot be a parenthesized expression. For any `send` or `send/apply`, if *obj-expr* does not produce an object, the `exn:fail:contract` exception is raised. If the object has no public method *method-name*, the `exn:fail:object` exception is raised.

The `send*` form calls multiple methods of an object in the specified order:

```
(send* obj-expr msg ···)

msg is one of
  (method-name arg-expr ···)
  (method-name arg-expr ··· . arg-list-expr)
```

where *arg-list-expr* is not a parenthesized expression.

Example:

```
(send* edit (begin-edit-sequence)
            (insert "Hello")
            (insert #\newline)
            (end-edit-sequence))
```

which 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))
```

The `with-method` form extracts a method from an object and binds a local name that can be applied directly (in the same way as declared methods within a class):

```
(with-method ((identifier (object-expr method-name)) ···)
  expr ···1)
```

Example:

```
(let ([s (new stack%)])
  (with-method ([push (s push!)]
                [pop (s pop!)])
    (push 10)
    (push 9)
    (pop)))
```

which is the same as

```
(let ([s (new stack%)])
  (send s push! 10)
  (send s push! 9)
  (send s pop!))
```

4.5.2 Fields

The `get-field` form,

```
(get-field identifier object-expr)
```

extracts the field named by *`identifier`* from the value of the *`object-expr`*.

The `field-bound?` form,

```
(field-bound? identifier object-expr)
```

produces `#t` if *`object-expr`* evaluates to an object that has a field named *`identifier`*, `#f` otherwise.

If you have access to the class of an object, the `class-field-accessor` and `class-field-mutator` forms provide efficient access to the object's fields.

- `(class-field-accessor` *`class-expr field-name`*`)` returns an accessor procedure that takes an instance of the class produced by *`class-expr`* and returns the value of the object's *`field-name`* field.
- `(class-field-mutator` *`class-expr field-name`*`)` returns a mutator procedure that takes an instance of the class produced by *`class-expr`* and a new value for the field, mutates the field in the object named by *`field-name`*, then returns void.

4.5.3 Generics

A *generic* can be used instead of a method name to avoid the cost of relocating a method by name within a class. The `make-generic` procedure and `generic` form create generics:

- `(make-generic` *`class-or-interface symbol`*`)` returns a generic that works on instances of *`class-or-interface`* (or an instance of a class/interface derived from *`class-or-interface`*) to call the method named by *`symbol`*.

  If *`class-or-interface`* does not contain a method with the (external and non-scoped) name *`symbol`*, the `exn:fail:object` exception is raised.
- `(generic` *`class-or-interface-expr name`*`)` is analogous to `(make-generic` *`class-or-interface-expr`* *`'name`*`)`, except that *`name`* can be a scoped method name declared by `define-local-member-name` (see §4.3.3.3).

A generic is applied with `send-generic`:

```
(send-generic obj-expr generic-expr arg-expr ···)
(send-generic obj-expr generic-expr arg-expr ··· . arg-list-expr)
```

where the value of *`obj-expr`* is an object and the value of *`generic-expr`* is a generic.

4.6 Mixins

A mixin is a class parameterization modeled on a paper published by Flatt, Felleisen, and Krishnamurthi, available at http://www.ccs.neu.edu/scheme/pubs/#popl98-fkf.

The implementation of these mixins in MzScheme is with the combination of lambda and class. This macro simplifies the checking and implementation of these mixins. Its syntax is very similar to the syntax for class∗. The shape of a mixin is:

```
(mixin (interface-expr ...) (interface-expr ...)
  class-clause ...)
```

This macro expands into a procedure that accepts a class. The argument passed to this procedure must match the interfaces of the first *interface-expr*s expressions. The procedure returns a class that is derived from its argument. This result class must match the interfaces specified in the second *interface-expr*s section; it has clauses specified by *instance-variable-clause*s. The syntax of the *initialization-variables* and *instance-variable-clause* are exactly the same as class∗/names.

The mixin macro does some checking to be sure that variables that the *instance-variable-clause*s refer to in their super class are in the interfaces. That checking and the checking that the input class matches the declared interfaces aside, the mixin macro's expansion is something like this:

```
(mixin (i<%> ...) (j<%> ...)
  class-clause ...)
=
(lambda (%)
  (class* % (j<%> ...)
    class-clause ...))
```

The *i*<%> interfaces do not appear in the output because they are only used for the error checking and are discarded by the time the class is created.

4.7 Object Serialization

The define-serializable-class and define-serializable-class∗ forms define classes whose instances are serializable using serialize (see §38).

```
(define-serializable-class class-id superclass-expr
  class-clause
  ···)

(define-serializable-class* class-id superclass-expr (interface-expr ···)
  class-clause
  ···)
```

These 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:*class-id* is also defined; if the definition is within a module, then the latter is provided from the module. The *superclass-expr*, *interface-expr*s, and *class-clause*s are as for class and class∗ (see §4.3).

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). The `externalizable<%>` interface includes only the `externalize` and `internalize` methods.

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.

4.8 Object, Class, and Interface Utilities

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

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

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

`(object=? `*`object`*` [`*`object`*`])` determines if two objects are the same object, or not (uses `eq?`, but also works properly with contracts).

`(object->vector `*`object`*` [`*`opaque-v`*`])` returns a vector representing *`object`* that shows its inspectable fields, analogous to `struct->vector` (see §4.8 in *PLT MzScheme: Language Manual*).

`(class->interface `*`class`*`)` returns the interface implicitly defined by *`class`* (see the overview at the beginning of Chapter 4).

`(object-interface `*`object`*`)` returns the interface implicitly defined by the class of *`object`*.

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

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

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

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

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

`(method-in-interface? `*`symbol interface`*`)` returns `#t` if *`interface`* (or any of its ancestor interfaces) includes a member with the name *symbol*, `#f` otherwise.

`(interface->method-names` *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 symbol k*`)` returns `#t` if *object* has a method named *symbol* that accepts *k* arguments, `#f` otherwise.

`(field-names` *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*`)` returns two values, analogous to the return values of `struct-info` (see §4.5 in *PLT MzScheme: Language Manual*):

- *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*`)` returns seven values, analogous to the return values of `struct-type-info` (see §4.5 in *PLT MzScheme: Language Manual*):

- *name-symbol*: the class's name as a symbol;
- *field-k*: 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-proc*: 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-k* (exclusive);
- *field-mutator-proc*: 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-k* (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.

4.9 Expanding to a Class Declaration

The `class/derived` form is like `class*`, but it 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`.

```
(class/derived original-datum
  (name-id super-expr (interface-expr ...) deserialize-id-expr)
  class-clause
  ···)
```

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*` (see §4.3).

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. class100.ss: Version-100-Style Classes

To load: `(require (lib "class100.ss"))`

The `class100` and `class100*` forms provide a syntax close to that of `class` and `class*` in MzScheme versions 100 through 103, but with the semantics of the current **class.ss** system (see Chapter 4). For a class defined with `class100`, keyword-based initialization arguments can be propagated to the superclass, but by-position arguments are not (i.e., the expansion of `class100` to `class` always includes an `init-rest` clause).

The `class100` form uses keywords (e.g., `public`) that are defined by the **class** library, so typically **class.ss** must be imported into any context that imports **class100.ss**.

The `class100*` form creates a new class:

```
(class100* superclass-expr (interface-expr ···) initialization-ids
  class100-clause
  ···)

initialization-ids is one of
  variable
  (variable ··· variable-with-default ···)
  (variable ··· variable-with-default ··· . variable)

variable-with-default is
  (variable default-value-expr)

class100-clause is one of
  (sequence expr ···)
  (public public-method-declaration ···)
  (override public-method-declaration ···)
  (augment public-method-declaration ···)
  (pubment public-method-declaration ···)
  (overment public-method-declaration ···)
  (augride public-method-declaration ···)
  (private private-method-declaration ···)
  (private-field private-var-declaration ···)
  (inherit inherit-method-declaration ···)
  (rename rename-method-declaration ···)

public-method-declaration is one of
  ((internal-id external-id) method-procedure)
  (identifier method-procedure)

private-method-declaration is one of
  (identifier method-procedure)

private-var-declaration is one of
```

```
  (identifier initial-value-expr)
  (identifier)
  identifier

inherit-method-declaration is one of
  identifier
  (internal-instance-id external-inherited-id)

rename-method-declaration is
  (internal-id external-id)
```

In *local-names*, if *super-instantiate-id* is not provided, the `instantiate`-like superclass initialization form will not be available in the `class100*/names` body.

The `class100` macro omits the *interface-expr*s:

```
(class100 superclass-expr initialization-ids
   class100-clause
   ···)
```

`(class100-asi` *superclass instance-id-clause* `···)` SYNTAX

Like `class100`, but all initialization arguments are automatically passed on to the superclass initialization procedure by position.

`(class100*-asi` *superclass interfaces instance-id-clause* `···)` SYNTAX

Like `class100*`, but all initialization arguments are automatically passed on to the superclass initialization procedure by position.

`(super-init` *init-arg-expr* `···)` SYNTAX

An alias for `super-make-object` in **class.ss**.

6. cm.ss: Compilation Manager

To load: `(require (lib "cm.ss"))`

`(make-compilation-manager-load/use-compiled-handler)` PROCEDURE

Returns a procedure suitable as a value for the `current-load/use-compiled` parameter (see §7.9.1.6 in *PLT MzScheme: Language Manual*). The returned procedure passes it arguments on to the current `current-load/use-compiled` procedure (i.e., the one installed when this procedure is called), but first it automatically compiles source files to a **.zo** file if

- the file is expected to contain a module (i.e., the second argument to the handler is a symbol);
- the value of each of `current-eval`, `current-load`, and `current-namespace` is the same as when `make-compilation-manager-load/use-compiled-handler` was called;
- the value of *`use-compiled-file-paths`* contains the first path that was present when `make-compilation-manager-l` was called;
- the value of `current-load/use-compiled` is the result of this procedure; and
- one of the following holds:
  - the source file is newer than the **.zo** file in the first sub-directory listed in *`use-compiled-file-paths`* (at the time that `make-compilation-manager-load/use-compiled-handler` was called)
  - no **.dep** file exists next to the **.zo** file;
  - the version recorded in the **.dep** file does not match the result of `(version)`;
  - one of the files listed in the **.dep** file has a **.zo** timestamp newer than the one recorded in the **.dep** file.

After the handler procedure compiles a **.zo** file, it creates a corresponding **.dep** file that lists the current version, plus the **.zo** timestamp for every file that is `require`d by the module in the compiled file (including `require-for-syntax`es and `require-for-template`s).

The handler caches timestamps when it checks **.dep** files, and the cache is maintained across calls to the same handler. The cache is not consulted to compare the immediate source file to its **.zo** file, which means that the caching behavior is consistent with the caching of the default module name resolver (see §5.4 in *PLT MzScheme: Language Manual*).

If *`use-compiled-file-paths`* contains an empty list when `make-compilation-manager-load/use-compiled-hand` is called, then `exn:fail:contract` exception is raised.

Do not install the result of `make-compilation-manager-load/use-compiled-handler` when the current namespace contains already-loaded versions of modules that may need to be recompiled — unless the already-loaded modules are never referenced by not-yet-loaded modules. References to already-loaded modules may produce compiled files with inconsistent timestamps and/or **.dep** files with incorrect information.

`(managed-compile-zo` *`file`*`)` PROCEDURE

Compiles the given module source file to a **.zo**, installing a compilation-manager handler while the file is compiled (so that required modules are also compiled), and creating a **.dep** file to record the timestamps of immediate files used to compile the source (i.e., files `require`d in the source, including `require-for-syntax`es and `require-for-template`s).

`(trust-existing-zos` *`on?`* [*`procedure`*]`)`

A parameter that is intended for use by **Setup PLT** when installing with pre-built **.zo** files. It causes a compilation-manager load/use-compiled handler to "touch" out-of-date **.zo** files instead of re-compiling from source.

`(make-caching-managed-compile-zo)` PROCEDURE

Returns a procedure that behaves like `managed-compile-zo`, but a cache of timestamp information is preserved across calls to the procedure.

`(manager-compile-notify-handler` [*`notify-proc`*]`)` PROCEDURE

A parameter for a procedure of one argument that is called whenever a compilation starts. The argument to the procedure is the file's path.

`(manager-trace-handler` [*`notify-proc`*]`)` PROCEDURE

A parameter for a procedure of one argument that is called to report compilation-manager actions, such as checking a file. The argument to the procedure is a string.

7. cm-accomplice.ss: Compilation Manager Hook for Syntax Transformers

To load: `(require (lib "cm-accomplice.ss"))`

`(register-external-file` *`file`*`)` PROCEDURE

Registers the complete path *`file`* with a compilation manager, if one is active. The compilation manager then records the path as contributing to the implementation of the module currently being compiled. Afterward, if the registered file is modified, the compilation manager will know to recompile the module.

The `include` macro, for example, calls this procedure with the path of an included file as it expands an `include` form.

8. cmdline.ss: Command-line Parsing

To load: `(require (lib "cmdline.ss"))`

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

Parses a command line according to the specification in the *`clause`*s. The *`program-name-expr`* should produce a string to be used as the program name for reporting errors when the command-line is ill-formed. The *`argv-expr`* must evaluate to a list or a vector of strings; typically, it is `(current-command-line-arguments)` or the cdr of an argument to a *`main`* procedure (when using '`-C`' to invoke a script).

The command-line is disassembled into flags (possibly with flag-specific arguments) followed by (non-flag) arguments. Command-line strings starting with "-" or "+" are parsed as flags, but arguments to flags are never parsed as flags, and integers and decimal numbers that start with "-" or "+" are not treated as flags. Non-flag arguments in the command-line must appear after all flags and the flags' arguments. No command-line string past the first non-flag argument is parsed as a flag. The built-in `--` flag signals the end of command-line flags; any command-line string past the `--` flag is parsed as a non-flag argument.

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

```
(multi flag-spec ···)
(once-each flag-spec ···)
(once-any flag-spec ···)
(final flag-spec ···)
(help-labels string ···)
(args arg-formals body-expr ···1)
(=> finish-proc-expr arg-help-expr help-proc-expr unknown-proc-expr)

flag-spec is one of
  (flags variable ··· help-str ···1 body-expr ···1)
  (flags => handler-expr help-expr)

flags is one of
  flag-str
  (flag-str ···1)

arg-formals is one of
  variable
  (variable ···)
  (variable ···1 . variable)
```

A `multi`, `once-each`, `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:

1. *`flags`* — a flag string, or a set of flag strings. If a set of flags is provided, all of the flags are equivalent. Each flag string must be of the form `"-`*`x`*`"` or `"+`*`x`*`"` for some character *`x`*, or `"--`*`x`*`"` or `"++`*`x`*`"` for some sequence of characters *`x`*. An *`x`* cannot contain only digits or digits plus a single decimal point, since simple (signed) numbers are not treated as flags. In addition, the flags `"--"`, `"-h"`, and `"--help"` are predefined and cannot be changed.
2. *`variable`*s — variables that are bound to the flag's arguments. The number of variables specified here determines how many arguments can be provided on the command line with the flag, and the names of these variables will appear in the help message describing the flag. The *`variable`*s are bound to string values in the *`body-expr`*s for handling the flag.
3. *`help-str`* — a string that describes the flag. This string is used in the help message generated by the handler for the built-in `-h` (or `--help`) flag. If multiple *`help-str`*s are provided, the rest are displayed on subsequent lines.
4. *`body-expr`*s — expressions that are evaluated when one of the *`flags`* appears on the command line. The flags are parsed left-to-right, and each sequence of *`body-expr`*s is evaluated as the corresponding flag is encountered. When the *`body-expr`*s are evaluated, the *`variable`*s are bound to the arguments provided for the flag on the command line.

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

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.

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

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

Example:

```
(command-line "compile" (current-command-line-arguments)
  (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 implies all optimizations of level 1"
                           (optimize-level 2)])
  (multi
    [("-l" "--link-flags") lf ; flag takes one argument
                           "Add a flag for the linker"
                           (link-flags (cons lf (link-flags)))])
  (args (filename) ; expects one command-line argument: a filename
   filename)) ; return a single filename to compile
```

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

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

When there are no more flags, the *finish-proc* procedure is called with a list of information accumulated for command-line flags (see below) and the remaining non-flag arguments from the command-line. The arity of the *finish-proc* procedure determines the number of non-flag arguments accepted and required from the command-line. For example, if *finish-proc* accepts either two or three arguments, then either one or two non-flag arguments must be provided on the command-line. The *finish-proc* procedure can have any arity (see §3.12.1 in *PLT MzScheme: Language Manual*) except 0 or a list of 0s (i.e., the procedure must at least accept one or more arguments).

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

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

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

strings.

2. A procedure to handle the flag and its arguments when one of the flags is found on the command line. The arity of this handler procedure determines the number of arguments consumed by the flag: the handler procedure is called with a flag string plus the next few arguments from the command line to match the arity of the handler procedure. The handler procedure must accept at least one argument to 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 (see §3.12.1 in *PLT MzScheme: Language Manual*).

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

3. A non-empty list 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 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 for the linker" "flag")]))
   (lambda (flag-accum file) file)  ; return a single filename to compile
   '("filename")) ; expects one command-line argument: a filename
```

9. cml.ss: Concurrent ML Compatibility

To load: `(require (lib "cml.ss"))`

This library defines a number of procedures that wrap MzScheme concurrency procedures. The wrapper procedures have names and interfaces that more closely match those of Concurrent ML.

`(spawn` *`thunk`*`)` PROCEDURE

Equivalent to `(thread/suspend-to-kill` *`thunk`*`)` (see §7.1 in *PLT MzScheme: Language Manual*).

`(channel)` procedure

Equivalent to `(make-channel)` (see §7.5 in *PLT MzScheme: Language Manual*).

`(channel-recv-evt` *`channel`*`)` PROCEDURE

Equivalent to *`channel`*.

`(channel-send-evt` *`channel v`*`)` PROCEDURE

Equivalent to `(channel-put-evt` *`channel v`*`)` (see §7.5 in *PLT MzScheme: Language Manual*).

`(thread-done-evt` *`thread`*`)` PROCEDURE

Equivalent to `(thread-dead-waitable` *`thread`*`)` (see §7.2 in *PLT MzScheme: Language Manual*).

`(current-time)` PROCEDURE

Equivalent to `(current-inexact-milliseconds)` (see §15.1 in *PLT MzScheme: Language Manual*).

`(time-evt` *`x`*`)` PROCEDURE

Equivalent to `(alarm-evt` *`x`*`)` (see §7.6 in *PLT MzScheme: Language Manual*).

10. compat.ss: Compatibility

To load: `(require (lib "compat.ss"))`

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

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

`(atom? v)` PROCEDURE

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

`(define-structure (name-identifier field-identifier ···))` SYNTAX

Like `define-struct`, except that the *name-identifier* is moved inside the parenthesis for fields. A second form of `define-structure`, below, supports initial-value expressions for fields.

`(define-structure (name-identifier field-identifier ···) ((init-field-identifier init-expr) ···))` SYNTAX

Like `define-struct`, except that the *name-identifier* is moved inside the parenthesis for fields, and additional fields can be specified with initial-value expressions.

The *init-field-identifier*s do not have corresponding arguments for the `make-`*name-identifier* constructor. Instead, the *init-field-identifier*'s *init-expr* is evaluated to obtain the field's value when the constructor is called. The *field-identifier*s are bound in *init-expr*s, but not the *init-field-identifier*s.

Example:

```
(define-structure (add left right) ([sum (+ left right)]))
(add-sum (make-add 3 6)) ; ⇒ 9
```

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

Gets a property value associated with the symbol *`sym`*. The *`property`* argument is also a symbol that names the property to be found. If the property is not found, *`default`* is returned. If the *`default`* argument is omitted, `#f` is used as the default.

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

Emulates Chez Scheme's `new-cafe`.

`(putprop` *`sym property value`*`)` PROCEDURE

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

11. compile.ss: Compiling Files

To load: `(require (lib "compile.ss"))`

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

Compiles the Scheme file *`src`* and saves the compiled code to *`dest`*. If *`dest`* is not specified, a filename is constructed by taking *`src`*'s directory path, adding a **compiled** subdirectory, and then adding *`src`*'s filename with its suffix replaced by **.zo**. Also, if *`dest`* is not provided and the **compiled** subdirectory does not already exist, the subdirectory is created. If the *`filter`* procedure is provided, it is applied to each source expression and the result is compiled (otherwise, the identity function is used as the filter). The result of `compile-file` is the destination file's path.

The `compile-file` procedure is designed for compiling modules files; each expression in *`src`* is compiled independently. If *`src`* does not contain a single `module` expression, then earlier expressions can affect the compilation of later expressions when *`src`* is loaded directly. An appropriate *`filter`* can make compilation behave like evaluation, but the problem is also solved (as much as possible) by the `compile-zos` procedure provided by the **compiler** collection's **compiler.ss** module.

See also `managed-compile-zo` in §6.

12. contract.ss: Contracts

To load: `(require (lib "contract.ss"))`

MzLib's **contract.ss** library defines new forms of expression that specify contracts and new forms of expression that attach contracts to values.

This section describes three classes of contracts: contracts for flat values (described in section 12.1), contracts for functions (described in section 12.2), and contracts for objects and classes (described in section 12.4).

In addition, this section describes how to establish a contract, that is, how to indicate that a particular contract should be enforced at a particular point in the program (in section 12.5).

12.1 Flat Contracts

A contract for a flat value can be a predicate that accepts the value and returns a boolean indicating if the contract holds.

`(flat-contract` *`predicate`*`)` FLAT-CONTRACT

Constructs a contract from *`predicate`*.

`(flat-named-contract` *`type-name predicate`*`)` FLAT-CONTRACT

For better error reporting, a flat contract can be constructed with *`flat-named-contract`*, a procedure that accepts two arguments. The first argument must be a string that describes the type that the predicate checks for. The second argument is the predicate itself.

`any/c` FLAT-CONTRACT

`any/c` is a flat contract that accepts any value.

If you are using this predicate as the result portion of a function contract, consider using *`any`* instead. It behaves the same, but in that one restrictive context has better memory performance.

`(or/c` *`contract`* `···)` OR/C

`or/c` accepts any number of predicates and at most one function contract and returns a contract that accepts any value that any one of the contracts accepts, individually.

If all of the arguments are predicates or flat contracts, it returns a flat contract.

`or/c` tests any values by applying the contracts in order, from left to right, with the exception that it always moves the non-flat contract (if any) to the end, applying it last.

`(and/c contract ···)` CONTRACT

`and/c` accepts any number of flat contracts and at most one function contract and returns a contract that checks that accepts any value that satisfies all of the contracts, simultaneously.

If all of the arguments are predicates or flat contracts, `and/c` produces a flat contract.

`and/c` tests any values by applying the contracts in order, from left to right, with the exception that it always moves the non-flat contract (if any) to the end, applying it last.

`(not/c flat-contract)` FLAT-CONTRACT

`not/c` accepts a flat contracts or a predicate and returns a flat contract that checks the inverse of the argument.

`(=/c number)` FLAT-CONTRACT

`=/c` accepts a number and returns a flat contract that requires the input to be a number and equal to the original input.

`(>=/c number)` FLAT-CONTRACT

`>=/c` accepts a number and returns a flat contract that requires the input to be a number and greater than or equal to the original input.

`(<=/c number)` FLAT-CONTRACT

`<=/c` accepts a number and returns a flat contract that requires the input to be a number and less than or equal to the original input.

`(between/c number number)` FLAT-CONTRACT

`between/c` accepts two numbers and returns a flat contract that requires the input to between the two numbers (or equal to one of them).

`(>/c number)` FLAT-CONTRACT

`>/c` accepts a number and returns a flat contract that requires the input to be a number and greater than the original input.

`(</c number)` FLAT-CONTRACT

`</c` accepts a number and returns a flat contract that requires the input to be a number and less than the original input.

`(integer-in number number)` FLAT-CONTRACT

`integer-in` accepts two numbers and returns a flat contract that recognizes if integers between the two inputs, or equal to one of its inputs.

`(real-in number number)` FLAT-CONTRACT

`real-in` accepts two numbers and returns a flat contract that recognizes real numbers between the two inputs, or equal to one of its inputs.

`natural-number/c` FLAT-CONTRACT

`natural-number/c` is a contract that recognizes natural numbers (*i.e.*, an integer that is either positive or zero).

`(string/len` *`number`*`)` FLAT-CONTRACT

`string/len` accepts a number and returns a flat contract that recognizes strings that have fewer than that number of characters.

`false/c` FLAT-CONTRACT

`false/c` is a flat contract that recognizes `#f`.

`printable/c` FLAT-CONTRACT

`printable/c` is a flat contract that recognizes values that can be written out and read back in with `write` and `read`.

`(one-of/c` *`value`* ···[1]`)` FLAT-CONTRACT

`one-of/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` *`symbol`* ···[1]`)` FLAT-CONTRACT

`symbols` accepts any number of symbols and returns a flat contract that recognizes those symbols.

`(is-a?/c` *`class-or-interface`*`)` FLAT-CONTRACT

`is-a?/c` accepts a class or interface and returns a flat contract that recognizes if objects are subclasses of the class or implement the interface.

`(implementation?/c` *`interface`*`)` FLAT-CONTRACT

`implementation?/c` accepts an interface and returns a flat contract that recognizes if classes are implement the given interface.

`(subclass?/c` *`class`*`)` FLAT-CONTRACT

`subclass?/c` accepts a class and returns a flat-contract that recognizes classes that are subclasses of the original class.

`(listof` *`flat-contract`*`)` FLAT-CONTRACT

`listof` accepts a flat contract (or a predicate which is converted to a flat contract) and returns a flat contract that checks for lists whose elements match the original flat contract.

`(list-immutableof` *`contract`*`)` CONTRACT

`list-immutableof` accepts a contract (or a predicate which is converted to a flat contract) and returns a

contract that checks for immutable lists whose elements match the original contract. In contrast to *listof*, *list-immutableof* accepts arbitrary contracts, not just flat contracts.

Beware, however, that when a value is applied to this contract, the result will not be `eq?` to the input.

`(vectorof` *`flat-contract`*`)` FLAT-CONTRACT

vectorof accepts a flat contract (or a predicate which is converted to a flat contract via *flat-contract*) and returns a predicate that checks for vectors whose elements match the original flat contract.

`(vector-immutableof` *`contract`*`)` CONTRACT

vector-immutableof accepts a contract (or a predicate which is converted to a flat contract) and returns a contract that checks for immutable lists whose elements match the original contract. In contrast to *vectorof*, *vector-immutableof* accepts arbitrary contracts, not just flat contracts.

Beware, however, that when a value is applied to this contract, the result will not be `eq?` to the input.

`(vector/c` *`flat-contract`* ···`)` FLAT-CONTRACT

vector/c accepts any number of flat contracts (or predicates which 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 the elements of the vector must match the corresponding flat contracts.

`(vector-immutable/c` *`contract`* ···`)` CONTRACT

vector-immutable/c accepts any number of contracts (or predicates which are converted to flat contracts via *flat-contract*) and returns a contract that recognizes vectors. The number of elements in the vector must match the number of arguments supplied to *vector-immutable/c* and the elements of the vector must match the corresponding contracts.

In contrast to *vector/c*, *vector-immutable/c* accepts arbitrary contracts, not just flat contracts. Beware, however, that when a value is applied to this contract, the result will not be `eq?` to the input.

`(box/c` *`flat-contract`*`)` FLAT-CONTRACT

box/c accepts a flat contract (or predicate that is converted to a flat contract via *flat-contract*) and returns a flat contract that recognizes for boxes whose contents match *box/c*'s argument.

`(box-immutable/c` *`contract`*`)` CONTRACT

box-immutable/c one contracts (or a predicate that is converted to a flat contract via *flat-contract*) and returns a contract that recognizes boxes. The contents of the box must match the contract passed to to *box-immutable/c*.

In contrast to *box/c*, *box-immutable/c* accepts an arbitrary contract, not just a flat contract. Beware, however, that when a value is applied to this contract, the result will not be `eq?` to the input.

`(cons/c` *`flat-contract flat-contract`*`)` FLAT-CONTRACT

cons/c accepts two flat contracts (or predicates that are converted to flat contracts via *flat-contract*) and returns a flat contract that recognizes cons cells whose car and cdr correspond to *cons/c*'s two arguments.

`(cons-immutable/c contract contract)` CONTRACT

cons-immutable/c accepts two contracts (or predicates that are converted to flat contracts via *flat-contract*) and returns a contract that recognizes immutable cons cells whose car and cdr correspond to *cons-immutable/c*'s two arguments. In contrast to *cons/c*, *cons-immutable/c* accepts arbitrary contracts, not just flat contracts.

Beware, however, that when a value is applied to this contract, the result will not be `eq?` to the input.

`(list/c flat-contract ···)` FLAT-CONTRACT

list/c accepts an arbitrary number of flat contracts (or predicates that are converted to flat contracts via *flat-contract*) and returns a flat contract that recognizes for lists whose length is the same as the number of arguments to *list/c* and whose elements match those arguments.

`(list-immutable/c contract ···)` CONTRACT

list-immutable/c accepts an arbitrary number of contracts (or predicates that are converted to flat contracts via *flat-contract*) and returns a contract that recognizes for lists whose length is the same as the number of arguments to *list-immutable/c* and whose elements match those contracts.

In contrast to *list/c*, *list-immutable/c* accepts arbitrary contracts, not just flat contracts. Beware, however, that when a value is applied to this contract, the result will not be `eq?` to the input.

`(syntax/c flat-contract)` FLAT-CONTRACT

syntax/c accepts a flat contract and produces a flat contract that recognizes syntax objects whose contents match the argument to *syntax/c*.

`(struct/c struct-name flat-contract ...)` FLAT-CONTRACT

struct/c accepts a struct name and as many flat contracts as there are fields in the named struct. It returns a contract that accepts instances of that struct whose fields match the given contracts.

`(flat-rec-contract name flat-contract ···)` SYNTAX

Each *flat-rec-contract* form constructs a flat recursive contract. The first argument is the name of the contract and the following arguments are flat contract expressions that may refer to *name*.

As an example, this 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 ([name flat-contract ...] ...) body ...)
```
SYNTAX

The *flat-murec-contract* form is a generalization of *flat-rec-contracts* for defining several mutually recursive flat contracts simultaneously.

Each of the names is visible in the entire *flat-murec-contract* and the result of the final body expression is the result of the entire form.

Note that if the contract is applied to a circular value, contract checking will not terminate.

12.2 Function Contracts

This section describes the contract constructors for function contracts. This is their shape:

```
contract-expr ::==
 | (case-> arrow-contract-expr ...)
 | arrow-contract-expr

arrow-contract-expr ::==
 | (-> expr ... expr)
 | (-> expr ... any)
 | (-> expr ... (values expr ...))

 | (->* (expr ...) (expr ...))
 | (->* (expr ...) any)
 | (->* (expr ...) expr (expr ...))
 | (->* (expr ...) expr any)

 | (->d expr ... expr)
 | (->d* (expr ...) expr)
 | (->d* (expr ...) expr expr)

 | (->r ((id expr) ...) expr)
 | (->r ((id expr) ...) any)
 | (->r ((id expr) ...) (values (id expr) ...))
 | (->r ((id expr) ...) id expr expr)
 | (->r ((id expr) ...) id expr any)
 | (->r ((id expr) ...) id expr (values (id expr) ...))

 | (->pp ((id expr) ...) pre-expr expr res-id post-expr)
 | (->pp ((id expr) ...) pre-expr any)
 | (->pp ((id expr) ...) pre-expr (values (id expr) ...) post-expr)

 | (->pp-rest ((id expr) ...) id expr pre-expr expr res-id post-expr)
 | (->pp-rest ((id expr) ...) id expr pre-expr any)
 | (->pp-rest ((id expr) ...) id expr pre-expr (values (id expr) ...) post-expr)

 | (opt-> (expr ...) (expr ...) expr)
 | (opt->* (expr ...) (expr ...) any)
 | (opt->* (expr ...) (expr ...) (expr ...))
```

where *expr* is any expression.

```
(-> expr ...)
```
SYNTAX

(-> *expr* ···**any**) SYNTAX

The -> contract is for functions that accept a fixed number of arguments and return a single result. The last argument to -> is the contract on the result of the function and the other arguments are the contracts on the arguments to the function. Each of the arguments to -> must be another contract expression or a predicate. For example, this expression:

```
(integer? boolean? . -> . integer?)
```

is a contract on functions of two arguments. The first must be an integer and the second a boolean and the function must return an integer. (This example uses MzScheme's infix notation so that the -> appears in a suggestive place; see §11.2.4 in *PLT MzScheme: Language Manual*).

If any is used as the last argument to ->, no contract checking is performed on the result of the function, and tail-recursion is preserved. Except for the memory performance, this is the same as using *any/c* in the result.

The final case of -> expressions treats values as a local keyword – that is, you may not return multiple values to this position, instead if the word values syntactically appears in the in the last argument to -> the function is treated as a multiple value return.

(->* *(expr ···) (expr ···)*) SYNTAX

(->* *(expr ···)* **any**) SYNTAX

(->* *(expr ···) expr (expr ···)*) SYNTAX

(->* *(expr ···) expr* **any**) SYNTAX

The ->* expression is for functions that return multiple results and/or have rest arguments. If two arguments are supplied, the first is the contracts on the arguments to the function and the second is the contract on the results of the function. These situations are also covered by ->.

If three arguments are supplied, the first argument contains the contracts on the arguments to the function (excluding the rest argument), the second contains the contract on the rest argument to the function and the final argument is the contracts on the results of the function. The final argument can be any which, like -> means that no contract is enforced on the result of the function and tail-recursion is preserved.

(->d *expr ···*) SYNTAX

(->d* *(expr ···) expr)*) SYNTAX

(->d* *(expr ···) expr expr*) SYNTAX

The ->d and ->d* contract constructors are like their **d**-less counterparts, except that the result portion is a function that accepts the original arguments to the function and returns the range contracts. The range contract function for ->d* must return multiple values: one for each result of the original function. As an example, this is the contract for sqrt:

```
(number?
 . ->d .
 (lambda (in)
   (lambda (out)
     (and (number? out)
```

```
        (abs (- (* out out) in) 0.01)))))
```

It says that the input must be a number and that the difference between the square of the result and the original number is less than `0.01`.

`(->r ([id expr] ···) expr)` SYNTAX

The `->r` contract allows you to build a contract where the arguments to a function may all depend on each other and the result of the function may depend on all of the arguments.

Each of the *`id`*s names one of the actual arguments to the function with the contract. Each of the names is available to all of the other contracts. For example, to define a function that accepts three arguments where the second argument and the result must both be between the first, you might write:

```
(->r ([x number?] [y (and/c (>=/c x) (<=/c z))] [z number?])
     (and/c number? (>=/c x) (<=/c z)))
```

`(->r ([id expr] ···) any)` SYNTAX

This variation on `->r` does not check anything about the result of the function, which preserves tail recursion.

`(->r ([id expr] ···) (values [id expr] ...))` SYNTAX

This variation on `->r` allows multiple value return values. The *`id`*s for the domain are bound in all of the *`expr`*s, but the *`id`*s for the range (the ones inside `values`) are only bound in the *`exprs`* inside the `values`.

As an example, this contract:

```
(->r () (values [x number?]
                [y (and/c (>=/c x) (<=/c z))]
                [z number?]))
```

matches functions that accept no arguments and that return three numeric values that are in ascending order.

`(->r ([id expr] ···) id expr expr)` SYNTAX

`(->r ([id expr] ···) id expr any)` SYNTAX

`(->r ([id expr] ···) id expr (values [id expr] ...))` SYNTAX

These three forms of the `->r` contract are just like the previous ones, except that the functions they matches must accept arbitrarily many arguments. The extra *`id`* and the *`expr`* just following it specify the contracts on the extra arguments. The value of *`id`* will alway be a list (of the extra arguments).

`(->pp ([id expr] ···) pre-expr expr res-id post-expr)` SYNTAX

`(->pp ([id expr] ···) pre-expr any)` SYNTAX

`(->pp ([id expr] ···) pre-expr (values [id expr] ...)  post-expr)` SYNTAX

`(->pp-rest ([id expr] ···) id expr pre-expr expr res-id post-expr)` SYNTAX

(->pp-rest *([id expr] ···) id expr pre-expr any*) SYNTAX

(->pp-rest *([id expr] ···) id expr pre-expr (values [id expr] ...) post-expr*) SYNTAX

These six shapes of ->pp match up to the six shapes of ->r forms explained above, with the addition that the extra pre- and post-condition expressions must not evaluate to #f.

If the pre-condition evaluates to #f, the caller is blamed and if the post-condition expression evaluates to #f the function itself is blamed.

The argument variables are bound in the *pre-expr* and the *post-expr* and the variables in the values result clauses are bound in the *post-expr*.

Additionally, the variable *res-id* is bound to the result in the first ->pp case and in the first ->*pp-rest* case.

(case-> *arrow-contract-expr ···*) CONTRACT-CASE-¿

The case-> expression constructs a contract for case-λ function. It's arguments must all be function contracts, built by one of ->, ->d, ->∗, or ->d∗.

(opt-> *(req-contracts ···) (opt-contracts ···) res-contract)*) SYNTAX

(opt->∗ *(req-contracts ···) (opt-contracts ···) (res-contracts ···)*) SYNTAX

(opt->∗ *(req-contracts ···) (opt-contracts ···) any*) SYNTAX

The opt-> expression constructs a contract for an opt-lambda function. The first arguments are the required parameters, the second arguments are the optional parameters and the final argument is the result. The *req-contracts* expressions, the *opt-contracts* expressions, and the *res-contract* expressions can be any expression that evaluates to a contract value.

Each opt-> expression expands into case->.

The opt->∗ expression constructs a contract for an opt-lambda function. The only difference between opt-> and opt->∗ is that multiple return values are permitted with opt->∗ and they are specified in the last clause of an opt->∗ expression. A result of any means any value or any number of values may be returned, and the contract does not inhibit tail-recursion.

12.3 Lazy Data-structure Contracts

Typically, constracts on data structures can be written using flat contracts. For example, one might write a sorted list contract as a function that accepts a list and traverses it, ensuring that the elements are in order. Such contracts, however, can change the asymptotic running time of the program, since the contract may end up exploring more of a function's input than the function itself does. To circumvent this problem, the define-contract-struct form introduces contract combinators that are *lazy* that is, they only verify the contract holds for the portion of some data structure that is actually inspected. More precisely, a lazy data structure contract on a struct is not checked until a selector extracts a field of a struct.

The form

```
(define-contract-struct struct-name (field ...))
```

is like the corresponding `define-struct`, with two differences: it does not define field mutators and it does define two contract constructors: *`struct-name/c`* and *`struct-name/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. It syntax is:

```
(struct-name/dc field-spec ...)
```

where each *`field-spec`* is one of the following two lines:

```
[field contract-expr]
[field (field ...) contract-expr]
```

In each case, the first field name specifies which field the contract applies to, and 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 field names in middle second indicate which fields it depends on. These dependencies can only be to fields that come earlier in the struct.

As an example consider this module:

```
(module product mzscheme
  (require (lib "contract.ss"))

  (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?)]))
```

It 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.

12.4 Object and Class Contracts

This section describes contracts on classes and objects. Here is the basic shape of an object contract:

```
contract-expr ::== ···
 | (object-contract meth/field-spec ···)

meth/field-spec ::==
   (meth-name meth-contract)
 | (field field-name contract-expr)

meth-contract ::==
   (opt-> (required-contract-expr ···)
          (optional-contract-expr ···)
          any)
   (opt-> (required-contract-expr ···)
          (optional-contract-expr ···)
          result-contract-expr)
 | (opt->* (required-contract-expr ···)
           (optional-contract-expr ···)
           (result-contract-expr ···))
 | (case-> meth-arrow-contract ···)
 | meth-arrow-contract

meth-arrow-contract ::==
   (-> dom-contract-expr ··· rng-contract-expr)
 | (-> dom-contract-expr ··· (values rng-contract-expr ···))
 | (->* (dom-contract-expr ···) (rng-contract-expr ···))
 | (->* (dom-contract-expr ···) rest-arg-contract-expr (rng-contract-expr ···))
 | (->d dom-contract-expr ··· rng-contract-proc-expr)
 | (->d* (dom-contract-expr ···) rng-contract-proc-expr)
 | (->d* (dom-contract-expr ···) rest-contract-expr rng-contract-proc-expr)
 | (->r ((id expr) ···) expr)
 | (->r ((id expr) ···) id expr expr)
 | (->pp ((id expr) ···) pre-expr expr res-id post-expr)
 | (->pp ((id expr) ···) pre-expr any)
 | (->pp ((id expr) ···) pre-expr (values (id expr) ···) post-expr)
 | (->pp-rest ((id expr) ···) id expr pre-expr expr res-id post-expr)
 | (->pp-rest ((id expr) ···) id expr pre-expr any)
 | (->pp-rest ((id expr) ···) id expr pre-expr (values (id expr) ···) post-expr)
```

Each of the contracts for methods has the same semantics as the corresponding function contract (discussed above), 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 the `->r`, `->pp`, and `->pp-rest` contracts implicitly bind `this` to the object itself.

`mixin-contract` CONTRACT

mixin-contract is a contract that recognizes mixins. It is a function contract. 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 class-or-interface ···)` CONTRACT

make-mixin-contract is a function that constructs mixins contracts. It accepts any number of classes and interfaces and returns a function contract. The function contract guarantees that the input to the function implements the interfaces and is derived from the classes and that the result of the function is a subclass of the input.

12.5 Attaching Contracts to Values

There are three special forms that attach contract specification to values: `provide/contract`, `define/contract`, and `contract`.

`(provide/contract p/c-item ···)` SYNTAX

```
p/c-item is one of
  (struct identifier ((identifier contract-expr) ···))
  (struct (identifier identifier) ((identifier contract-expr) ···))
  (rename id id contract-expr)
  (id contract-expr)
```

A `provide/contract` form can only appear at the top-level of a module (see §5 in *PLT MzScheme: Language Manual*). As with `provide`, each identifier is provided from the module. In addition, clients of the module must live up to the contract specified by *contract-expr*.

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. Each field has a contract that dictates the contents of the fields.

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.

Note that the struct definition must come before the provide clause in the module's body.

`(define/contract id contract-expr init-value-expr)` SYNTAX

The `define/contract` form 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 negative-blame*) SYNTAX

(contract *contract-expr to-protect-expr positive-blame negative-blame contract-source*) SYNTAX

The `contract` special form is the primitive mechanism for attaching a contract to a value. Its purpose is as a target for the expansion of some higher-level contract specifying form.

The `contract` form has this shape:

(contract *expr to-protect-expr positive-blame negative-blame contract-source*)

The `contract` expression adds the contract specified by the first argument to the value in the second argument. 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 expressions *positive-blame* and *negative-blame* must be symbols indicating how to assign blame for positive and negative positions of the contract specified by *contract-expr*. Finally, *contract-source*, if specified, indicates where the contract was assumed. It 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.

12.6 Contract Utility

`contract?` PREDICATE

The procedure *contract?* returns `#t` if its argument is a contract (ie, constructed with one of the combinators described in this section).

`flat-contract?` PREDICATE

This predicate returns true when its argument is a contract that has been constructed with *flat-contract* (and thus is essentially just a predicate).

(flat-contract-predicate *value*) SELECTOR

This function extracts the predicate from a flat contract.

(contract-violation->string [*violation-renderer*]) PROCEDURE

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*) SYNTAX

Unfortunately, the standard contract combinators (like `->`, etc) evaluate their arguments eagerly, leading to either references to undefined variables or infinite loops, while building recursive contracts.

The *recursive-contract* form delays the evaluation of its argument until the contract is checked, making recursive contracts possible.

13. date.ss: Dates

To load: `(require (lib "date.ss"))`

See also §15.1 in *PLT MzScheme: Language Manual*.

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

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

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

Parameter that determines the date display format, one of `'american`, `'chinese`, `'german`, `'indian`, `'irish`, `'iso-8601`, `'rfc2822`, or `'julian`. The initial format is `'american`.

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

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

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

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

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

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

14. deflate.ss: Deflating (Compressing) Data

To load: `(require (lib "deflate.ss"))`

`(gzip` *`in-filename`* [*`out-filename`*]`)` PROCEDURE

Compresses data to the same format as the GNU `gzip` utility, writing the compressed data directly to a file. The *`in-filename`* argument is the name of the file to compress. The default output file name is *`in-filename`* with **.gz** appended. If the file named by *`out-filename`* exists, it will be overwritten. The return value is void.

`(gzip-through-ports` *`in out orig-filename timestamp`*`)` PROCEDURE

Reads the port *`in`* for data and compresses it to *`out`*, outputting the same format as the GNU `gzip` utility. The *`orig-filename`* string is embedded in this output; *`orig-filename`* can be `#f` to omit the filename from the compressed stream. The *`timestamp`* number is also embedded in the output stream, as the modification date of the original file (in Unix seconds, as `file-or-directory-modify-seconds` would report under Unix). The return value is void.

`(deflate` *`in out`*`)` PROCEDURE

Writes `pkzip`-format "deflated" data to the port *`out`*, compressing data from the port *`in`*. The data in a file created by `gzip` uses this format (preceded with some header information).

The result is three values: the number of bytes read from *`in`*, the number of bytes written to *`out`*, and a cyclic redundancy check (CRC) value for the input.

15. defmacro.ss: Non-Hygienic Macros

To load: `(require (lib "defmacro.ss"))`

`(define-macro identifier expr)` SYNTAX

`(define-macro (identifier . formals) expr ···1)` SYNTAX

Defines a (non-hygienic) macro *identifier* as a procedure that manipulates S-expressions (as opposed to syntax objects). In the first form, *expr* must produce a procedure. In the second form, *formals* determines the formal arguments of the procedure, as in `lambda`, and the *expr*s are the procedure body. In both cases, the procedure is generated in the transformer environment, not the normal environment (see §12 in *PLT MzScheme: Language Manual*).

In a use of the macro,

```
(identifier expr ···)
```

`syntax-object->datum` is applied to the expression (see §12.2.2 in *PLT MzScheme: Language Manual*), and the macro procedure is applied to the `cdr` of the resulting list. If the number of *expr*s does not match the procedure's arity (see §3.12.1 in *PLT MzScheme: Language Manual*) or if *identifier* is used in a context that does not match the above pattern, then a syntax error is reported.

After the macro procedure returns, the result is compared to the procedure's arguments. For each value that appears exactly once within the arguments (or, more precisely, within the S-expression derived from the original source syntax), if the same value appears in the result, it is replaced with a syntax object from the original expression. This heuristic substitution preserves source location information in many cases, despite the macro procedure's operation on raw S-expressions.

After substituting syntax objects for preserved values, the entire macro result is converted to syntax with `datum->syntax-object` (see §12.2.2 in *PLT MzScheme: Language Manual*). The original expression supplies the lexical context and source location for converted elements.

`(defmacro identifier formals expr ···1)` SYNTAX

Same as `(define-macro (identifier . formals) expr ···1)`.

Important: `define-macro` is still restricted by MzScheme's phase separation rules. This means that a macro cannot access run-time bindings because it is executed in the syntax expansion phase. Translating code that involves `define-macro` or `defmacro` from an implementation without this restriction usually implies separating macro related functionality into a `begin-for-syntax` or a module (that will be imported with `require-for-syntax`) and properly distinguishing syntactic information from run-time information.

16. etc.ss: Useful Procedures and Syntax

To load: `(require (lib "etc.ss"))`

`(begin-lifted expr ···1)` SYNTAX

Lifts the *expr*s so that they are evaluated once at the "top level" of the current context, and the result of the last *expr* is used for every evaluation of the `begin-lifted` form.

When this form is used as a run-time expression within a module, the "top level" corresponds to the module's top level, so that each *expr* is evaluated once for each invocation of the module. When it is used as a run-time expression outside of a module, the "top level" corresponds to the true top level. When this form is used in a `define-syntax`, `letrec-syntax`, etc. binding, the "top level" corresponds to the beginning of the binding's right-hand side. Other forms may redefine "top level" (using `local-expand/capture-lifts`) for the expressions that they enclose.

`(begin-with-definitions defn-or-expr ···)` SYNTAX

Supports a mixture of expressions and mutually recursive definitions, much like a `module` body. Unlike in a `module`, however, syntax definitions cannot be used to generate other immediate definitions (though they can be used for expressions).

The result of the `begin-with-definitions` form is the result of the last `defn-or-expr` if it is an expression, void otherwise. If no `defn-or-expr` is provided (after flattening `begin` forms), the result is void.

`(boolean=? bool1 bool2)` PROCEDURE

Returns `#t` if *bool1* and *bool2* are both `#t` or both `#f`, and returns `#f` otherwise. If either *bool1* or *bool2* is not a Boolean, the `exn:fail:contract` exception is raised.

`(build-list n f)` PROCEDURE

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

`(build-string n f)` PROCEDURE

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

`(build-vector n f)` PROCEDURE

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

`(compose f ···1)` PROCEDURE

Returns a procedure that composes the given functions, applying the last `f` first and the first `f` 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.

For example, `(compose f g)` returns the equivalent of `(lambda l (call-with-values (lambda () (apply g l)) f))`.

`(define-syntax-set (identifier ···) defn ···)` SYNTAX

This form is similar to `define-syntaxes`, but instead of a single body expression, a sequence of definitions follows the sequence of defined identifiers. For each *`identifier`*, the *`defn`*s should include a definition for *`identifier`*`/proc`. The value for *`identifier`*`/proc` is used as the (expansion-time) value for *`identifier`*.

The `define-syntax-set` form is especially useful for defining a set of syntax transformers that share helper functions.

Example:

```
(define-syntax-set (let-current-continuation let-current-escape-continuation)
  (define (mk call-id)
     (lambda (stx)
       (syntax-case stx ()
         [(_ id body1 body ...)
          (with-syntax ([call call-id])
            (syntax (call (lambda (id) body1 body ...))))])))
  (define let-current-continuation/proc (mk (quote-syntax call/cc)))
  (define let-current-escape-continuation/proc (mk (quote-syntax call/ec))))
```

`(evcase key-expr (value-expr body-expr ···) ···1)` SYNTAX

The `evcase` form is similar to `case`, except that expressions are provided in each clause instead of a sequence of data. After *`key-expr`* is evaluated, each *`value-expr`* is evaluated until a value is found that is `eqv?` to the key value; when a matching value is found, the corresponding *`body-expr`*s are evaluated and the value(s) for the last is the result of the entire `evcase` expression.

A *`value-expr`* can be the special identifier `else`. This identifier is recognized as in `case` (see §2.3 in *PLT MzScheme: Language Manual*).

`false` BOOLEAN

Boolean false.

`(identity v)` PROCEDURE

Returns *`v`*.

`(let+ clause body-expr ···1)` SYNTAX

A new binding construct that specifies scoping on a per-binding basis instead of a per-expression basis. It helps eliminate rightward-drift in programs. It looks similar to `let`, except each clause has an additional keyword tag before the binding variables.

Each *clause* has one of the following forms:

- (val *target expr*) binds *target* non-recursively to *expr*.
- (rec *target expr*) binds *target* recursively to *expr*.
- (vals (*target expr*) ···) the *target*s are bound to the *expr*s. The environment of the *expr*s is the environment active before this clause.
- (recs (*variable expr*) ···) the *targets*s are bound to the *expr*s. The environment of the *expr*s includes all of the *targets*s.
- (_ *expr* ···) evaluates the *expr*s without binding any variables.

The clauses bind left-to-right. Each *target* above can either be an identifier or (values *variable* ···). In the latter case, multiple values returned by the corresponding expression are bound to the multiple variables.

Examples:

```
(let+ ([val (values x y) (values 1 2)])
   (list x y)) ; ⇒ '(1 2)

(let ([x 1])
   (let+ ([val x 3]
          [val y x])
      y)) ; ⇒ 3
```

(local (*definition* ···) *body-expr* ···[1]) SYNTAX

This is a binding form similar to letrec, except that each *definition* is a define-values expression (after partial macro expansion). The *body-expr*s are evaluated in the lexical scope of these definitions.

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

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

```
(define loop-until
  (lambda (start done? next f)
    (let loop ([i start])
      (unless (done? i)
        (f i)
        (loop (next i))))))
```

(namespace-defined? *symbol*) PROCEDURE

Returns #t if namespace-variable-value would return a value for *symbol*, #f otherwise. See §8.2 in *PLT MzScheme: Language Manual* for further information.

(nand *expr* ···) SYNTAX

Returns (not (and *expr* ···)).

`(nor expr ···)` SYNTAX

Returns `(not (or expr ···))`.

`(opt-lambda formals body-expr ···¹)` SYNTAX

The `opt-lambda` form is like `lambda`, except that default values are assigned to arguments (C++-style). Default values are defined in the *`formals`* list by replacing each *`variable`* by `[`*`variable default-value-expression`*`]`. If a variable has a default value expression, then all (non-aggregate) variables after it must have default value expressions. A final aggregate variable can be used as in `lambda`, but it cannot be given a default value. Each default value expression is evaluated only if it is needed. The environment of each default value expression includes the preceding arguments.

For example:

```
(define f
  (opt-lambda (a [b (add1 a)] . c)
    ...))
```

In the example, `f` is a procedure which takes at least one argument. If a second argument is specified, it is the value of *`b`*, otherwise *`b`* is `(add1 a)`. If more than two arguments are specified, then the extra arguments are placed in a new list that is the value of *`c`*.

See also §22 for a library that generalizes both optional and keyword arguments.

`(recur name bindings body-expr ···¹)` SYNTAX

This is equivalent to a named `let`: `(let name bindings body-expr ···¹)`.

`(rec name value-expr)` SYNTAX

This is equivalent to a `letrec` expression that returns its binding: `(letrec ((name value-expr)) name)`.

`(symbol=? symbol1 symbol2)` PROCEDURE

Returns `#t` if *`symbol1`* and *`symbol2`* are equivalent (as determined by `eq?`), `#f` otherwise. If either *`symbol1`* or *`symbol2`* is not a symbol, the `exn:fail:contract` exception is raised.

`(this-expression-source-directory)` SYNTAX

Expands to an expression that evaluates to the name of the directory of the file containing the source expression. The source expression's file is determined through source location information associated with the syntax if it is present. Otherwise, `current-load-relative-directory` is used if it is not `#f`, and `current-directory` is used if all else fails. The expression is a simple `(bytes->path #"...")`, unless the directory is in the PLT home, which will make an expression that uses '*`plthome`*' to get the PLT home path determined at runtime, therefore not hard-wiring the path to the resulting syntax.

`(this-expression-file-name)` SYNTAX

Expands to an expression that evaluates to the file name of the source expression. The source expression's file name is determined through source location information associated with the syntax if it is present. If this information is missing, or is not a path (e.g., for a standard-input expression), then `#f` will be used instead.

`true` BOOLEAN

Boolean true.

`(hash-table` *`'flag ...(key value) ...`*`)` SYNTAX

This creates a new hash-table providing the quoted flags (if any) to `make-hash-table`, and make each of the keys map to the corresponding values. (Flags must be specified by a quoted form.)

17. file.ss: Filesystem Utilities

To load: `(require (lib "file.ss"))`

See also §11.3 in *PLT MzScheme: Language Manual*.

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

Like `build-path` (see §11.3 in *PLT MzScheme: Language Manual*), but *`base`* is required to be an absolute pathname. If *`base`* is not an absolute pathname, `error` is called.

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

Like `build-path` (see §11.3 in *PLT MzScheme: Language Manual*), but *`base`* is required to be a relative pathname. If *`base`* is not a relative pathname, `error` is called.

`(call-with-input-file*` *`pathname proc flag-symbol`* `···)` PROCEDURE

Like `call-with-input-file`, except that the opened port is closed if control escapes from the body of *`proc`*.

`(call-with-output-file*` *`pathname proc flag-symbol`* `···)` PROCEDURE

Like `call-with-output-file`, except that the opened port is closed if control escapes from the body of *`proc`*.

`(copy-directory/files` *`src-path dest-path`*`)` PROCEDURE

Copies the file or directory *`src-path`* to *`dest-path`*, raising `exn:fail:filesystem` if the file or directory cannot be copied, possibly because *`dest-path`* exists already. If *`src-path`* 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`*`)` PROCEDURE

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. The return value is void.

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

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

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

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

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

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

`(find-files` *`predicate`* [*`start-pathname`*]`)` PROCEDURE

Traverses the filesystem starting at *`start-pathname`* and creates a list of all files and directories for which *`predicate`* returns true. If *`start-pathname`* is `#f` (the default), then the traversal starts from the current directory (as determined by `current-directory`; see §7.9.1.1 in *PLT MzScheme: Language Manual*). The resulting list has directories precede their contents.

The *`predicate`* procedure is called with a single argument for each file or directory. If *`start-pathname`* is `#f`, the argument is a pathname string that is relative to the current directory. Otherwise, it is a pathname that starts with *`start-pathname`*. Consequently, supplying `(current-directory)` for *`start-pathname`* 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-pathname`* is `#f`.

The `find-files` traversal follows soft links. To avoid following links, use the more general `fold-files` procedure.

If *`start-pathname`* does not refer to an existing file or directory, then *`predicate`* will be called exactly once with *`start-pathname`* as the argument.

`(pathlist-closure` *`path-list`*`)` PROCEDURE

This procedure consumes a list of paths, either absolute or relative to the current directory. The paths in the given *`path-list`* are all expected to be path names of existing directories and files. The return value is a list of paths 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 points at a directory, all of its descendants are also included in the result;
- ancestor directories come before their descendants.

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

Returns the path of the specified library (see Chapter 16 in *PLT MzScheme: Language Manual*), returning `#f` if the specified library or collection cannot be found. The *`collection`* argument is optional, defaulting to `"mzlib"`.

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

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

`(fold-files` *`proc init-val`* [*`start-pathname follow-links?`*]`)` PROCEDURE

Traverses the filesystem starting at *`start-pathname`*, calling *`proc`* on each discovered file, directory, and link. If *`start-pathname`* is `#f` (the default), then the traversal starts from the current directory (as determined by `current-directory`; see §7.9.1.1 in *PLT MzScheme: Language Manual*).

The *proc* procedure is called with three arguments for each file, directory, or link:

- If *start-pathname* 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-pathname*. Consequently, supplying `(current-directory)` for *start-pathname* 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-pathname* is `#f`.
- The second argument is a symbol, either `'file`, `'dir`, or `'link`. The second argument can be `'link` only when *follow-links?* is `#f`, in which case the filesystem traversal does not follow links. The default for *follow-links?* is `#t`.
- 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`.

If *start-pathname* does not refer to an existing file or directory, then *proc* will be called exactly once with *start-pathname* as the first argument, `'file` as the second argument, and *init-val* as the third argument.

`(get-preference` *name* [*failure-thunk flush-cache? filename*]`)` PROCEDURE

Extracts a preference value from the file designated by `(find-system-path 'pref-file)` (see §11.3 in *PLT MzScheme: Language Manual*), 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. The default *failure-thunk* returns `#f`.

Preference settings from the standard preference file are cached (weakly) across calls to `get-preference`; if *flush-cache?* is provided as `#f`, the cache is used instead of the re-consulting the preferences file.

See also `put-preferences`. The **framework** collection supports a more elaborate preference system; see *PLT Framework: GUI Application Framework* for details.

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

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

`(make-temporary-file` [*format-string copy-from-filename directory*]`)` PROCEDURE

Creates a new temporary file and returns a pathname string for the file. Instead of merely generating a fresh file name, the file is actually created; this prevents other threads or processes from picking the same temporary name. 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` or not provided, the temporary file is created as empty. If *copy-from-filename* is `'directory`, then the temporary "file" is created as a directory. If *directory* is provided and is not `#f`, then the file name generated from *format-string* is combined with *directory* to obtain a full path.

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 §11.1.3 in *PLT MzScheme: Language Manual*) and to delete it when it is no longer needed.

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

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

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

Letter case is *not* normalized by `normalize-path`. For this and other reasons, the result of `normalize-path` is not suitable for comparisons that determine whether two paths refer to the same file (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*`)` PROCEDURE

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

`(put-preferences` *name-list val-list* [*locked-proc filename*]`)` PROCEDURE

See also `get-preference`.

Installs a set of preference values and writes all current values to the preference file designated by `(find-system-path 'pref-file)` (see §11.3 in *PLT MzScheme: Language Manual*), or *filename* if it is supplied and not `#f`. The *name-list* argument must be a list of symbols for the preference names, and *val-list* must have the same length as *name-list*. Each element of *val-list* 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; see §7.9.1.4 in *PLT MzScheme: Language Manual*).

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-proc* is called with a single argument: the path of the lock file. The default *locked-proc* 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 *name-list*.

18. **foreign.ss**: Foreign Interface

To load: `(require (lib "foreign.ss"))`

The **foreign.ss** module provides functionality for interfacing with foreign functions and data, as well as making some of MzScheme's internal functionality available from Scheme. Unlike other modules in this manual, **foreign.ss** is intended to be used as a substitute for C extensions, making it inherently unsafe — code that uses such unsafe functionality *can crash* the running process. It is therefore documented in its own manual: *PLT Foreign Interface Manual*.

19. include.ss: Textually Including Source

To load: `(require (lib "include.ss"))`

`(include path-spec)` SYNTAX

Inlines the syntax in the designated file in place of the `include` expression.

The *`path-spec`* can be any of the following:

- a literal string that specifies a path to include (parsed according to the platform's conventions).
- a path construction of the form `(build-path elem ···`[1]`)`, where `build-path` is `module-identifier=?` either to the `build-path` export from `mzscheme` or to the top-level `build-path`, and where each *`elem`* is a path string, `up` (unquoted), or `same` (unquoted). The *`elem`*s are combined in the same way as for the `build-path` function (see §11.3.1 in *PLT MzScheme: Language Manual*) to obtain the path to include.
- a path construction of the form `(lib file-string collection-string ···)`, where `lib` is free or refers to a top-level `lib` variable. The *`collection-string`*s are passed to `collection-path` to obtain a directory; if no *`collection-string`*s are supplied, `"mzlib"` is used. The *`file-string`* is then appended to the directory using `build-path` to obtain the path to include.

If *`path-spec`* specifies a relative path to include, 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 the current load relative directory if one is available, or to the current directory otherwise.

The included syntax is given the lexical context of the `include` expression.

`(include-at/relative-to context source path-spec)` SYNTAX

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-at/relative-to/reader context source path-spec reader-expr)` SYNTAX

Combines `include-at/relative-to` and `include/reader`.

`(include/reader path-spec reader-expr)` SYNTAX

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.

20. inflate.ss: Inflating Compressed Data

To load: `(require (lib "inflate.ss"))`

`(gunzip` *`file`* [*`output-name-filter`*]`)` PROCEDURE

Extracts data that was compressed using the GNU `gzip` utility (or `gzip` in the **deflate.ss** library; see §14), writing the uncompressed data directly to a file. The *`file`* argument is the name of the file containing compressed data. The default output file name is the original name of the compressed file as stored in *`file`*. If a file by this name exists, it will be overwritten. If no original name is stored in the source file, `"unzipped"` is used as the default output file name.

The *`output-name-filter`* procedure is applied to two arguments — the default destination file name and a Boolean that is `#t` if this name was read from *`file`* — before the destination file is created. The return value of the file is used as the actual destination file name (opened with the `'truncate` flag). The default *`output-name-filter`* procedure returns its first argument.

The return value is void. If the compressed data is corrupted, the `exn:fail` exception is raised.

`(gunzip-through-ports` *`in out`*`)` PROCEDURE

Reads the port *`in`* for compressed data that was created using the GNU `gzip` utility, writing the uncompressed data to the port *`out`*.

The return value is void. If the compressed data is corrupted, the `exn:fail` exception is raised.

`(inflate` *`in out`*`)` PROCEDURE

Reads `pkzip`-format "deflated" data from the port *`in`* and writes the uncompressed ("inflated") data to the port *`out`*. The data in a file created by `gzip` uses this format (preceded with some header information).

The return value is void. If the compressed data is corrupted, the `exn:fail` exception is raised.

21. integer-set.ss: Integer Sets

To load: `(require (lib "integer-set.ss"))`

The **integer-set.ss** module provides functions for working with finite sets of integers. This module is designed for sets that are compactly represented as groups of intervals, even when their cardinality is large. For example, the set of integers from -1000000 to 1000000 except for 0, can be represented as $\{[-1000000, -1], [1, 1000000]\}$. This data structure would not be a good choice for the set of all odd integers between 0 and 1000000 (which would be $\{[1, 1], [3, 3], \ldots [999999, 999999]\}$).

In addition to the *`integer-set`* abstract type, we define a *`well-formed-set`* to be a list of pairs of exact integers, where each pair represents a closed range of integers, and the entire set is the union of the ranges. The ranges must be disjoint and increasing. Further, adjacent ranges must have at least one integer between them. For example: `'((-1 . 2) (4 . 10))` is a well-formed-set as is `'((1 . 1) (3 . 3))`, but `'((1 . 5) (6 . 7))`, `'((1 . 5) (-3 . -1))`, `'((5 . 1))`, and `'((1 . 5) (3 . 6))` are not.

`(make-integer-set` *`well-formed-set`*`)` PROCEDURE

Creates an integer set from a well-formed set.

`(integer-set-contents` *`integer-set`*`)` PROCEDURE

Produces a well-formed set from an integer set.

`(set-integer-set-contents!` *`integer-set well-formed-set`*`)` PROCEDURE

Mutates an integer set.

`(integer-set?` *`v`*`)` PROCEDURE

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

`(make-range)` make-range/empty

Produces an empty integer set.

`(make-range` *`k`*`)` PROCEDURE

Produces an integer set containing only *`k`*.

`(make-range` *`start-k end-k`*`)` PROCEDURE

Produces an integer set containing the integers from *`start-k`* to *`end-k`* inclusive, where *`start-k`* <= *`end-k`*.

`(intersect` *`x-integer-set y-integer-set`*`)` PROCEDURE

Returns the intersection of the given sets.

`(difference` *`x-integer-set y-integer-set`*`)` PROCEDURE

Returns the difference of the given sets (i.e., elements in *`x-integer-set`* that are not in *`y-integer-set`*).

`(union` *`x-integer-set y-integer-set`*`)` PROCEDURE

Returns the union of the given sets.

`(split` *`x-integer-set y-integer-set`*`)` PROCEDURE

Produces three values: the first is the intersection of *`x-integer-set`* and *`y-integer-set`*, the second is the difference *`x-integer-set`* remove *`y-integer-set`*, and the third is the difference *`y-integer-set`* remove *`x-integer-set`*.

`(complement` *`integer-set start-k end-k`*`)` PROCEDURE

Returns the a set containing the elements between *`start-k`* to *`end-k`* inclusive that are not in *`integer-set`*, where *`start-k`* <= *`end-k`*.

`(xor` *`x-integer-set y-integer-set`*`)` PROCEDURE

Returns an integer set containing every member of *`x-integer-set`* and *`y-integer-set`* that is not in both sets.

`(member?` *`k integer-set`*`)` PROCEDURE

Returns `#t` if *`k`* is in *`integer-set`*, `#f` otherwise.

`(get-integer` *`integer-set`*`)` PROCEDURE

Returns a member of *`integer-set`*, or `#f` if *`integer-set`* is empty.

`(foldr` *`proc base-v integer-set`*`)` PROCEDURE

Applies *`proc`* to each member of *`integer-set`* in ascending order, where the first argument to *`proc`* is the set member, and the second argument is the fold result starting with *`base-v`*. For example, `(foldr cons null` *`x`*`)` returns a list of all the integers in *`x`*, sorted in increasing order.

`(partition` *`integer-set-list`*`)` PROCEDURE

Returns the coarsest refinement of the sets in *`integer-set-list`* such that the sets in the result list are pairwise disjoint. For example, partitioning the sets that represent `'((1 . 2) (5 . 10))` and `'((2 . 2) (6 . 6) (12 . 12))` produces the a list containing the sets for `'((1 . 1) (5 . 5) (7 . 10))` `'((2 . 2) (6 . 6))`, and `'((12 . 12))`.

`(card` *`integer-set`*`)` PROCEDURE

Returns the number of integers in the given integer set.

(subset? *x-integer-set y-integer-set*) PROCEDURE

Returns true if every integer in *x-integer-set* is also in *y-integer-set*, otherwise #f.

22. kw.ss: Keyword Arguments

To load: `(require (lib "kw.ss"))`

The **kw.ss** library provides the `lambda/kw` and `define/kw` forms.

`(lambda/kw` *`formals body-expr`* `···`[1]`)` SYNTAX

Like `lambda`, but with optional and keyword-based argument processing. This form is similar to an extended version of Common Lisp procedure arguments (but note the differences below). When used with plain variable names, `lambda/kw` expands to a plain `lambda`, so `lambda/kw` is suitable for a language module that will use it to replace `lambda`. Also, when used with only optionals, the resulting procedure is similar to `opt-lambda` (but a bit faster). This facility uses MzScheme keyword values (see §3.8 in *PLT MzScheme: Language Manual*) for its implementation.

In addition to `lambda/kw`, this library provides a `define/kw` form that is similar to the built-in `define` (see §2.8.1 in *PLT MzScheme: Language Manual*), except that the *`formals`* are as in `lambda/kw`. Like `define`, this form can be used with nested parenthesis for curried functions (the MIT-style generalization of §2.8.1 in *PLT MzScheme: Language Manual*).

The syntax of `lambda/kw` is the same as `lambda`, except for the list of formal argument specifications. These specifications can hold (zero or more) plain argument names, then an optionals (and defaults) section that begins after an `#:optional` marker, then a keyword section that is marked by `#:keyword`, and finally a section holding rest and "rest-like" arguments which are described below, together with argument processing flag directives. Each section is optional, but the order of the sections must be as listed.

More formally, the syntax is:

```
(lambda/kw kw-formals body ...)

kw-formals is one of
  variable
  (variable ··· [#:optional optional-spec ···]
                [#:key key-spec ···]
                [rest/mode-spec ···])
  (variable ··· . variable)

optional-spec is one of
  variable
  (variable default-expr)

key-spec is one of
  variable
  (variable default-expr)
  (variable keyword default-expr)

rest/mode-spec is one of
  #:rest variable
```

```
#:other-keys variable
#:other-keys+body variable
#:all-keys variable
#:body kw-formals
#:allow-other-keys
#:forbid-other-keys
#:allow-duplicate-keys
#:forbid-duplicate-keys
#:allow-body
#:forbid-body
#:allow-anything
#:forbid-anything
```

Of course, all bound *identifier*s must be unique. The following section describes each part of the *kw-formals*.

22.1 Required Arguments

Required arguments correspond to *identifier*s that appear before any keyword marker in the argument list. They determine the minimum arity of the resulting procedure.

22.2 Optional Arguments

The optional-arguments section follows an `#:optional` marker in the *kw-formals*. Each optional argument can take the form of a parenthesized variable and a default expression; the latter is used if a value is not given at the call site. The default expression can be omitted (along with the parentheses), in which case `#f` is the default.

The default expression's environment includes all previous arguments, both required and optional names. With k optionals after n required arguments, and with no keyword arguments or rest-like arguments, the resulting procedure has an arity `'(`$n+k$ `...` $n+1$ n`)`. Adding keywords or rest-like arguments makes the first arity `(make-arity-at-least` $n+k$`)`.

The treatment of optionals is efficient, with an important implication: default expressions appear multiple times in the resulting `case-lambda`. For example, the default expression for the last optional argument appears $k-1$ times (but no expression is ever evaluated more than once in a procedure call). This expansion risks exponential blow-up is if `lambda/kw` is used in a default expression of a `lambda/kw`, etc. The bottom line, however, is that `lambda/kw` is a sensible choice, due to its enhanced efficiency, even when you need only optional arguments.

Using both optional and keyword arguments is possible, but note that the resulting behavior differs from traditional keyword facilities (including the one in Common Lisp). See the following section for details.

22.3 Keyword Arguments

A keyword argument section is marked by a `#:key`. If it is used with optional arguments, then the keyword specifications must follow the optional arguments (which mirrors the use in call sites; where optionals are given before keywords).

When a procedure accepts both optional and keyword arguments, the argument-handling convention is slightly different than in traditional keyword-argument facilities: a keyword after required arguments marks the beginning of keyword arguments, no matter how many optional arguments have been provided before the keyword. This convention restricts the procedure's non-keyword optional arguments to non-keyword values, but it also avoids confusion when mixing optional arguments and keywords. For example, when a procedure that takes two optional arguments and a keyword argument `#:x` is called with `#:x 1`, then the optional arguments get their default values and the keyword argument is bound to `1`. (The traditional behavior would bind `#:x` and `1` to the two optional arguments.) When the same procedure is called with `1 #:x 2`, the first optional argument is bound to `1`, the second optional argument is bound to its default, and the keyword argument is bound to `2`. (The traditional behavior would report an error, because `2` is provided where `#:x` is expected.)

Like optional arguments, each keyword argument is specified as a parenthesized variable name and a default expression. The default expression can be omitted (with the parentheses), in which case `#f` is the default value. The keyword used at a call site for the corresponding variable has the same name as the variable; a third form of keyword arguments has three parts — a variable name, a keyword, and a default expression — to allow the name of the locally bound variable to differ from the keyword used at call sites.

When calling a procedure with keyword arguments, the required argument (and all optional arguments, if specified) must be followed by an even number of arguments, where the first argument is a keyword that determines which variable should get the following value, etc. If the same keyword appears multiple times (and if multiple instances of the keyword are allowed; see §22.6), the value after the first occurrence is used for the variable:

```
((lambda/kw (#:key x [y 2] [z #:zz 3] #:allow-duplicate-keys) (list x y z))
 #:x 'x #:zz 'z #:x "foo")
```

⇒ `'(x 2 z)`

Default expressions are evaluated only for keyword arguments that do not receive a value for a particular call. Like optional arguments, each default expression is evaluated in an environment that includes all previous bindings (required, optional, and keywords that were specified on its left).

See §22.6 for information on when duplicate or unknown keywords are allowed at a call site.

22.4 Rest and Rest-like Arguments

The last *kw-formals* section — after the required, optional, and keyword arguments — may contain specifications for rest-like arguments and/or mode keywords. Up to five rest-like arguments can be declared, each with a *variable* to bind:

- `#:rest` — the variable is bound to the list of "rest" arguments, which is the list of all values after the required and the optional values. This list includes all keyword-value pairs, exactly as they are specified at the call site.

  Scheme's usual dot-notation is accepted in *kw-formals* only if no other meta-keywords are specified, since it is not clear whether it should specify the same binding as a `#:rest` or as a `#:body`. The dot notation is allowed without meta-keywords to make the `lambda/kw` syntax compatible with `lambda`.

- `#:body` — the variable is bound to all arguments after keyword–value pairs. (This is different from Common Lisp's `&body`, which is a synonym for `&rest`.) More generally, a `#:body` specification can be followed by another *kw-formals*, not just a single *variable*; see §22.5 for more information.

- `#:all-keys` — the variable is bound to the list of all keyword-values from the call site, which is always a proper prefix of a `#:rest` argument. (If no `#:body` arguments are declared, then `#:all-keys` binds the same as `#:rest`.) See also `keyword-get` in §22.7.

- `#:other-keys` — the variable is bound like an `#:all-keys` variable, except that all keywords specified in the *kw-formals* are removed from the list. When a keyword is used multiple times at a call cite (and this is allowed), only the first instances is removed for the `#:other-keys` binding.

- `#:other-keys+body` — the variable is bound like a `#:rest` variable, except that all keywords specified in the *kw-formals* are removed from the list. When a keyword is used multiple times at a call site (and this is allowed), only the first instance us removed for the `#:other-keys+body` binding. (When no `#:body` variables are specified, then `#:other-keys+body` is the same as `#:other-keys`.)

In the following example, all rest-like arguments are used and have different bindings:

```
((lambda/kw (#:key x y
             #:rest r
             #:other-keys+body rk
             #:all-keys ak
             #:other-keys ok
             #:body b)
   (list r rk b ak ok))
 #:z 1 #:x 2 2 3 4)
```

⇒

```
'((#:z 1 #:x 2 2 3 4)
  (#:z 1 2 3 4)
  (2 3 4)
  (#:z 1 #:x 2)
  (#:z 1))
```

Note that the following invariants always hold:

- *rest* = (append *all-keys body*)
- *other-keys+body* = (append *other-keys body*)

To write a procedure that uses a few keyword argument values, and that also calls another procedure with the same list of arguments (including all keywords), use `#:other-keys` (or `#:other-keys+body`). The Common Lisp approach is to specify `:allow-other-keys`, so that the second procedure call will not cause an error due to unknown keywords, but the `:allow-other-keys` approach risks confusing the two layers of keywords.

22.5 Body Argument

The most notable divergence from Common Lisp in `lambda/kw` is the `#:body` argument, and the fact that it is possible at a call site to pass plain values after keyword–value pairs. The `#:body` binding is useful for procedure calls that use keyword–value pairs as sort of an attribute list before the actual arguments to the procedure. For example, consider a procedure that accepts any number of numeric arguments and will apply a procedure to them, but the procedure can be specified as an optional keyword argument. It is easily implemented with a `#:body` argument:

```
(define/kw (mathop #:key [op +] #:body b)
  (apply op b))
(mathop 1 2 3) ; ⇒ 6
(mathop #:op max 1 2 3) ; ⇒ 3
```

(Note that the first body value cannot itself be a keyword.)

A `#:body` declaration works as an arbitrary *kw-formals*, not just a single variable like *b* in the above example. For example, to make the above `mathop` work only on three arguments that follow the keyword, use `(x y z)` instead of *b*:

```
(define/kw (mathop #:key [op +] #:body (x y z))
  (op x y z))
```

In general, `#:body` handling is compiled to a sub procedure using `lambda/kw`, so that a procedure can use more then one level of keyword arguments. For example:

```
(define/kw (mathop #:key [op +]
                   #:body (x y z #:key [convert values]))
  (op (convert x) (convert y) (convert z)))
(mathop #:op * 2 4 6 #:convert exact->inexact) --> 48.0
```

Obviously, nested keyword arguments works only when non-keyword arguments separate the sets.

Run-time errors during such calls report a mismatch for a procedure with a name that is based on the original name plus a `~body` suffix:

```
(mathop #:op * 2 4)
```

⇒ procedure mathop body: expects at least 3 arguments, given 2: 2 4

22.6 Mode Keywords

Finally, the argument list of a `lambda/kw` can contain keywords that serve as mode flags to control error reporting.

- `#:allow-other-keys` — the keyword–value sequence at the call site *can* include keywords that are not listed in the keyword part of the `lambda/kw` form.
- `#:forbid-other-keys` — the keyword–value sequence at the call site *cannot* include keywords that are not listed in the keyword part of the `lambda/kw` form, otherwise `exn:fail:contract` exception is raised.
- `#:allow-duplicate-keys` — the keyword–value list at the call site *can* include duplicate values associated with same keyword, the first one is used.
- `#:forbid-duplicate-keys` — the keyword–value list at the call site *cannot* include duplicate values for keywords, otherwise `exn:fail:contract` exception is raised. This restriction applies only to keywords that are listed in the keyword part of the `lambda/kw` form — if other keys are allowed, this restriction does not apply to them.
- `#:allow-body` — body arguments *can* be specified at the call site after all keyword–value pairs.
- `#:forbid-body` — body arguments *cannot* be specified at the call site after all keyword–value pairs.
- `#:allow-anything` — allows all of the above, and treat a single keyword at the end of an argument list as a `#:body`, a situation that is usually an error. When this is used and no rest-like arguments are used except `#:rest`, an extra loop is saved and calling the procedures is faster (around 20%).
- `#:forbid-anything` — forbids all of the above, ensuring that calls are as restricted as possible.

These mode markers are rarely needed, because the default modes are determined by the declared rest-like arguments:

- The default is to allow other keys if a `#:rest`, `#:other-keys+body`, `#:all-keys`, or `#:other-keys` variable is declared (and an `#:other-keys` declaration requires allowing other keys).
- The default is to allow duplicate keys if a `#:rest` or `#:all-keys` variable is declared;
- The default is to allow body arguments if a `#:rest`, `#:body`, or `#:other-keys+body` variable is declared (and a `#:body` argument requires allowing them).

Here's an alternate specification, which maps rest-like arguments to the behavior that they imply:

- `#:rest`: everything is allowed (a body, other keys, and duplicate keys);
- `#:other-keys+body`: other keys and body are allowed, but duplicates are not;
- `#:all-keys`: other keys and duplicate keys are allowed, but a body is not;
- `#:other-keys`: other keys must be allowed (on by default, cannot use with `#:forbid-other-keys`), and duplicate keys and body are not allowed;
- `#:body`: body must be allowed (on by default, cannot use with `#:forbid-body`) and other keys and duplicate keys and body are not allowed;
- Except for the previous two "must"s, defaults can be overridden by an explicit `#:allow-...` or a `#:forbid-...` mode.

22.7 Property Lists

`(keyword-get` *`args keyword`* [*`not-found-thunk`*]`)` PROCEDURE

Searches a list of keyword arguments (a "property list" or "plist" in Lisp jargon) for the given keyword, and returns the associated value. It is the facility that is used by `lambda/kw` to search for keyword values.

The *`args`* list is scanned from left to right, if the keyword is found, then the next value is returned. If the *`keyword`* was not found, then the *`not-found-thunk`* value is used to produce a value by applying it. If the *`keyword`* was not found, and *`not-found-thunk`* is not given, `#f` is returned. (No exception is raised if the *`args`* list is imbalanced, and the search stops at a non-keyword value.)

23. list.ss: List Utilities

To load: `(require (lib "list.ss"))`

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

`(assf f l)` PROCEDURE

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

`(cons? v)` PROCEDURE

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

`empty` EMPTY LIST

The empty list.

`(empty? v)` PROCEDURE

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

`(filter f l)` PROCEDURE

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

`(findf f l)` PROCEDURE

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

`(first l)` PROCEDURE

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

`(foldl f init l ···`[1]`)` PROCEDURE

Like `map`, `foldl` applies a procedure *f* to the elements of one or more lists. While `map` combines the return values into a list, `foldl` combines the return values in an arbitrary way that is determined by *f*. Unlike `foldr`, `foldl` processes *l* in constant space (plus the space for each call to *f*).

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

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

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

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

Like `foldl`, but the lists are traversed from right to left. Unlike `foldl`, `foldr` processes *l* in space proportional to the length of *l* (plus the space for each call to *f*).

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

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

(last-pair *list*) PROCEDURE

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

(memf *f l*) PROCEDURE

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

(sort *list less-than?*) PROCEDURE

Sorts *list* using the comparison procedure *less-than?*. This implementation is stable (i.e., if two elements in the input are "equal," their relative positions in the output will be the same).

(sort! *list less-than?*) PROCEDURE

The destructive version of `sort`. (Actually, `sort` is implemented by copying the list and using *sort!* on the copy.)

(merge-sorted-lists! *list1 list2 less-than?*) PROCEDURE

Merges the two sorted input lists by modifying cdr fields, to create a single sorted output list. The merged result is stable: equal items in both lists stay in the same order, and these in *list1* precede *list2*. This is used by *sort!*, but is also useful by itself.

(merge-sorted-lists *list1 list2 less-than?*) PROCEDURE

The non-destructive version of *merge-sorted-lists!*.

`(mergesort list less-than?)` PROCEDURE

Deprecated: use `sort` instead.

This is a different name for `sort`, provided for backward compatibility.

`(quicksort list less-than?)` PROCEDURE

Deprecated: use `sort` instead.

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

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

Returns *`list`* without the first instance of *`item`*, where an instance is found by comparing *`item`* to the list items using *`equal?`*. The default value for *`equal?`* is `equal?`. When *`equal?`* is invoked, *`item`* is the first argument.

`(remove* items list [equal?])` PROCEDURE

Like `remove`, except that the first argument is a list of items to remove instead of a single item, and all instances of these items are removed rather than just the first.

`(remq item list)` PROCEDURE

Calls `remove` with `eq?` as the comparison procedure.

`(remq* items list)` PROCEDURE

Calls `remove*` with `eq?` as the comparison procedure.

`(remv item list)` PROCEDURE

Calls `remove` with `eqv?` as the comparison procedure.

`(remv* items list)` PROCEDURE

Calls `remove*` with `eqv?` as the comparison procedure.

`(rest l)` PROCEDURE

Returns a list that contains all but the first element of the non-empty list *`l`*. (The `rest` procedure is a synonym for `cdr`.)

`(set-first! l v)` PROCEDURE

Destructively modifies *`l`* so that its first element is *`v`*. (The `set-first!` procedure is a synonym for `set-car!`.)

`(set-rest! l1 l2)` PROCEDURE

Destructively modifies *l1* so that the rest of the list (after the first element) is *l2*. (The `set-rest!` procedure is a synonym for `set-cdr!`.)

24. match.ss: Pattern Matching

To load: `(require (lib "match.ss"))`

This library provides functions for pattern-matching Scheme values. (This chapter adapted from Andrew K. Wright and Bruce Duba's original manual, entitled *Pattern Matching for Scheme*. The PLT Scheme port was originally done by Bruce Hauman and is maintained by Sam Tobin-Hochstadt.) The following forms are provided:

```
(match expr clause ...)
(match-lambda clause ...)
(match-lambda* clause ...)
(match-let ((pat expr) ...) expr ···¹)
(match-let* ((pat expr) ...) expr ···¹)
(match-letrec ((pat expr) ...) expr ···¹)
(match-let var ((pat expr) ...) expr ···¹)
(match-define pat expr)

clause is one of
 (pat expr ···¹)
 (pat (=> identifier) expr ···¹)
```

Figure 24.1 gives the full syntax for *pat* patterns. The next subsection describes the various patterns.

The `match-lambda` and `match-lambda*` forms are convenient combinations of `match` and `lambda`, and can be explained as follows:

```
 (match-lambda (pat expr ···¹) ...)  =  (lambda (x) (match x (pat expr ···¹) ...))
(match-lambda* (pat expr ···¹) ...)  =  (lambda x (match x (pat expr ···¹) ...))
```

where *x* is a unique variable. The `match-lambda` form is convenient when defining a single argument function that immediately destructures its argument. The `match-lambda*` form constructs a function that accepts any number of arguments; the patterns of `match-lambda*` should be lists.

The `match-let`, `match-let*`, `match-letrec`, and `match-define` forms generalize Scheme's `let`, `let*`, `letrec`, and `define` expressions to allow patterns in the binding position rather than just variables. For example, the following expression:

```
(match-let ([(x y z) (list 1 2 3)]) body)
```

binds `x` to `1`, `y` to `2`, and `z` to `3` in the body. These forms are convenient for destructuring the result of a function that returns multiple values. As usual for `letrec` and `define`, pattern variables bound by `match-letrec` and `match-define` should not be used in computing the bound value.

The `match`, `match-lambda`, and `match-lambda*` forms allow the optional syntax `(=> identifier)` between the pattern and the body of a clause. When the pattern match for such a clause succeeds, the *identifier* is bound to a *failure procedure* of zero arguments within the body. If this procedure is invoked, it jumps back to the pattern

pat	::=	*identifier*	Match anything, bind *identifier* as a variable
	|	_	Match anything
	|	*literal*	Match *literal*
	|	'*datum*	Match `equal?` *datum*
	|	'*symbol*	Match `equal?` *symbol* (special case of *datum*)
	|	(*lvp* ...)	Match sequence of *lvp*s
	|	(*lvp* ... . *pat*)	Match sequence of *lvp*s `cons`ed onto a *pat*
	|	#(*lvp* ...)	Match vector of *pat*s
	|	#&*pat*	Match boxed *pat*
	|	($ *struct-name* *pat* ...)	Match *struct-name* instance with matching fields
	|	(and *pat* ...)	Match when all *pat*s match
	|	(or *pat* ...)	Match when any *pat* match
	|	(not *pat* ...)	Match when no *pat* match
	|	(= *expr* *pat*)	Match when result of applying *expr* matches *pat*
	|	(? *expr* *pat* ...)	Match if *expr* is true and all *pat*s match
	|	(set! *identifier*)	Match anything, bind *identifier* as a setter
	|	(get! *identifier*)	Match anything, bind *identifier* as a getter
	|	`*qp*	Match quasi*pat*tern
literal	::=	#t	Match true
	|	#f	Match false
	|	*string*	Match `equal?` *string*
	|	*number*	Match `equal?` *number*
	|	*character*	Match `equal?` *character*
lvp	::=	*pat* *ooo*	Greedily match *pat* instances
	|	*pat*	Match *pat*
ooo	::=	...	Zero or more (where ... is a keyword)
	|	___	Zero or more
	|	..*k*	*k* or more, where *k* is a non-negative integer
	|	__*k*	*k* or more, where *k* is a non-negative integer
qp	::=	*literal*	Match *literal*
	|	*identifier*	Match `equal?` *symbol*
	|	(*qp* ...)	Match sequences of *qp*s
	|	(*qp* ... . *qp*)	Match sequence of *qp*s `cons`ed onto a *qp*
	|	(*qp* ... *qp* *ooo*)	Match *qp*s `cons`ed onto a repeated *qp*
	|	#(*qp* ...)	Match vector of *qp*s
	|	#&*qp*	Match boxed *qp*
	|	,*pat*	Match *pat*
	|	,@*pat*	Match *pat*, spliced

Figure 24.1: Pattern Syntax

matching expression, and resumes the matching process as if the pattern had failed to match. The body must not mutate the object being matched, otherwise unpredictable behavior may result.

24.1 Patterns

Figure 24.1 gives the full syntax for patterns. Explanations of these patterns follow.

- `identifier` (excluding the reserved names ?, =, $, _, `and`, `or`, `not`, `set!`, `get!`, `...`, and `..k` for non-negative integers *k*) — matches anything, and binds a variable of this name to the matching value in the body.
- _ — matches anything, without binding any variables.
- `#t`, `#f`, `string`, `number`, `character`, `'s-expression` — constant patterns that match themselves (i.e., the corresponding value must be `equal?` to the pattern).
- (`pat`$_1$ ··· `pat`$_n$) matches a proper list of *n* elements that match `pat`$_1$ through `pat`$_n$.
- (`lvp`$_1$ ··· `lvp`$_n$) generalizes the preceding pattern, where each `lvp` corresponds to a "spliced" list of greedy matches.

  For example, (`pat`$_1$ ··· `pat`$_n$ `pat`$_{n+1}$ `...`) matches a proper list of *n* or more elements, where each element past the *n*th matches `pat`$_{n+1}$. Each pattern variable in `pat`$_{n+1}$ is bound to a list of the matching values. For example, the expression:

  ```
  (match '(let ([x 1][y 2]) z)
    [('let ((binding vals) ...) exp) expr ···1])
  ```

  binds `binding` to the list `'(x y)`, `vals` to the list `'(1 2)`, and `exp` to `'z` in the body of the `match`-expression. For the special case where `pat`$_{n+1}$ is a pattern variable, the list bound to that variable may share with the matched value.

  Instead of `...` or ___ (which are equivalent), `..k` or __k can be used to match a sequence that is at least *k* long. The pattern keywords `..0`, `...`, and ___ are equivalent.
- (`pat`$_1$ ··· `pat`$_n$. `pat`$_{n+1}$) — matches a (possibly improper) list of at least *n* elements that ends in something matching `pat`$_{n+1}$.
- (`lvp`$_1$ ··· `lvp`$_n$. `pat`$_{n+1}$) — generalizes the preceding pattern with greedy-sequence `lvp`s.
- `#(pat`$_1$ ··· `pat`$_n$`)` — matches a vector of length *n*, whose elements match *pat*$_1$ through *pat*$_n$. The generalization to `lvp`s matches consecutive elements of the vector greedily.
- `#&pat` — matches a box containing something matching `pat`.
- (`$ struct-name pat`$_1$ ··· `pat`$_n$) — matches an instance of a structure type `struct-name`, where the instance contains *n* fields.

  Usually, `struct-name` is defined with `define-struct`. More generally, `struct-name` must be bound to expansion-time information for a structure type (see §12.6.4 in *PLT MzScheme: Language Manual*), where the information includes at least a predicate binding and some field accessor bindings (and `pat`$_1$ through `pat`$_n$ correspond to the provided accessors). In particular, a module import or a `unit/sig` import with a signature containing a `struct` declaration (see §50.2) can provide the structure type information.
- `(= field pat)` — applies `field` to the object being matched and uses `pat` to match the extracted object. The `field` subexpression may be any expression, but is often useful as a struct selector.
- (`and pat`$_1$ ··· `pat`$_n$) — matches if all of the subpatterns match. This pattern is often used as `(and x pat)` to bind `x` to to the entire value that matches `pat`.

- `(or` *pat*$_1$ ⋯ *pat*$_n$`)` — matches if any of the subpatterns match. At least one subpattern must be present. All subpatterns must bind the same set of pattern variables.

- `(not` *pat*$_1$ ⋯ *pat*$_n$`)` — matches if none of the subpatterns match. The subpatterns may not bind any pattern variables.

- `(?` *predicate-expr pat*$_1$ ⋯ *pat*$_n$`)` — In this pattern, *predicate-expr* must be an expression evaluating to a single argument function. This pattern matches if *predicate-expr* applied to the corresponding value is true, and the subpatterns *pat*$_1$ through *pat*$_n$ all match. The *predicate-expr* should not have side effects, as the code generated by the pattern matcher may invoke predicates repeatedly in any order. The *predicate-expr* expression is bound in the same scope as the match expression, so free variables in *predicate-expr* are not bound by pattern variables.

- `(set!` *identifier*`)` — matches anything, and binds *identifier* to a procedure of one argument that mutates the corresponding field of the matching value. This pattern must be nested within a pair, vector, box, or structure pattern. For example, the expression:

  ```
  (define x (list 1 (list 2 3)))
  (match x [(_ (_ (set! setit)))  (setit 4)])
  ```

  mutates the *cadadr* of *x* to `4`, so that `x` is `'(1 (2 4))`.

- `(get!` *identifier*`)` — matches anything, and binds *identifier* to a procedure of zero arguments that accesses the corresponding field of the matching value. This pattern is the complement to `set!`. As with `set!`, this pattern must be nested within a pair, vector, box, or structure pattern.

- `‘`*quasipattern* — introduces a quasipattern, in which identifiers are considered to be symbolic constants. Like Scheme's quasiquote for data, `unquote` (`,`) and `unquote-splicing` (`,@`) escape back to normal patterns.

If no clause matches the value, an `exn:misc:match` exception is raised.

24.2 Extending Match

There are two ways to extend or alter the behavior of match.

The `match-equality-test` parameter controls the behavior of non-linear patterns:

`(match-equality-test` [*expr*]`)` PROCEDURE

When a variable appears more than once in a pattern, the values matched by each instance are constrained to be the same in the sense of the runtime value of `match-equality-test`. The default value of this parameter is `equal?`.

The `define-match-expander` form extends the syntax of match patterns:

`(define-match-expander` *id proc-expr*`)` SYNTAX

`(define-match-expander` *id proc-expr proc-expr*`)` SYNTAX

`(define-match-expander` *id proc-expr proc-expr proc-expr*`)` SYNTAX

This form binds an identifier to a pattern transformer.

The first *proc-expr* subexpression must evaluate to a transformer that produces a pattern in the syntax of Chapter 31. Whenever *id* appears as the beginning of a pattern in a the context of the pattern matching forms defined in Chapter 31, 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.

If a second *proc-expr* subexpression is provided, it must produce a similar transformer, but in the context of patterns written in the syntax of the current chapter.

A transformer produced by a third *proc-expr* subexpression is used when the *id* keyword is used in a traditional macro use context. In this way, *id* can be given meaning both inside and outside patterns.

24.3 Examples

This section illustrates the convenience of pattern matching with some examples. The following function recognizes some s-expressions that represent the standard Y operator:

```
(define Y?
  (match-lambda
    [('lambda (f1)
       ('lambda (y1)
         ((('lambda (x1) (f2 ('lambda (z1) ((x2 x3) z2))))
           ('lambda (a1) (f3 ('lambda (b1) ((a2 a3) b2)))))
          y2)))
     (and (symbol? f1) (symbol? y1) (symbol? x1) (symbol? z1) (symbol? a1) (symbol? b1)
          (eq? f1 f2) (eq? f1 f3) (eq? y1 y2)
          (eq? x1 x2) (eq? x1 x3) (eq? z1 z2)
          (eq? a1 a2) (eq? a1 a3) (eq? b1 b2))]
    [_ #f]))
```

Writing an equivalent piece of code in raw Scheme is tedious.

The following code defines abstract syntax for a subset of Scheme, a parser into this abstract syntax, and an unparser.

```
(define-struct Lam (args body))
(define-struct Var (s))
(define-struct Const (n))
(define-struct App (fun args))

(define parse
  (match-lambda
    [(and s (? symbol?) (not 'lambda))
     (make-Var s)]
    [(? number? n)
     (make-Const n)]
    [('lambda (and args ((? symbol?) ...) (not (? repeats?))) body)
     (make-Lam args (parse body))]
    [(f args ...)
     (make-App
       (parse f)
       (map parse args))]
    [x (error 'syntax "invalid expression")]))

(define repeats?
  (lambda (l)
```

```
    (and (not (null? l))
         (or (memq (car l) (cdr l)) (repeats? (cdr l))))))

(define unparse
  (match-lambda
    [($ Var s) s]
    [($ Const n) n]
    [($ Lam args body) `(lambda ,args ,(unparse body))]
    [($ App f args) `(,(unparse f) ,@(map unparse args))]))
```

With pattern matching, it is easy to ensure that the parser rejects *all* incorrectly formed inputs with an error message.

With `match-define`, it is easy to define several procedures that share a hidden variable. The following code defines three procedures, *inc*, *value*, and *reset*, that manipulate a hidden counter variable:

```
(match-define (inc value reset)
  (let ([val 0])
    (list
      (lambda () (set! val (add1 val)))
      (lambda () val)
      (lambda () (set! val 0)))))
```

Although this example is not recursive, the bodies could recursively refer to each other. The following code illustrates the creation of a match-expander that works for both `(lib "match.ss")` and `(lib "plt-match.ss")` syntax.

```
(require (prefix plt: (lib "plt-match.ss")))
(define-struct point (x y))
(define-match-expander Point
  (lambda (stx)
    (syntax-case stx ()
      ((Point a b) #'(struct point (a b)))))
  (lambda (stx)
    (syntax-case stx ()
      ((Point a b) #'($ point a b))))
  (lambda (stx)
    (syntax-case stx ()
      ((Point a b) #'(make-point a b)))))

(define p (Point 3 4))

(match p
   ((Point x y) (+ x y)))
;; => 7
(plt:match p
   ((Point x y) (* x y)))
;; => 12
```

25. math.ss: Math

To load: `(require (lib "math.ss"))`

`(conjugate z)` PROCEDURE

Returns the complex conjugate of *z*.

`(cosh z)` PROCEDURE

Returns the hyperbolic cosine of *z*.

`e` NUMBER

Approximation of Euler's number, equivalent to `(exp 1.0)`.

`pi` NUMBER

Approximation of π, equivalent to `(atan 0.0 -1.0)`.

`(sinh z)` PROCEDURE

Returns the hyperbolic sine of *z*.

`(sgn n)` PROCEDURE

Returns `1` if *n* is positive, `-1` if *n* is negative, and `0` otherwise. If *n* is exact, the result is exact, otherwise the result is inexact.

`(sqr z)` PROCEDURE

Returns `(* z z))`.

26. md5.ss: MD5 Message Digest

To load: `(require (lib "md5.ss"))`

`(md5 `*`input-port`*`)` PROCEDURE

`(md5 `*`bytes`*`)` PROCEDURE

Produces a byte string containing 32 hexadecimal digits (lowercase) that is the MD5 hash of the given input-port or byte string. For example, `(md5 #"abc")` produces `#"900150983cd24fb0d6963f7d28e17f72"`.

27. `os.ss`: System Utilities

To load: `(require (lib "os.ss"))`

`(gethostname)` PROCEDURE

Returns a string for the current machine's hostname (including its domain).

`(getpid)` PROCEDURE

Returns an exact integer identifying the current process within the operating system.

`(truncate-file` *`path`* [*`size-k`*]`)` PROCEDURE

Truncates or extends the given file so that it is *`size-k`* bytes long, where *`size-k`* defaults to `0`. If the file does not exist, or if the process does not have sufficient privilege to truncate the file, the `exn:fail` exception is raised.

WARNING: under Unix, the implementation assumes that the system's `ftruncate` function accepts a `long long` second argument.

28. package.ss: Local-Definition Scope Control

To load: `(require (lib "package.ss"))`

The `package` form provides fine-grained control over binding visibility. A package is an expansion-time entity only; it has no run-time identity.The `package` and `open` constructs correspond to `module` and `import` in Chez Scheme. The `package*` and `open*` constructs correspond to structures in Standard ML (without types).

`(package` *`name`* `(`*`export`* `···)` *`body-expr-or-defn`* `···`[1]`)` SYNTAX

`(package` *`name all-defined body-expr-or-defn`* `···`[1]`)` SYNTAX

Defines *`name`* (in any definition context) to a compile-time package description, much in the way that `(define-syntax` *`a`* `(syntax-rules ...))` binds *`a`* to a syntax expander, or `(define-struct` *`a`* `())` binds *`a`* to a compile-time structure type description.

Each *`export`* must be an identifier that is defined within the package body. The `all-defined` variant is shorthand for listing all identifiers that are defined in the package body.

Although `package` does not introduce a new binding scope, it hides all of the definitions in its body from definitions and expressions that are outside the package. The exported definitions become visible only when the package is opened with forms such as `open`.

Each *`body-expr-or-defn`* can be a definition or expression. Each defined identifier is visible in the entire package body, except definitions introduced by `define*`, `define*-syntax`, `define*-values`, `define*-syntaxes`, `open*`, `package*`, or `define*-dot`. The ∗ forms expose identifiers to expressions and definitions that appear later in the package body, only, much like the sequential binding of `let*`. As with `let*`, an identifier can be defined multiple times within the package using ∗ forms; if such an identifier is exported, the export corresponds to the last definition. For any other form of definition, the identifiers that it defines must be defined only once within the package.

When used in an internal-definition context (see §2.8.5 in *PLT MzScheme: Language Manual*), *`name`* is immediately available for use with other forms, such as `open`, in the same internal-definition sequence.

For example, see `open`, below.

`(package*` *`name`* `(`*`export`* `···)` *`body-expr-or-defn`* `···`[1]`)` SYNTAX

`(package*` *`name all-defined body-expr-or-defn`* `···`[1]`)` SYNTAX

Like `package`, but within a package body, the package name is visible only to later definitions and expressions.

`(open` *`name`* `···`[1]`)` SYNTAX

If a single *`name`* is provided, it must be defined as a package, and the package's exports are exposed in the definition context of the `open` declaration.

The `open` form acts like a definition form, in that it introduces bindings in a definition context, and such bindings can be exported from a package (even using `all-defined`). More precisely, however, `open` exposes bindings hidden by a package, rather than introducing identifiers. This exposure overrides any identifier that would shadow the binding (were it not hidden by the package in the first place).

If multiple *`name`*s are provided, the first name must correspond to a defined package, the second must correspond to a package exported from the first, and so on. Only the package corresponding to the last name is opened into the `open`'s definition context.

Examples:

```
(package p (f)
  (define (f a) (+ a x))
  (define x 1))
(f 0) ; ⇒ error: reference to undefined identifier f
(let ([p 5])
  (open p) ...)  ; ⇒ error: p is not a package name
(open p)
(f 0) ; ⇒ 1

(let ([f (lambda (x) x)])
  (open p)
  (f 0)) ; ⇒ 1

(let ([x 2])
  (open p)
  (f 0)) ; ⇒ 1

(package p (p2)
  (package p2 (f)
    (define (f a) (− a x)))
  (define x 2))
(open p p2)
(f 3) ; ⇒ 1

(package p (p2)
  (package p2 (f)
    (define (f a) (− a x)))
  (define x 2))
(open p p2)
(f 3) ; ⇒ 1

(package p1 (x f1 p2 p3)
  (define x 1)
  (define (f1) x)
  (package p2 (x f2)
    (define x 2)
    (define (f2) x))
  (package p3 (f3)
    (open p2)
    (define (f3) x)))
(open p1)
x ; ⇒ 1
```

```
(f1) ; ⇒ 1
(open p2)
x ; ⇒ 2
(f2) ; ⇒ 2
(open p3)
(f3) ; ⇒ 2
(open p1)
x ; ⇒ 1

(define-syntax package2
  (syntax-rules ()
   [(_ name id def)
    (package name (id foo)
      def
      (define foo 3))]))
(let ()
  (package2 p foo (define foo 1))
  (open p)
  foo) ; ⇒ 1
(let ()
  (package2 p bar (define bar 1))
  (open p)
  foo) ; ⇒ error: reference to undefined identifier foo

(define-syntax open2
  (syntax-rules ()
   [(_ name) (open name)]))
(let ()
  (package p (x) (define x 1))
  (open2 p)
  x) ; ⇒ 1

(define-syntax package3
  (syntax-rules ()
   [(_ name id)
    (package name (id foo)
      (define (id) foo)
      (define foo 3))]))
(let ([foo 17])
  (package3 p f)
  (open p)
  (+ foo (f))) ; ⇒ 20
```

`(open* name ...`^{1}`)` SYNTAX

Like `open`, but within a package, the opened package's exports are exposed only to later definitions and expressions.

`(dot name ...`^{1}`export)` SYNTAX

Equivalent to `(let () (open name ...`^{1}`) export)` when *export* is exported from the package selected by *name* ...^{1}.

Example:

```
(package p (x)
  (define x 1))
(+ 2 (dot p x)) ; ⇒ 3
```

(define-dot *variable name* ···[1]) SYNTAX

Defines *variable* as an alias for the package export selected by *name* ···[1]. The export can correspond to a nested package, in which case the alias is available for immediate use in forms like `open` or `define-dot`.

(define∗-dot *variable name* ···[1]) SYNTAX

Like `define-dot`, but within a package, the alias applies only to later definitions and expressions.

(rename-potential-package *old-name new-name*) SYNTAX

Introduces *old-name* as an alias for *new-name*.

Although `make-rename-transformer` (see §12.6 in *PLT MzScheme: Language Manual*) can be used to create an alias for a package name, only an alias created by `rename-potential-package`, `define-dot`, or `define∗-dot` is available for immediate use by forms such as `open`.

(define∗ *variable expr*) SYNTAX

(define∗ (*header* . *formals*) *expr* ···[1]) SYNTAX

(define∗-syntax *variable expr*) SYNTAX

(define∗-syntax (*header* . *formals*) *expr* ···[1]) SYNTAX

(define∗-values (*variable* ···) *expr*) SYNTAX

(define∗-syntaxes (*variable* ···) *expr*) SYNTAX

(rename∗-potential-package *old-name new-name*) SYNTAX

Like `define`, etc., but when used in a package, they define identifiers that are visible only to later definitions and expressions.

(package/derived *expr name* (*export* ···) *body-expr-or-defn* ···[1]) SYNTAX

(package/derived *expr name all-defined body-expr-or-defn* ···[1]) SYNTAX

Like `package`, but syntax errors (such as duplicate definitions) are reported as originating from *expr*.

This form is useful for writing macros that expand to `package` and rely on the syntax checks of the `package` transformer, but where syntax errors should be reported in terms of the source expression or declaration.

(open/derived *expr orig-name name* ···[1]) SYNTAX (open∗/derived *expr orig-name name* $\cdots^1$) SYNTAX

Like open and open∗, but syntax errors (such as duplicate definitions) are reported as originating from *expr*. Furthermore, if *name* is not a package name, the error message reports that *orig-name* is not defined as a package.

29. pconvert.ss: Converted Printing

To load: `(require (lib "pconvert.ss"))`

This library defines routines for printing Scheme values as `eval`uable S-expressions rather than `read`able S-expressions. The `print-convert` procedure does not print values; rather, it converts a Scheme value into another Scheme value such that the new value pretty-prints as a Scheme expression that evaluates to the original value. For example, `(pretty-print (print-convert '(9 ,(box 5) #(6 7))))` prints the literal expression `(list 9 (box 5) (vector 6 7))` to the current output port.

To install print converting into the `read-eval-print` loop, require **pconvert.ss** and call the procedure `install-converting-printer`.

In addition to `print-convert`, this library provides `print-convert`, `build-share`, `get-shared`, and `print-convert-expr`. The last three are used to convert sub-expressions of a larger expression (potentially with shared structure).

See also `prop:print-convert-constructor-name` in §30.

`(abbreviate-cons-as-list [abbreviate?])` PROCEDURE

Parameter that controls how lists are represented with constructor-style conversion. If the parameter's value is `#t`, lists are represented using `list`. Otherwise, lists are represented using `cons`. The initial value of the parameter is `#t`.

`(booleans-as-true/false [use-name?])` PROCEDURE

Parameter that controls how `#t` and `#f` are represented. If the parameter's value is `#t`, then `#t` is represented as `true` and `#f` is represented as `false`. The initial value of the parameter is `#t`.

`(use-named/undefined-handler [use-handler])` PROCEDURE

Parameter for a procedure that controls how values that have inferred names are represented. The procedure is passed a value. If the parameter returns `#t`, the procedure associated with *`named/undefined-handler`* is invoked to render that value. Only values that have inferred names but are not defined at the top-level are used with this handler.

The initial value of the parameter is `(lambda (x) #f)`.

`(named/undefined-handler [use-handler])` PROCEDURE

Parameter for a procedure that controls how values that have inferred names are represented. The procedure is called only if *`use-named/undefined-handler`* returns true for some value. In that case, the procedure is passed that same value, and the result of the parameter is used as the representation for the value.

The initial value of the parameter is `(lambda (x) #f)`.

`(build-share v)` PROCEDURE

Takes a value and computes sharing information used for representing the value as an expression. The return value is an opaque structure that can be passed back into `get-shared` or `print-convert-expr`.

`(constructor-style-printing [use-constructors?])` PROCEDURE

Parameter that controls how values are represented after conversion. If this parameter is `#t`, then constructors are used, e.g., pair containing 1 and 2 is represented as `(cons 1 2)`. Otherwise, quasiquote-style syntax is used, e.g. the pair containing 1 and 2 is represented as `‘(1 . 2)`. The initial value of the parameter is `#f`.

See also `quasi-read-style-printing`, and see `prop:print-convert-constructor-name` in §30.

`(current-build-share-hook [hook])` PROCEDURE

Parameter that sets a procedure used by `print-convert` and `build-share` to assemble sharing information. The procedure *hook* takes three arguments: a value *v*, a procedure *basic-share*, and a procedure *sub-share*; the return value is ignored. The *basic-share* procedure takes *v* and performs the built-in sharing analysis, while the *sub-share* procedure takes a component of *v* ands analyzes it. These procedures return void; sharing information is accumulated as values are passed to *basic-share* and *sub-share*.

A `current-build-share-hook` procedure usually works together with a `current-print-convert-hook` procedure.

`(current-build-share-name-hook [hook])` PROCEDURE

Parameter that sets a procedure used by `print-convert` and `build-share` to generate a new name for a shared value. The *hook* procedure takes a single value and returns a symbol for the value's name. If *hook* returns `#f`, a name is generated using the form "`-n-`" (where *n* is an integer).

`(current-print-convert-hook [hook])` PROCEDURE

Parameter that sets a procedure used by `print-convert` and `print-convert-expr` to convert values. The procedure *hook* takes three arguments — a value *v*, a procedure *basic-convert*, and a procedure *sub-convert* — and returns the converted representation of *v*. The *basic-convert* procedure takes *v* and returns the default conversion, while the *sub-convert* procedure takes a component of *v* and returns its conversion.

A `current-print-convert-hook` procedure usually works together with a `current-build-share-hook` procedure.

`(current-read-eval-convert-print-prompt [str])` PROCEDURE

Parameter that sets the prompt used by `install-converting-printer`. The initial value is `"|- "`.

`(get-shared share-info [cycles-only?])` PROCEDURE

The *share-info* value must be a result from `build-share`. The procedure returns a list matching variables to shared values within the value passed to `build-share`. For example,

```
(get-shared (build-share (shared ([a (cons 1 b)][b (cons 2 a)]) a)))
```

might return the list

```
((-1- (cons 1 -2-)) (-2- (cons 2 -1-)))
```

The default value for `cycles-only?` is `#f`; if it is not `#f`, `get-shared` returns only information about cycles.

`(install-converting-printer)` PROCEDURE

Sets the current print handler to print values using `print-convert`. The current read handler is also set to use the prompt returned by `current-read-eval-convert-print-prompt`.

`(print-convert v [cycles-only?])` PROCEDURE

Converts the value `v`. If `cycles-only?` is not `#f`, then only circular objects are included in the output. The default value of `cycles-only?` is the value of `(show-sharing)`.

`(print-convert-expr share-info v unroll-once?)` PROCEDURE

Converts the value `v` using sharing information `share-info` previously returned by `build-share` for a value containing `v`. If the most recent call to `get-shared` with `share-info` requested information only for cycles, then `print-convert-expr` will only display sharing among values for cycles, rather than showing all value sharing.

The `unroll-once?` argument is used if `v` is a shared value in `share-info`. In this case, if `unroll-once?` is `#f`, then the return value will be a shared-value identifier; otherwise, the returned value shows the internal structure of `v` (using shared value identifiers within `v`'s immediate structure as appropriate).

`(quasi-read-style-printing [on?])` PROCEDURE

Parameter that controls how vectors and boxes are represented after conversion when the value of `constructor-style-printing` is `#f`. If `quasi-read-style-printing` is set to `#f`, then boxes and vectors are unquoted and represented using constructors. For example, the list of a box containing the number 1 and a vector containing the number 1 is represented as `‘(,(box 1) ,(vector 1))`. If the parameter is `#t`, then `#&` and `#()` are used, e.g., `‘(#&1 #(1))`. The initial value of the parameter is `#t`.

`(show-sharing [show?])` PROCEDURE

Parameter that determines whether sub-value sharing is conserved (and shown) in the converted output by default. The initial value of the parameter is `#t`.

`(whole/fractional-exact-numbers [whole-frac?])` PROCEDURE

Parameter that controls how exact, non-integer numbers are converted when the numerator is greater than the denominator. If the parameter's value is `#t`, the number is converted to the form `(+ integer fraction)` (i.e., a list containing `'+`, an exact integer, and an exact rational less than `1` and greater than `-1`). The initial value of the parameter is `#f`.

30. pconvert-prop.ss: Converted Printing Property

To load: `(require (lib "pconvert-prop.ss"))`

`prop:print-convert-constructor-name` PROPERTY

`(print-convert-named-constructor? v)` PROCEDURE

`(print-convert-constructor-name v)` PROCEDURE

The `prop:print-convert-constructor-name` property can be given a symbol value for a structure type. In that case, for constructor-style print conversion via `print-convert` (see §29), instances of the structure are shown using the symbol as the constructor name. Otherwise, the constructor name is determined by prefixing `make-` onto the result of `object-name`.

The `print-convert-named-constructor?` predicate recognizes instances of structure types that have the `prop:print-convert-constructor-name` property, and `print-convert-constructor-name` extracts the property value.

31. plt-match.ss: Pattern Matching

To load: `(require (lib "plt-match.ss"))`

This library provide a pattern matcher just like Chapter 24, but with an improved syntax for patterns. This pattern syntax uses keywords for each of the different pattern matches, making the syntax both extensible and more clear. It also provides extensions that are unavailable in **match.ss**.

The only difference between **plt-match.ss** and **match.ss** is the syntax of the patterns and the set of available patterns. The forms where the patterns may appear are identical.

Figure 31.1 gives the full syntax for patterns.

```
pat      ::= identifier [not ooo]                 Match anything, bind identifier as a variable
           |  _                                    Match anything
           |  literal                              Match literal
           |  'datum                               Match equal? datum
           |  'symbol                              Match equal? symbol (special case of datum)
           |  (list lvp ...)                       Match sequence of lvps
           |  (list-rest lvp ... pat)              Match sequence of lvps consed onto a pat
           |  (list-no-order pat ... lvp)          Match arguments in a list in any order
           |  (vector lvp ... lvp)                 Match vector of pats
           |  (hash-table (pat pat) ...)           Match hash table mapping pats to pats
           |  (hash-table (pat pat) ... ooo)       Match hash table mapping pats to pats
           |  (box pat)                            Match boxed pat
           |  (struct struct-name (pat ...))       Match struct-name instance with matching fields
           |  (regexp rx-expr)                     Match string using (regexp-match rx-expr ...)
           |  (regexp rx-expr pat)                 Match string to rx-expr, pat matches regexp result
           |  (pregexp prx-expr)                   Match string using (pregexp-match prx-expr ...)
           |  (pregexp prx-expr pat)               Match string to prx-expr, pat matches pregexp result
           |  (and pat ...)                        Match when all pats match
           |  (or pat ...)                         Match when any pat match
           |  (not pat ...)                        Match when no pat match
           |  (app expr pat)                       Match when result of applying expr matches pat
           |  (? expr pat ...)                     Match if expr is true and all pats match
           |  (set! identifier)                    Match anything, bind identifier as a setter
           |  (get! identifier)                    Match anything, bind identifier as a getter
           |  `qp                                  Match a quasipattern
literal  ::= ()                                   Match the empty list
           |  #t                                   Match true
           |  #f                                   Match false
           |  string                               Match equal? string
           |  number                               Match equal? number
           |  character                            Match equal? character
lvp      ::= pat ooo                              Greedily match pat instances
           |  pat                                  Match pat
ooo      ::= ...                                  Zero or more (where ... is a keyword)
           |  ___                                  Zero or more
           |  ..k                                  k or more, where k is a non-negative integer
           |  __k                                  k or more, where k is a non-negative integer
qp       ::= literal                              Match literal
           |  identifier                           Match equal? symbol
           |  (qp ...)                             Match sequences of qps
           |  (qp ... . qp)                        Match sequence of qps consed onto a qp
           |  (qp ... ooo)                         Match qps consed onto a repeated qp
           |  #(qp ...)                            Match vector of qps
           |  #&qp                                 Match boxed qp
           |  ,pat                                 Match pat
           |  ,@(list lvp ...)                     Match lvp sequence, spliced
           |  ,@(list-rest lvp ... pat)            Match lvp sequence plus pat, spliced
           |  ,@'qp                                Match list-matching qp, spliced
```

Figure 31.1: Pattern Syntax

32. port.ss: Port Utilities

To load: `(require (lib "port.ss"))`

`(convert-stream` *`from-encoding-string input-port from-encoding-string output-port`*`)`
PROCEDURE

Reads data from *`input-port`*, converts it using `(bytes-open-converter` *`from-encoding-string to-encoding-string`*`)` and writes the converted bytes to *`output-port`*. The `convert-stream` procedure returns after reaching `eof` in *`input-port`*.

See §3.6 in *PLT MzScheme: Language Manual* for more information on `bytes-open-converter`. If opening the converter fails, the `exn:fail` exception is raised. Similarly, if a conversion error occurs at any point while reading *`input-port`*, then `exn:fail` exception is raised.

`(copy-port` *`input-port output-port`* `···`[1]`)` PROCEDURE

Reads data from *`input-port`* and writes it back out to *`output-port`*, returning when *`input-port`* produces `eof`. The copy is efficient, and it is without significant buffer delays (i.e., a byte that becomes available on *`input-port`* is immediately transferred to *`output-port`*, even if future reads on *`input-port`* must block). If *`input-port`* produces a special non-byte value, it is transferred to *`output-port`* using `write-special`.

This function is often called from a "background" thread to continuously pump data from one stream to another.

If multiple *`output-port`*s are provided, case data from *`input-port`* is written to every *`output-port`*. The different *`output-port`*s block output to each other, because each quantum of data read from *`input-port`* is written completely to one *`output-port`* before moving to the next *`output-port`*. The *`output-port`*s are written in the provided order, so non-blocking ports (e.g., to a file) should be placed first in the argument list.

`(input-port-append` *`close-at-eof? input-port`* `···)` PROCEDURE

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-proc optional-fast-peek-proc close-proc`*`)`
PROCEDURE

Similar to `make-input-port`, but the given *`read`* procedure must never block, and if it returns an event, the event's value must be `0`. The resulting port's peek operation is implemented automatically (in terms of *`read-proc`*) 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-proc* and *close-proc* procedures are the same as for `make-input-port`. The *optional-fast-peek-proc* 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 *optional-fast-peek-proc* can either implement the requested peek, or it can dispatch to its third argument to implement the peek. The *optional-fast-peek-proc* is not used when a peek request has an associated progress event.

`(make-limited-input-port` *input-port limit-k* [*close-orig?*]`)` PROCEDURE

Returns a port whose content is drawn from *input-port*, but where an end-of-file is reported after *limit-k* 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 *input-port* only when they are consumed from the returned port. In particular, peeking into the returned port peeks into the original port.

If *input-port* is used directly while the resulting port is also used, then the *limit-k* bytes provided by the port need not be contiguous parts of the original port's stream.

`(make-pipe-with-specials` [*limit-k in-name-v out-name-v*]`)` PROCEDURE

Returns two ports: an input port and an output port. The pipes 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-k* 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-k* is reached.

The optional *in-name-v* and *out-name-v* arguments determine the names of the result ports, and both names default to `'pipe`.

`(merge-input` *a-input-port b-input-port* [*limit-k*]`)` PROCEDURE

Accepts two input ports and returns a new input port. The new port merges the data from two original ports, so data can be read from the new port whenever it is available from either original port. The data from the original ports are interleaved. When EOF has been read from an original port, it no longer contributes characters to the new port. After EOF has been read from both original ports, the new port returns EOF. Closing the merged port does not close the original ports.

The optional *limit-k* argument limits the number of bytes to be buffered from *a-input-port* and *b-input-port*, so that the merge process does not advance arbitrarily beyond the rate of consumption of the merged data. A `#f` value disables the limit; the default is `4096`. As for `make-pipe-with-specials`, *limit-k* 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?*]`)` PROCEDURE

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, and it defaults to `'nowhere`. If the *special-ok?* argument is true (the default), then the resulting port supports `write-special`, otherwise it does not.

`(peeking-input-port` *`input-port`* [*`name skip-k`*]`)` PROCEDURE

Returns an input port whose content is determined by peeking into *`input-port`*. In other words, the resulting port contains an internal skip count, and each read of the port peeks into *`input-port`* 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, and it defaults to `(object-name` *`input-port`*`)`. The *`skip-k`* argument is the port initial skip count, and it defaults to `0`.

`(eof-evt` *`input-port`*`)` PROCEDURE

Returns a synchronizable event (see §7.7 in *PLT MzScheme: Language Manual*) is that is ready when *`input-port`* produces an `eof`. If *`input-port`* 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 input-port`*`)` PROCEDURE

Returns a synchronizable event (see §7.7 in *PLT MzScheme: Language Manual*) is that is ready when *`k`* bytes can be read from *`input-port`*, or when an end-of-file is encountered in *`input-port`*. 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 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 *`input-port`* must support progress events, and it must not produce a special non-byte value during the read attempt.

`(read-bytes!-evt` *`mutable-bytes input-port`*`)` PROCEDURE

Like `read-bytes-evt`, except that the read bytes are placed into *`mutable-bytes`*, and the number of bytes to read corresponds to `(bytes-length` *`mutable-bytes`*`)`. The event's result is either `eof` or the number of read bytes.

The *`mutable-bytes`* string may be mutated any time after the first synchronization attempt on the event. If the event is not synchronized multiple times concurrently, *`mutable-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` *`mutable-bytes input-port`*`)` PROCEDURE

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 input-port`*`)` PROCEDURE

Like `read-bytes-evt`, but for character strings instead of byte strings.

`(read-string!-evt` *`mutable-string input-port`*`)` PROCEDURE

Like `read-bytes!-evt`, but for a character string instead of a byte string.

`(read-line-evt` *`input-port`* [*`mode-symbol`*]`)` PROCEDURE

Returns a synchronizable event (see §7.7 in *PLT MzScheme: Language Manual*) that is ready when a line of characters or end-of-file can be read from *`inport`*. The meaning an interpretation of *`mode-symbol`* is the same as for `read-line` (see §11.2.1 in *PLT MzScheme: Language Manual*). 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` *`input-port`* [*`mode-symbol`*]`)` PROCEDURE

Like `read-line`, but returns a byte string instead of a string.

`(peek-bytes-evt` *`k skip-k progress-evt input-port`*`)` PROCEDURE

`(peek-bytes-bytes!-evt` *`mutable-bytes skip-k progress-evt input-port`*`)` PROCEDURE

`(peek-bytes-avail!-evt` *`mutable-bytes skip-k progress-evt input-port`*`)` PROCEDURE

`(peek-string-evt` *`k input-port`*`)` PROCEDURE

`(peek-string!-evt` *`mutable-string input-port`*`)` PROCEDURE

Like the `read-...-evt` functions, but for peeking. The *`skip-k`* argument indicates the number of bytes to skip, and *`progress-evt`* indicates an event that effectively cancels the peek (so that the event never becomes ready). The *`progress-evt`* argument can be `#f`, in which case the event is never cancelled.

`(reencode-input-port` *`input-port encoding-str`* [*`error-bytes close? name-v`*]`)` PROCEDURE

Produces an input port that draws bytes from *`input-port`*, but converts the byte stream using `(bytes-open-converter` *`encoding-str`* `"UTF-8")`.

If *`error-bytes`* is provided and not `#f`, then the given byte sequence is used in place of bytes from *`input-port`* that trigger conversion errors. Otherwise, if a conversion is encountered, the `exn:fail` exception is raised.

If *`close?`* is true, then closing the result input port also closes *`input-port`*.

If *`name-v`* is provided, it is used as the name of the result input port, otherwise the port is named by `(object-name` *`input-port`*`)`.

In non-buffered mode, the resulting input port attempts to draw bytes from *`input-port`* 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` *`output-port encoding-str`* [*`error-bytes close? name-v buffer-sym`*]`)`

PROCEDURE

Produces an output port that direct bytes to *output-port*, but converts its byte stream using (bytes-open-converter *encoding-str* "UTF-8").

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, if a conversion is encountered, the exn:fail exception is raised.

If *close?* is true, then closing the result output port also closes *output-port*.

If *name-v* is provided, it is used as the name of the result output port, otherwise the port is named by (object-name *output-port*).

The *buffer-sym* argument determines the buffer mode of the output port, and it must be 'block, 'line, or 'none. If *output-port* is a file-stream port, the default is (file-stream-buffer-mode *output-port*), otherwise the default is '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.

(regexp-match-evt *pattern input-port*) PROCEDURE

Returns a synchronizable event (see §7.7 in *PLT MzScheme: Language Manual*) that is ready when *pattern* matches the stream of bytes/characters from *input-port* (see also §10 in *PLT MzScheme: Language Manual*). 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-string caret ("^") 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 *input-port* must support progress events. If *input-port* returns a special non-byte value during the match attempt, it is treated like eof.

(relocate-input-port *input-port line-k column-k position-k* [*close?*]) PROCEDURE

Produces an input port that is equivalent to *input-port* except in how it reports location information. The resulting port's content starts with the remaining content of *input-port*, and it starts at the given line, column, and position. The *line-k* argument must be a positive exact integer or #f, *column-k* must be a non-negative exact integer or #f, and *position-k* must be a positive exact integer (#f is not allowed for *position-k*). A #f for the line or column means that the line and column will always be reported as #f.

The *line-k* and *column-k* values are used only if line counting is enabled for *input-port* and for the resulting port, typically through port-count-lines! (see §11.2.1.1 in *PLT MzScheme: Language Manual*). The *column-k* value determines the column for the first line (i.e., the one numbered *line-k*), and later lines start at column 0. The given *position-k* is used even if line counting is not enabled.

When line counting is on for the resulting port, reading from *input-port* 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 *input-port*.

If *close?* is true (the default), then closing the resulting port also closes *input-port*. If *close?* is #f, then closing the resulting port does not close *input-port*.

(relocate-output-port *output-port line-k column-k position-k* [*close?*]) PROCEDURE

Like relocate-input-port, but for output ports.

(transplant-input-port *input-port position-thunk position-k* [*close? count-lines!-proc*]) PROCEDURE

Like *relocate-input-port*, except that arbitrary position information can be produced (when line counting is enabled) via *position-thunk*. If *position-thunk* is #f, then the port counts lines in the usual way, independent of locations reported by *input-port*.

If *count-lines!-proc* is supplied, it is called when line counting is enabled for the resulting port. The default is void.

(transplant-output-port *input-port position-thunk position-k* [*close? count-lines!-proc*]) PROCEDURE

Like transplant-input-port, but for output ports.

(strip-shell-command-start *input-port*) PROCEDURE

Reads and discards a leading #! in *input-port* (plus continuing lines if the line ends with a backslash) in the same way as the default load handler.

33. pregexp.ss: Perl-Style Regular Expressions

To load: `(require (lib "pregexp.ss"))`

This library provides regular expressions modeled on Perl's , and includes such powerful directives as numeric and nongreedy quantifiers, capturing and non-capturing clustering, POSIX character classes, selective case- and space-insensitivity, backreferences, alternation, backtrack pruning, positive and negative lookahead and lookbehind, in addition to the more basic directives familiar to all regexp users.

See also MzScheme's built-in `regexp` functions (§10 in *PLT MzScheme: Language Manual*), which are less expressive in some ways, but typically match more efficiently.

33.1 Introduction

A *regexp* is a string that describes a pattern. A regexp matcher tries to *match* this pattern against (a portion of) another string, which we will call the *text string*. The text string is treated as raw text and not as a pattern.

Most of the characters in a regexp pattern are meant to match occurrences of themselves in the text string. Thus, the pattern `"abc"` matches a string that contains the characters `a`, `b`, `c` in succession.

In the regexp pattern, some characters act as *metacharacters*, and some character sequences act as *metasequences*. That is, they specify something other than their literal selves. For example, in the pattern `"a.c"`, the characters `a` and `c` do stand for themselves but the *metacharacter* '`.`' can match *any* character (other than newline). Therefore, the pattern `"a.c"` matches an `a`, followed by *any* character, followed by a `c`.

If we needed to match the character '`.`' itself, we *escape* it, ie, precede it with a backslash (`\`). The character sequence `\.` is thus a *metasequence*, since it doesn't match itself but rather just '`.`'. So, to match `a` followed by a literal '`.`' followed by `c`, we use the regexp pattern `"a\\.c"`.[1] Another example of a metasequence is `\t`, which is a readable way to represent the tab character.

We will call the string representation of a regexp the *U-regexp*, where *U* can be taken to mean *Unix-style* or *universal*, because this notation for regexps is universally familiar. Our implementation uses an intermediate tree-like representation called the *S-regexp*, where *S* can stand for *Scheme*, *symbolic*, or *s-expression*. S-regexps are more verbose and less readable than U-regexps, but they are much easier for Scheme's recursive procedures to navigate.

33.2 Regexp procedures

This library provides the procedures `pregexp`, `pregexp-match-positions`, `pregexp-match`, `pregexp-split`, `pregexp-replace`, `pregexp-replace*`, and `pregexp-quote`.

[1]The double backslash is an artifact of Scheme strings, not the regexp pattern itself. When we want a literal backslash inside a Scheme string, we must escape it so that it shows up in the string at all. Scheme strings use backslash as the escape character, so we end up with two backslashes — one Scheme-string backslash to escape the regexp backslash, which then escapes the dot. Another character that would need escaping inside a Scheme string is '`"`'.

33.2.1 `pregexp`

`(pregexp` *`U-regexp`*`)` PROCEDURE

Takes a U-regexp, which is a string, and returns an S-regexp, which is a tree.

```
(pregexp "c.r")
=> (:sub (:or (:seq #\c :any #\r)))
```

There is rarely any need to look at the S-regexps returned by `pregexp`.

33.2.2 `pregexp-match-positions`

`(pregexp-match-positions` *`regexp text-string`* [*`start end`*]`)` PROCEDURE

Takes a regexp pattern and a text string, and returns a *match* if the regexp *matches* (some part of) the text string.

The regexp may be either a U- or an S-regexp. (`pregexp-match-positions` will internally compile a U-regexp to an S-regexp before proceeding with the matching. If you find yourself calling `pregexp-match-positions` repeatedly with the same U-regexp, it may be advisable to explicitly convert the latter into an S-regexp once beforehand, using `pregexp`, to save needless recompilation.)

`pregexp-match-positions` returns `#f` if the regexp did not match the string; and a list of *index pairs* if it did match. Eg,

```
(pregexp-match-positions "brain" "bird")
=> #f

(pregexp-match-positions "needle" "hay needle stack")
=> ((4 . 10))
```

In the second example, the integers 4 and 10 identify the substring that was matched. 4 is the starting (inclusive) index and 10 the ending (exclusive) index of the matching substring.

```
(substring "hay needle stack" 4 10)
=> "needle"
```

Here, `pregexp-match-positions`'s return list contains only one index pair, and that pair represents the entire substring matched by the regexp. When we discuss *subpatterns* later, we will see how a single match operation can yield a list of *submatches*.

`pregexp-match-positions` takes optional third and fourth arguments that specify the indices of the text string within which the matching should take place.

```
(pregexp-match-positions "needle"
  "his hay needle stack -- my hay needle stack -- her hay needle stack"
  24 43)
=> ((31 . 37))
```

Note that the returned indices are still reckoned relative to the full text string.

33.2.3 `pregexp-match`

(`pregexp-match` *regexp text-string* [*start end*]) PROCEDURE

Called like `pregexp-match-positions` but instead of returning index pairs it returns the matching substrings:

```
(pregexp-match "brain" "bird")
=> #f

(pregexp-match "needle" "hay needle stack")
=> ("needle")
```

`pregexp-match` also takes optional third and fourth arguments, with the same meaning as does `pregexp-match-positions`.

33.2.4 `pregexp-split`

(`pregexp-split` *regexp text-string*) PROCEDURE

Takes two arguments, a regexp pattern and a text string, and returns a list of substrings of the text string, where the pattern identifies the delimiter separating the substrings.

```
(pregexp-split ":" "/bin:/usr/bin:/usr/bin/X11:/usr/local/bin")
=> ("/bin" "/usr/bin" "/usr/bin/X11" "/usr/local/bin")

(pregexp-split " " "pea soup")
=> ("pea" "soup")
```

If the first argument can match an empty string, then the list of all the single-character substrings is returned.

```
(pregexp-split "" "smithereens")
=> ("s" "m" "i" "t" "h" "e" "r" "e" "e" "n" "s")
```

To identify one-or-more spaces as the delimiter, take care to use the regexp `" +"`, not `" *"`.

```
(pregexp-split " +" "split pea     soup")
=> ("split" "pea" "soup")

(pregexp-split " *" "split pea     soup")
=> ("s" "p" "l" "i" "t" "p" "e" "a" "s" "o" "u" "p")
```

33.2.5 `pregexp-replace`

(`pregexp-replace` *regexp text-string proc-or-insert-string*) PROCEDURE

Replaces the matched portion of the text string by another string. The first argument is the pattern, the second the text string, and the third is either the *insert string* (string to be inserted) or a procedure to convert matches to the insert string.

```
(pregexp-replace "te" "liberte" "ty")
=> "liberty"
(pregexp-replace "." "scheme" string-upcase)
=> "Scheme"
```

If the pattern doesn't occur in the text string, the returned string is identical (`eq?`) to the text string.

33.2.6 `pregexp-replace*`

(pregexp-replace* *regexp text-string proc-or-insert-string*) PROCEDURE

Replaces *all* matches in the text string by the insert string:

```
(pregexp-replace* "te" "liberte egalite fraternite" "ty")
=> "liberty egality fratyrnity"
(pregexp-replace* "[ds]" "drscheme" string-upcase)
=> "DrScheme"
```

As with `pregexp-replace`, if the pattern doesn't occur in the text string, the returned string is identical (`eq?`) to the text string.

33.2.7 `pregexp-quote`

(pregexp-quote *string*) PROCEDURE

Takes an arbitrary string and returns a U-regexp (string) that precisely represents it. In particular, characters in the input string that could serve as regexp metacharacters are escaped with a backslash, so that they safely match only themselves.

```
(pregexp-quote "cons")
=> "cons"

(pregexp-quote "list?")
=> "list\\?"
```

`pregexp-quote` is useful when building a composite regexp from a mix of regexp strings and verbatim strings.

33.3 The regexp pattern language

Here is a complete description of the regexp pattern language recognized by the `pregexp` procedures.

33.3.1 Basic assertions

The *assertions* `^` and `$` identify the beginning and the end of the text string respectively. They ensure that their adjoining regexps match at one or other end of the text string. Examples:

```
(pregexp-match-positions "^contact" "first contact")
=> #f
```

The regexp fails to match because `contact` does not occur at the beginning of the text string.

```
(pregexp-match-positions "laugh$" "laugh laugh laugh laugh")
=> ((18 . 23))
```

The regexp matches the *last* `laugh`.

The metasequence `\b` asserts that a *word boundary* exists.

```
(pregexp-match-positions "yack\\b" "yackety yack")
=> ((8 . 12))
```

The `yack` in `yackety` doesn't end at a word boundary so it isn't matched. The second `yack` does and is.

The metasequence `\B` has the opposite effect to `\b`. It asserts that a word boundary does not exist.

```
(pregexp-match-positions "an\\B" "an analysis")
=> ((3 . 5))
```

The `an` that doesn't end in a word boundary is matched.

33.3.2 Characters and character classes

Typically a character in the regexp matches the same character in the text string. Sometimes it is necessary or convenient to use a regexp metasequence to refer to a single character. Thus, metasequences `\n`, `\r`, `\t`, and `\.` match the newline, return, tab and period characters respectively.

The *metacharacter* period (`.`) matches *any* character other than newline.

```
(pregexp-match "p.t" "pet")
=> ("pet")
```

It also matches `pat`, `pit`, `pot`, `put`, and `p8t` but not `peat` or `pfffft`.

A *character class* matches any one character from a set of characters. A typical format for this is the *bracketed character class* `[`...`]`, which matches any one character from the non-empty sequence of characters enclosed within the brackets.[2] Thus `"p[aeiou]t"` matches `pat`, `pet`, `pit`, `pot`, `put` and nothing else.

Inside the brackets, a hyphen (`-`) between two characters specifies the ascii range between the characters. Eg, `"ta[b-dgn-p]"` matches `tab`, `tac`, `tad`, *and* `tag`, *and* `tan`, `tao`, `tap`.

An initial caret (`^`) after the left bracket inverts the set specified by the rest of the contents, ie, it specifies the set of characters *other than* those identified in the brackets. Eg, `"do[^g]"` matches all three-character sequences starting with `do` except `dog`.

Note that the metacharacter `^` inside brackets means something quite different from what it means outside. Most other metacharacters (`.`, `*`, `+`, `?`, etc) cease to be metacharacters when inside brackets, although you may still escape them for peace of mind. `-` is a metacharacter only when it's inside brackets, and neither the first nor the last character.

Bracketed character classes cannot contain other bracketed character classes (although they contain certain other types of character classes — see below). Thus a left bracket (`[`) inside a bracketed character class doesn't have to be a metacharacter; it can stand for itself. Eg, `"[a[b]"` matches `a`, `[`, and `b`.

Furthermore, since empty bracketed character classes are disallowed, a right bracket (`]`) immediately occurring after the opening left bracket also doesn't need to be a metacharacter. Eg, `"[]ab]"` matches `]`, `a`, and `b`.

33.3.2.1 SOME FREQUENTLY USED CHARACTER CLASSES

Some standard character classes can be conveniently represented as metasequences instead of as explicit bracketed expressions. `\d` matches a digit (`[0-9]`); `\s` matches a whitespace character; and `\w` matches a character that could

[2]Requiring a bracketed character class to be non-empty is not a limitation, since an empty character class can be more easily represented by an empty string.

be part of a "word".[3]

The upper-case versions of these metasequences stand for the inversions of the corresponding character classes. Thus `\D` matches a non-digit, `\S` a non-whitespace character, and `\W` a non-"word" character.

Remember to include a double backslash when putting these metasequences in a Scheme string:

```
(pregexp-match "\\d\\d"
  "0 dear, 1 have 2 read catch 22 before 9")
=> ("22")
```

These character classes can be used inside a bracketed expression. Eg, `"[a-z\\d]"` matches a lower-case letter or a digit.

33.3.2.2 POSIX CHARACTER CLASSES

A *POSIX character class* is a special metasequence of the form `[:...:]` that can be used only inside a bracketed expression. The POSIX classes supported are

`[:alnum:]`	letters and digits
`[:alpha:]`	letters
`[:algor:]`	the letters `c`, `h`, `a` and `d`
`[:ascii:]`	7-bit ascii characters
`[:blank:]`	widthful whitespace, ie, space and tab
`[:cntrl:]`	"control" characters, viz, those with code < 32
`[:digit:]`	digits, same as `\d`
`[:graph:]`	characters that use ink
`[:lower:]`	lower-case letters
`[:print:]`	ink-users plus widthful whitespace
`[:space:]`	whitespace, same as `\s`
`[:upper:]`	upper-case letters
`[:word:]`	letters, digits, and underscore, same as `\w`
`[:xdigit:]`	hex digits

For example, the regexp `"[[:alpha:]_]"` matches a letter or underscore.

```
(pregexp-match "[[:alpha:]_]" "--x--")
=> ("x")

(pregexp-match "[[:alpha:]_]" "--_--")
=> ("_")

(pregexp-match "[[:alpha:]_]" "--:--")
=> #f
```

The POSIX class notation is valid *only* inside a bracketed expression. For instance, `[:alpha:]`, when not inside a bracketed expression, will *not* be read as the letter class. Rather it is (from previous principles) the character class containing the characters `:`, `a`, `l`, `p`, `h`.

```
(pregexp-match "[:alpha:]" "--a--")
=> ("a")
```

[3]Following regexp custom, we identify "word" characters as `[A-Za-z0-9_]`, although these are too restrictive for what a Schemer might consider a "word".

```
(pregexp-match "[:alpha:]" "--_--")
=> #f
```

By placing a caret (`^`) immediately after `[:`, you get the inversion of that POSIX character class. Thus, `[:^alpha]` is the class containing all characters except the letters.

33.3.3 Quantifiers

The *quantifiers* `*`, `+`, and `?` match respectively: zero or more, one or more, and zero or one instances of the preceding subpattern.

```
(pregexp-match-positions "c[ad]*r" "cadaddadddr")
=> ((0 . 11))
(pregexp-match-positions "c[ad]*r" "cr")
=> ((0 . 2))

(pregexp-match-positions "c[ad]+r" "cadaddadddr")
=> ((0 . 11))
(pregexp-match-positions "c[ad]+r" "cr")
=> #f

(pregexp-match-positions "c[ad]?r" "cadaddadddr")
=> #f
(pregexp-match-positions "c[ad]?r" "cr")
=> ((0 . 2))
(pregexp-match-positions "c[ad]?r" "car")
=> ((0 . 3))
```

33.3.3.1 Numeric quantifiers

You can use braces to specify much finer-tuned quantification than is possible with `*`, `+`, `?`.

The quantifier `{m}` matches *exactly* `m` instances of the preceding *subpattern*. `m` must be a nonnegative integer.

The quantifier `{m,n}` matches at least `m` and at most `n` instances. `m` and `n` are nonnegative integers with `m <= n`. You may omit either or both numbers, in which case `m` defaults to 0 and `n` to infinity.

It is evident that `+` and `?` are abbreviations for `{1,}` and `{0,1}` respectively. `*` abbreviates `{,}`, which is the same as `{0,}`.

```
(pregexp-match "[aeiou]{3}" "vacuous")
=> ("uou")

(pregexp-match "[aeiou]{3}" "evolve")
=> #f

(pregexp-match "[aeiou]{2,3}" "evolve")
=> #f

(pregexp-match "[aeiou]{2,3}" "zeugma")
=> ("eu")
```

33.3.3.2 Non-greedy quantifiers

The quantifiers described above are *greedy*, ie, they match the maximal number of instances that would still lead to an overall match for the full pattern.

```
(pregexp-match "<.*>" "<tag1> <tag2> <tag3>")
=> ("<tag1> <tag2> <tag3>")
```

To make these quantifiers *non-greedy*, append a `?` to them. Non-greedy quantifiers match the minimal number of instances needed to ensure an overall match.

```
(pregexp-match "<.*?>" "<tag1> <tag2> <tag3>")
=> ("<tag1>")
```

The non-greedy quantifiers are respectively: `*?`, `+?`, `??`, `{m}?`, `{m,n}?`. Note the two uses of the metacharacter `?`.

33.3.4 Clusters

Clustering, ie, enclosure within parens `(...)`, identifies the enclosed *subpattern* as a single entity. It causes the matcher to *capture* the *submatch*, or the portion of the string matching the subpattern, in addition to the overall match.

```
(pregexp-match "([a-z]+) ([0-9]+), ([0-9]+)" "jan 1, 1970")
=> ("jan 1, 1970" "jan" "1" "1970")
```

Clustering also causes a following quantifier to treat the entire enclosed subpattern as an entity.

```
(pregexp-match "(poo )*" "poo poo platter")
=> ("poo poo " "poo ")
```

The number of submatches returned is always equal to the number of subpatterns specified in the regexp, even if a particular subpattern happens to match more than one substring or no substring at all.

```
(pregexp-match "([a-z ]+;)*" "lather; rinse; repeat;")
=> ("lather; rinse; repeat;" " repeat;")
```

Here the `*`-quantified subpattern matches three times, but it is the last submatch that is returned.

It is also possible for a quantified subpattern to fail to match, even if the overall pattern matches. In such cases, the failing submatch is represented by `#f`.

```
(define date-re
  ;match ‘month year’ or ‘month day, year’.
  ;subpattern matches day, if present
  (pregexp "([a-z]+) +([0-9]+,)? *([0-9]+)"))

(pregexp-match date-re "jan 1, 1970")
=> ("jan 1, 1970" "jan" "1," "1970")

(pregexp-match date-re "jan 1970")
=> ("jan 1970" "jan" #f "1970")
```

33.3.4.1 Backreferences

Submatches can be used in the insert string argument of the procedures `pregexp-replace` and `pregexp-replace*`. The insert string can use `\n` as a *backreference* to refer back to the *n*th submatch, ie, the substring that matched the *n*th subpattern. `\0` refers to the entire match, and it can also be specified as `\&`.

```
(pregexp-replace "_(.+?)_"
  "the _nina_, the _pinta_, and the _santa maria_"
  "*\\1*")
=> "the *nina*, the _pinta_, and the _santa maria_"

(pregexp-replace* "_(.+?)_"
  "the _nina_, the _pinta_, and the _santa maria_"
  "*\\1*")
=> "the *nina*, the *pinta*, and the *santa maria*"

;recall: \S stands for non-whitespace character

(pregexp-replace "(\\S+) (\\S+) (\\S+)"
  "eat to live"
  "\\3 \\2 \\1")
=> "live to eat"
```

Use `\\` in the insert string to specify a literal backslash. Also, `\$` stands for an empty string, and is useful for separating a backreference `\n` from an immediately following number.

Backreferences can also be used within the regexp pattern to refer back to an already matched subpattern in the pattern. `\n` stands for an exact repeat of the *n*th submatch.[4]

```
(pregexp-match "([a-z]+) and \\1"
  "billions and billions")
=> ("billions and billions" "billions")
```

Note that the backreference is not simply a repeat of the previous subpattern. Rather it is a repeat of *the particular substring already matched by the subpattern*.

In the above example, the backreference can only match `billions`. It will not match `millions`, even though the subpattern it harks back to — `([a-z]+)` — would have had no problem doing so:

```
(pregexp-match "([a-z]+) and \\1"
  "billions and millions")
=> #f
```

The following corrects doubled words:

```
(pregexp-replace* "(\\S+) \\1"
  "now is the the time for all good men to to come to the aid of of the party"
  "\\1")
=> "now is the time for all good men to come to the aid of the party"
```

The following marks all immediately repeating patterns in a number string:

```
(pregexp-replace* "(\\d+)\\1"
  "123340983242432420980980234"
  "{\\1,\\1}")
=> "12{3,3}40983{24,24}3242{098,098}0234"
```

[4] `0`, which is useful in an insert string, makes no sense within the regexp pattern, because the entire regexp has not matched yet that you could refer back to it.

33.3.4.2 Non-capturing clusters

It is often required to specify a cluster (typically for quantification) but without triggering the capture of submatch information. Such clusters are called *non-capturing*. In such cases, use `(?:` instead of `(` as the cluster opener. In the following example, the non-capturing cluster eliminates the "directory" portion of a given pathname, and the capturing cluster identifies the basename.

```
(pregexp-match "^(?:[a-z]*/)*([a-z]+)$"
  "/usr/local/bin/mzscheme")
=> ("/usr/local/bin/mzscheme" "mzscheme")
```

33.3.4.3 Cloisters

The location between the `?` and the `:` of a non-capturing cluster is called a *cloister*.[5] You can put *modifiers* there that will cause the enclustered subpattern to be treated specially. The modifier `i` causes the subpattern to match *case-insensitively*:

```
(pregexp-match "(?i:hearth)" "HeartH")
=> ("HeartH")
```

The modifier `x` causes the subpattern to match *space-insensitively*, ie, spaces and comments within the subpattern are ignored. Comments are introduced as usual with a semicolon (`;`) and extend till the end of the line. If you need to include a literal space or semicolon in a space-insensitized subpattern, escape it with a backslash.

```
(pregexp-match "(?x: a   lot)" "alot")
=> ("alot")

(pregexp-match "(?x: a  \\  lot)" "a lot")
=> ("a lot")

(pregexp-match "(?x:
   a \\ man  \\; \\   ; ignore
   a \\ plan \\; \\   ; me
   a \\ canal         ; completely
   )"
 "a man; a plan; a canal")
=> ("a man; a plan; a canal")
```

The global variable `*pregexp-comment-char*` contains the comment character (`#\;`). For Perl-like comments,

```
(set! *pregexp-comment-char* #\#)
```

You can put more than one modifier in the cloister.

```
(pregexp-match "(?ix:
   a \\ man  \\; \\   ; ignore
   a \\ plan \\; \\   ; me
   a \\ canal         ; completely
   )"
 "A Man; a Plan; a Canal")
=> ("A Man; a Plan; a Canal")
```

A minus sign before a modifier inverts its meaning. Thus, you can use `-i` and `-x` in a *subcluster* to overturn the insensitivities caused by an enclosing cluster.

[5] A useful, if terminally cute, coinage from the abbots of Perl .

```
(pregexp-match "(?i:the (?-i:TeX)book)"
  "The TeXbook")
=> ("The TeXbook")
```

This regexp will allow any casing for `the` and `book` but insists that `TeX` not be differently cased.

33.3.5 Alternation

You can specify a list of *alternate* subpatterns by separating them by `|`. The `|` separates subpatterns in the nearest enclosing cluster (or in the entire pattern string if there are no enclosing parens).

```
(pregexp-match "f(ee|i|o|um)" "a small, final fee")
=> ("fi" "i")

(pregexp-replace* "([yi])s(e[sdr]?|ing|ation)"
   "it is energising to analyse an organisation
   pulsing with noisy organisms"
   "\\1z\\2")
=> "it is energizing to analyze an organization
   pulsing with noisy organisms"
```

Note again that if you wish to use clustering merely to specify a list of alternate subpatterns but do not want the submatch, use `(?:` instead of `(`.

```
(pregexp-match "f(?:ee|i|o|um)" "fun for all")
=> ("fo")
```

An important thing to note about alternation is that the leftmost matching alternate is picked regardless of its length. Thus, if one of the alternates is a prefix of a later alternate, the latter may not have a chance to match.

```
(pregexp-match "call|call-with-current-continuation"
  "call-with-current-continuation")
=> ("call")
```

To allow the longer alternate to have a shot at matching, place it before the shorter one:

```
(pregexp-match "call-with-current-continuation|call"
  "call-with-current-continuation")
=> ("call-with-current-continuation")
```

In any case, an overall match for the entire regexp is always preferred to an overall nonmatch. In the following, the longer alternate still wins, because its preferred shorter prefix fails to yield an overall match.

```
(pregexp-match "(?:call|call-with-current-continuation) constrained"
  "call-with-current-continuation constrained")
=> ("call-with-current-continuation constrained")
```

33.3.6 Backtracking

We've already seen that greedy quantifiers match the maximal number of times, but the overriding priority is that the overall match succeed. Consider

```
(pregexp-match "a*a" "aaaa")
```

The regexp consists of two subregexps, `a*` followed by `a`. The subregexp `a*` cannot be allowed to match all four `a`'s in the text string `"aaaa"`, even though `*` is a greedy quantifier. It may match only the first three, leaving the last one for the second subregexp. This ensures that the full regexp matches successfully.

The regexp matcher accomplishes this via a process called *backtracking*. The matcher tentatively allows the greedy quantifier to match all four `a`'s, but then when it becomes clear that the overall match is in jeopardy, it *backtracks* to a less greedy match of *three* `a`'s. If even this fails, as in the call

```
(pregexp-match "a*aa" "aaaa")
```

the matcher backtracks even further. Overall failure is conceded only when all possible backtracking has been tried with no success.

Backtracking is not restricted to greedy quantifiers. Nongreedy quantifiers match as few instances as possible, and progressively backtrack to more and more instances in order to attain an overall match. There is backtracking in alternation too, as the more rightward alternates are tried when locally successful leftward ones fail to yield an overall match.

33.3.6.1 Disabling backtracking

Sometimes it is efficient to disable backtracking. For example, we may wish to *commit* to a choice, or we know that trying alternatives is fruitless. A nonbacktracking regexp is enclosed in `(?>...)`.

```
(pregexp-match "(?>a+)." "aaaa")
=> #f
```

In this call, the subregexp `?>a*` greedily matches all four `a`'s, and is denied the opportunity to backpedal. So the overall match is denied. The effect of the regexp is therefore to match one or more `a`'s followed by something that is definitely non-`a`.

33.3.7 Looking ahead and behind

You can have assertions in your pattern that look *ahead* or *behind* to ensure that a subpattern does or does not occur. These "look around" assertions are specified by putting the subpattern checked for in a cluster whose leading characters are: `?=` (for positive lookahead), `?!` (negative lookahead), `?<=` (positive lookbehind), `?<!` (negative lookbehind). Note that the subpattern in the assertion does not generate a match in the final result. It merely allows or disallows the rest of the match.

33.3.7.1 Lookahead

Positive lookahead (`?=`) peeks ahead to ensure that its subpattern *could* match.

```
(pregexp-match-positions "grey(?=hound)"
  "i left my grey socks at the greyhound")
=> ((28 . 32))
```

The regexp `"grey(?=hound)"` matches `grey`, but *only* if it is followed by `hound`. Thus, the first `grey` in the text string is not matched.

Negative lookahead (`?!`) peeks ahead to ensure that its subpattern could not possibly match.

```
(pregexp-match-positions "grey(?!hound)"
  "the gray greyhound ate the grey socks")
=> ((27 . 31))
```

The regexp `"grey(?!hound)"` matches `grey`, but only if it is *not* followed by `hound`. Thus the `grey` just before `socks` is matched.

33.3.7.2 LOOKBEHIND

Positive lookbehind (`?<=`) checks that its subpattern *could* match immediately to the left of the current position in the text string.

```
(pregexp-match-positions "(?<=grey)hound"
  "the hound in the picture is not a greyhound")
=> ((38 . 43))
```

The regexp `(?<=grey)hound` matches `hound`, but only if it is preceded by `grey`.

Negative lookbehind (`?<!`) checks that its subpattern could not possibly match immediately to the left.

```
(pregexp-match-positions "(?<!grey)hound"
  "the greyhound in the picture is not a hound")
=> ((38 . 43))
```

The regexp `(?<!grey)hound` matches `hound`, but only if it is *not* preceded by `grey`.

Lookaheads and lookbehinds can be convenient when they are not confusing.

33.4 An extended example

Here's an extended example from Friedl's *Mastering Regular Expressions* that covers many of the features described above. The problem is to fashion a regexp that will match any and only IP addresses or *dotted quads*, ie, four numbers separated by three dots, with each number between 0 and 255. We will use the commenting mechanism to build the final regexp with clarity. First, a subregexp `n0-255` that matches 0 through 255.

```
(define n0-255
  "(?x:
  \\d          ;  0 through   9
   \\d\\d     ; 00 through  99
   [01]\\d\\d ;000 through 199
   2[0-4]\\d  ;200 through 249
   25[0-5]    ;250 through 255
  )")
```

The first two alternates simply get all single- and double-digit numbers. Since 0-padding is allowed, we need to match both 1 and 01. We need to be careful when getting 3-digit numbers, since numbers above 255 must be excluded. So we fashion alternates to get 000 through 199, then 200 through 249, and finally 250 through 255.[6]

An IP-address is a string that consists of four `n0-255`s with three dots separating them.

```
(define ip-re1
  (string-append
    "^"        ;nothing before
    n0-255     ;the first n0-255,
    "(?x:"     ;then the subpattern of
```

[6]Note that *`n0-255`* lists prefixes as preferred alternates, something we cautioned against in sec 33.3.5. However, since we intend to anchor this subregexp explicitly to force an overall match, the order of the alternates does not matter.

```
    "\\."       ;a dot followed by
    n0-255      ;an n0-255,
    ")"         ;which is
    "{3}"       ;repeated exactly 3 times
    "$"         ;with nothing following
    ))
```

Let's try it out.

```
(pregexp-match ip-re1
  "1.2.3.4")
=> ("1.2.3.4")

(pregexp-match ip-re1
  "55.155.255.265")
=> #f
```

which is fine, except that we also have

```
(pregexp-match ip-re1
  "0.00.000.00")
=> ("0.00.000.00")
```

All-zero sequences are not valid IP addresses! Lookahead to the rescue. Before starting to match `ip-re1`, we look ahead to ensure we don't have all zeros. We could use positive lookahead to ensure there *is* a digit other than zero.

```
(define ip-re
  (string-append
    "(?=.*[1-9])" ;ensure there’s a non-0 digit
    ip-re1))
```

Or we could use negative lookahead to ensure that what's ahead isn't composed of *only* zeros and dots.

```
(define ip-re
  (string-append
    "(?![0.]*$)" ;not just zeros and dots
                 ;(note: dot is not metachar inside [])
    ip-re1))
```

The regexp `ip-re` will match all and only valid IP addresses.

```
(pregexp-match ip-re
  "1.2.3.4")
=> ("1.2.3.4")

(pregexp-match ip-re
  "0.0.0.0")
=> #f
```

34. pretty.ss: Pretty Printing

To load: `(require (lib "pretty.ss"))`

`(pretty-display v [port])` PROCEDURE

Same as `pretty-print`, but *v* is printed like `display` instead of like `write`.

`(pretty-print v [port])` PROCEDURE

Pretty-prints the value *v* using the same printed form as `write`, 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`.

If *port* is provided, *v* is printed into *port*; otherwise, *v* is printed to the current output port.

In addition to the parameters defined by the **pretty** library, `pretty-print` conforms to the `print-graph`, `print-struct`, and `print-vector-length` parameters.

The pretty printer also detects structures that have the `prop:custom-write` property (see §11.2.10 in *PLT MzScheme: Language Manual*) 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 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-print-current-style-table style-table [procedure])`

Parameter that holds a table of style mappings. See `pretty-print-extend-style-table`.

`(pretty-print-columns [width])` PROCEDURE

Parameter that sets the default width for pretty printing to *width* and returns void. If no *width* argument is provided, the current value is returned instead.

If the display width is `'infinity`, then pretty-printed output is never broken into lines, and a newline is not added to the end of the output.

`(pretty-print-depth [depth])` PROCEDURE

Parameter that 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 [`*`as-decimal?`*`])` PROCEDURE

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-extend-style-table` *`style-table symbol-list like-symbol-list`*`)` PROCEDURE

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-handler` *`v`*`)` PROCEDURE

Pretty-prints *`v`* if *`v`* is not void or prints nothing otherwise. Pass this procedure to `current-print` to install the pretty printer into the `read-eval-print` loop.

`(pretty-print-newline` *`port width-k`*`)` PROCEDURE

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-k`* argument should be the target column width, typically obtained from `pretty-print-columns`.

(pretty-print-print-hook [*proc*]) PROCEDURE

Parameter that sets the print hook for pretty-printing to *proc*. If *proc* is not provided, the current hook is returned.

The print hook is applied to a value for printing when the sizing hook (see `pretty-print-size-hook`) returns an integer size for the value.

The print hook receives three arguments. The first argument is the value to print. The second argument is a Boolean: `#t` for printing like `display` and `#f` for printing like `write`. The third argument is the destination port; 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-print-line [*proc*]) PROCEDURE

Parameter that sets a procedure for printing the newline separator between lines of a pretty-printed value. The *proc* 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`.

(pretty-print-show-inexactness [*explicit?*]) PROCEDURE

Parameter that determines how inexact numbers are printed. If the parameter's value is `#t`, then inexact numbers are always printed with a leading `#i`. The initial value is `#f`.

(pretty-print-style-table? *v*) PROCEDURE

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

(pretty-print-post-print-hook [*proc*]) PROCEDURE

Parameter that sets 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).

(pretty-print-pre-print-hook [*proc*]) PROCEDURE

Parameter that sets 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-size-hook [hook])` PROCEDURE

Parameter that sets the sizing hook for pretty-printing to *`hook`*. If *`hook`* is not provided, the current hook is returned.

The sizing hook is applied to each value to be printed. If the hook returns `#f`, then printing is handled internally by the pretty-printer. Otherwise, the value should be an integer specifying the length of the printed value in characters; the print hook will be called to actually print the value (see `pretty-print-print-hook`).

The sizing hook receives three arguments. The first argument is the value to print. The second argument is a Boolean: `#t` for printing like `display` and `#f` for printing like `write`. The third argument is the destination port; the 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-.-symbol-without-bars [bool])` PROCEDURE

Parameter that controls the printing of the symbol whose print name is just a period. If set to a true value, it is printed as only the period. If set to a false value, it is printed as a period with vertical bars surrounding it.

`(pretty-printing [on?])` PROCEDURE

Parameter that is set to `#t` when the pretty printer calls a custom-write procedure (see §11.2.10 in *PLT MzScheme: Language Manual*) 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 output-port width-k overflow-thunk)` PROCEDURE

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 *`output-port`* 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-k`* 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-k`* 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-output-port output-port)` PROCEDURE

Causes the data written to *`tentative-output-port`* to be transferred as if written to *`output-port`*. The *`tentative-output-port`* argument should be a port produced by `make-tentative-pretty-print-output-port`, and *`output-port`* should be either a pretty-printing port (provided to a custom-write procedure) or another tentative output port.

`(tentative-pretty-print-port-cancel` *`tentative-output-port`*`)` PROCEDURE

Cancels the content of *`tentative-output-port`*, which was produced by `make-tentative-pretty-print-output-por`
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`*`=`.

35. process.ss: Process and Shell-Command Execution

To load: `(require (lib "process.ss"))`

This library builds on MzScheme's `subprocess` procedure; see also §15.2 in *PLT MzScheme: Language Manual*.

`(system 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-string`* argument is a string containing no null characters. If the command succeeds, the return value is `#t`, `#f` otherwise.

`(system* command-string arg-string ···)` is like `system`, except that *`command-string`* is a filename that is executed directly (instead of through a shell command), and the *`arg-string`*s are the arguments. The executed file is passed the specified string arguments (which must contain no null characters). Under Windows, the first *`arg-string`* can be `'exact` where the second *`arg-string`* is a complete command line; see §15.2 in *PLT MzScheme: Language Manual* for details.

`(system/exit-code command-string)` is like `system`, except that it returns the exit-code returned by the subprocess instead of a boolean (a result of 0 indicates success).

`(system*/exit-code command-string)` is like `system*`, except that it returns the exit-code like `system/exit-code` does.

`(process command-string)` executes a shell command asynchronously. If the subprocess is launched successfully, the result is a list of five values:

- an input port piped from the subprocess's standard output,
- an output port piped to the subprocess standard input,
- the system process id of the subprocess,
- an input port piped from the subprocess's standard error,[1] 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` and `close-output-port`.

`(process* command-string arg-string ···)` is like `process`, except that *`command-string`* is a filename that is executed directly, and the *`arg-string`*s are the arguments. Under Windows, as for `system*`, the first *`arg-string`* can be `'exact`.

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

`(process/ports` *`output-port input-port error-output-port command-string`*`)` is like `process`, except that *`output-port`* is used for the process's standard output, *`input-port`* is used for the process's standard input, and *`error-output-port`* is used for the process's standard error. The provided ports need not be file-stream ports. 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 returned value is `#f`.

`(process*/ports` *`output-port input-port error-output-port command-string arg-string`* · · ·`)` is like `process*`, but with the port handling of `process/ports`.

36. restart.ss: Simulating Stand-alone MzScheme

To load: `(require (lib "restart.ss"))`

`(restart-mzscheme` *`init-argv adjust-flag-table argv init-namespace`*`)` PROCEDURE

Simulates starting the stand-alone version of MzScheme with the vector of command-line strings *`argv`*. The *`init-argv`*, *`adjust-flag-table`*, and *`init-namespace`* arguments are used to modify the default settings for command-line flags, adjust the parsing of command-line flags, and customize the initial namespace, respectively.

The vector of strings *`init-argv`* is read first with the standard MzScheme command-line parsing. Flags that load files or evaluate expressions (e.g., `-f` and `-e`) are ignored, but flags that set MzScheme's modes (e.g., `-g` or `-m`) effectively set the default mode before *`argv`* is parsed.

Before *`argv`* is parsed, the procedure *`adjust-flag-table`* is called with a command-line flag table as accepted by `parse-command-line` (see §8). The return value must also be a table of command-line flags, and this table is used to parse *`argv`*. The intent is to allow `adjust-flag-table` to add or remove flags from the standard set.

After *`argv`* is parsed, a new thread and a namespace are created for the "restarted" MzScheme. (The new namespace is installed as the current namespace in the new thread.) In the new thread, restarting performs the following actions:

- The *`init-namespace`* procedure is called with no arguments. The return value is ignored.
- Expressions and files specified by *`argv`* are evaluated and loaded. If an error occurs, the remaining expressions and files are ignored, and the return value for `restart-mzscheme` is set to `#f`.
- The `read-eval-print-loop` procedure is called, unless a flag in *`init-argv`* or *`argv`* disables it. When `read-eval-print-loop` returns, the return value for `restart-mzscheme` is set to `#t`.

Before evaluating command-line arguments, an exit handler is installed that immediately returns from `restart-mzscheme` with the value supplied to the handler. This exit handler remains in effect when `read-eval-print-loop` is called (unless a command-line argument changes it). If `restart-mzscheme` returns normally, the return value is determined as described above. (Note that an error in a command-line expression followed by `read-eval-print-loop` produces a `#t` result. This is consistent with MzScheme's stand-alone behavior.)

37. sendevent.ss: AppleEvents

To load: `(require (lib "sendevent.ss"))`

37.1 AppleEvents

(send-event *receiver-byte-string event-class-byte-string event-id-byte-string* [*direct-argum*
argument-list]) PROCEDURE

Sends an AppleEvent or raises `exn:fail:unsupported`. Currently AppleEvents are supported only within MrEd under Mac OS X.

The *receiver-byte-string*, *event-class-byte-string*, and *event-id-byte-string* arguments specify the signature of the receiving application, the class of the AppleEvent, and the ID of the AppleEvent. Each of these must be a four-character byte string, otherwise the `exn:fail:contract` exception is raised.

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

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

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

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

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

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

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

If the AppleEvent reply contains a value that cannot be reverse-converted, the `exn:fail` exception is raised.

38. serialize.ss: Serializing Data

To load: `(require (lib "serialize.ss"))`

`(define-serializable-struct` *`id`* `(`*`field-id`* `···)` *`[inspector-expr]`*`)` SYNTAX

`(define-serializable-struct` *`(id super-id)`* `(`*`field-id`* `···)` *`[inspector-expr]`*`)` SYNTAX

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).

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 `provide`s this binding.

Naturally, `define-serializable-struct` 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/ver`

When *`super-id`* is supplied, compile-time information bound to *`super-id`* 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).

Example:

```
(define-serializable-struct point (x y))
(deserialize (serialize (make-point 1 2))) ; ⇒ (make-point 1 2)
```

`(define-serializable-struct/version` *`id vers-num`* `(`*`field-id`* `···)` `((`*`other-vers-num make-proc-expr cycle-make-proc-expr`*`))` *`[inspector-expr]`*`)` SYNTAX

`(define-serializable-struct/version` *`(id super-id) vers-num`* `(`*`field-id`* `···)` `((`*`other-vers-num make-proc-expr cycle-make-proc-expr`*`))` *`[inspector-expr]`*`)` SYNTAX

Like `define-serializable-struct`, but the generated deserializer binding is `deserialize-info:`*`id-vers-num`*. In addition, `deserialize-info:`*`id-other-vers-num`* is bound for each *`other-vers-num`*.

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`*.

Example:

```
(define-serializable-struct point (x y))
(define ps (serialize (make-point 1 2)))
(deserialize ps) ; ⇒ (make-point 1 2)

(define x (make-point 1 10))
(set-point-x! x x)
(define xs (serialize x))
(deserialize xs) ; ⇒ x0, where x0 is (make-point x0 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))))))]))
(deserialize (serialize (make-point 4 5 6))) ; ⇒ (make-point 4 5 6)
(deserialize ps) ; ⇒ (make-point 1 2 0)
(deserialize xs) ; ⇒ x1, where x1 is (make-point x1 10 0)
```

`(serialize v)` PROCEDURE

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);
- instances of classes defined with `define-serializable-class` or `define-serializable-class` (see §4.7);
- booleans, numbers, characters, symbols, strings, byte strings, paths, void, and the empty list;
- pairs, vectors, boxes, and hash tables; and
- `date` and `arity-at-least` structures.

Of course, 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)` PROCEDURE

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 elements:

- 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 types. 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.
- 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;
    - `'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 flags for `make-hash-table` for a hash table.
    - `'date` for a `date` structure;
    - `'arity-at-least` for an `arity-at-least` structure;

    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 `'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 `'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 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 to be used as flags for `make-hash-table`, 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 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 *`graphs`* 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.

`(make-deserialize-info` *`make-proc cycle-make-proc`*`)` 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-proc`* 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-proc`* 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-proc`*, and it transfers the field values of this instance into *`x`*.

`prop:serializeable` 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-proc deserialize-id can-cycle? dir-path`*`)` PROCEDURE

Produces a value to be associated with a structure type through the `prop:serializable` property. This value is used by `serialize`.

The *`to-vector-proc`* 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. The *`deserialize-id`* value can be an identifier syntax object, a symbol, or a pair:

- 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 either a symbol or 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-path`* 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))`.

`(serializable? `*`v`*`)` PROCEDURE

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.

39. shared.ss: Graph Constructor Syntax

To load: `(require (lib "shared.ss"))`

`(shared (`*`shared-binding`* `···) `*`body-expr`* `···`[1]`)` SYNTAX

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

The `shared` form is similar to `letrec`. Each *`shared-binding`* has the form:

`(`*`variable value-expr`*`)`

The *`variable`*s are bound to the result of *`value-expr`*s in the same way as for a `letrec` expression, except for *`value-expr`*s with the following special forms (after partial expansion):

- `(cons` *`car-expr cdr-expr`*`)`
- `(list` *`element-expr`* `···)`
- `(box` *`box-expr`*`)`
- `(vector` *`element-expr`* `···)`
- `(`*`prefix:`*`make-`*`name element-expr`* `···)` where *`prefix:name`* is the name of a structure type (or, more generally, is bound to expansion-time information about a structure type)

The `cons` above means an identifier that is `module-identifier=?` either to the `cons` export from `mzscheme` or to the top-level `cons`. The same is true of `list`, `box`, and `vector`. In the `\var{prefix:}make-\var{name}` case, the expansion-time information associated with *`prefix:name`* must provide a constructor binding and a complete set of field mutator bindings.

For each of the special forms, the cons cell, list, box, vector, or structure is allocated with undefined content. The content expressions are not evaluated until all of the bindings have values; then the content expressions are evaluated and the values are inserted into the appropriate locations. In this way, values with shared structure (even cycles) can be constructed.

Examples:

```
(shared ([a (cons 1 a)]) a) ; => infinite list of 1s
(shared ([a (cons 1 b)]
         [b (cons 2 a)])
  a) ; => (1 2 1 2 1 2 ···)
(shared ([a (vector b b b)]
         [b (box 1)])
  (set-box! (vector-ref a 0) 2)
  a) ; => #(#&2 #&2 #&2)
```

40. string.ss: String Utilities

To load: `(require (lib "string.ss"))`

`(eval-string` *`str`* [*`err-display err-result`*]`)` PROCEDURE

Reads and evaluates S-expressions from the string *`str`*, returning a result for each expression. Note that if *`str`* contains only whitespace and comments, zero values are returned, while if *`str`* contains two expressions, two values are returned.

If *`err-display`* is not `#f` (the default), then errors are caught and *`err-display`* is used as the error display handler. If *`err-result`* is specified, it must be a thunk that returns a value to be returned when an error is caught; otherwise, `#f` is returned when an error is caught.

`(expr->string` *`expr`*`)` PROCEDURE

Prints *`expr`* into a string and returns the string.

`(read-from-string` *`str`* [*`err-display err-result`*]`)` PROCEDURE

Reads the first S-expression from the string *`str`* and returns it. The *`err-display`* and *`err-result`* are as in `eval-str`.

`(read-from-string-all` *`str`* [*`err-display err-result`*]`)` PROCEDURE

Reads all S-expressions from the string *`str`* and returns them in a list. The *`err-display`* and *`err-result`* are as in `eval-str`.

`(regexp-match*` *`pattern string-or-input-port`* [*`start-k end-k`*]`)` PROCEDURE

Like `regexp-match` (see §10 in *PLT MzScheme: Language Manual*), but the result is a list of strings corresponding to a sequence of matches of *`pattern`* in *`string-or-input-port`*. (Unlike `regexp-match`, results for parenthesized sub-patterns in *`pattern`* are not returned.) If *`pattern`* matches a zero-length string along the way, the `exn:fail` exception is raised.

If *`string-or-input-port`* contains no matches (in the range *`start-k`* to *`end-k`*), `null` is returned. Otherwise, each string in the resulting list is a distinct substring in *`string-or-input-port`* that matches *`pattern`*. The *`end-k`* argument can be `#f` to match to the end of *`string-or-input-port`*.

`(regexp-match/fail-without-reading` *`pattern input-port`* [*`start-k end-k output-port`*]`)`
PROCEDURE

Like `regexp-match` on input ports (see §10 in *PLT MzScheme: Language Manual*), except that if the match fails, no characters are read and discarded from *`input-port`*.

This procedure is especially useful with a *`pattern`* that begins with a start-of-string caret ("^") or with a non-`#f` *`end-k`*, since each limits the amount of peeking into the port.

`(regexp-match-exact?` *`pattern string-or-input-port`*`)` PROCEDURE

This procedure is like MzScheme's built-in `regexp-match` (see §10 in *PLT MzScheme: Language Manual*), but the result is always `#t` or `#f`; `#t` is only returned when the entire content of *`string-or-input-port`* matches *`pattern`*.

`(regexp-match-peek-positions*` *`pattern input-port`* [*`start-k end-k`*]`)` PROCEDURE

Like `regexp-match-positions*`, but it works only on input ports, and the port is peeked instead of read for matches.

`(regexp-match-positions*` *`pattern string-or-input-port`* [*`start-k end-k`*]`)` PROCEDURE

Like `regexp-match-positions` (see §10 in *PLT MzScheme: Language Manual*), but the result is a list of integer pairs corresponding to a sequence of matches of *`pattern`* in *`string-or-input-port`*. (Unlike `regexp-match-positions`, results for parenthesized sub-patterns in *`pattern`* are not returned.) If *`pattern`* matches a zero-length string along the way, the `exn:fail` exception is raised.

If *`string-or-input-port`* contains no matches (in the range *`start-k`* to *`end-k`*), `null` is returned. Otherwise, each position pair in the resulting list corresponds to a distinct substring in *`string-or-input-port`* that matches *`pattern`*. The *`end-k`* argument can be `#f` to match to the end of *`string-or-input-port`*.

`(regexp-quote` *`str`* [*`case-sensitive?`*]`)` PROCEDURE

Produces a string suitable for use with `regexp` (see §10 in *PLT MzScheme: Language Manual*) to match the literal sequence of characters in *`str`*. If *`case-sensitive?`* is true, the resulting regexp matches letters in *`str`* case-insensitively, otherwise (and by default) it matches case-sensitively.

`(regexp-replace-quote` *`str`*`)` PROCEDURE

Produces a string suitable for use as the third argument to `regexp-replace` (see §10 in *PLT MzScheme: Language Manual*) to insert the literal sequence of characters in *`str`* as a replacement. Concretely, every backslash and ampersand in *`str`* is protected by a quoting backslash.

`(glob->regexp` *`str`* [*`hide-dots? case-sensitive?`*]`)` PROCEDURE

Produces a regexp for a an input "glob pattern" in *`str`*. A glob pattern is one that matches "`*`" with any string, and "`?`" with a single character, and character ranges are the same as in regexps. In addition, the resulting regexp does not match strings that begin with a period, unless explicitly part of the glob string. The resulting regexp can be used with string file names to check the glob pattern.

If *`hide-dots?`* is true (the default), the resulting regexp will not match names that begin with a dot.

If *`case-sensitive?`* is given, it determines whether the resulting regexp is case-sensitive; otherwise the default case sensitivity depends on the system-type.

`(regexp-split` *`pattern string-or-input-port`* [*`start-k end-k`*]`)` PROCEDURE

The complement of `regexp-match*` (see above): the result is a list of sub-strings in *`string-or-input-port`* that are separated by matches to *`pattern`*; adjacent matches are separated with `""`. If *`pattern`* matches a zero-length string along the way, the `exn:fail` exception is raised.

If *`string-or-input-port`* contains no matches (in the range *`start-k`* to *`end-k`*), the result will be a list containing *`string-or-input-port`* (from *`start-k`* to *`end-k`*). If a match occurs at the beginning of *`string-or-input-port`* (at *`start-k`*), the resulting list will start with an empty string, and if a match occurs at the end (at *`end-k`*), the list will end with an empty string. The *`end-k`* argument can be `#f`, in which case splitting goes to the end of *`string-or-input-port`*.

`(string-lowercase!!` *`str`*`)` PROCEDURE

Destructively changes *`str`* to contain only lowercase characters.

`(string-uppercase!!` *`str`*`)` PROCEDURE

Destructively changes *`str`* to contain only uppercase characters.

41. struct.ss: Structure Utilities

To load: `(require (lib "struct.ss"))`

`(copy-struct` *`struct-id struct-expr (accessor-id field-expr)`* `···)` SYNTAX

This form provides "functional update" for structure instances. The result of evaluating *`struct-expr`* must be an instance of the structure type named by *`struct-id`*. The result of the `copy-struct` expression is a fresh instance of *`struct-id`* with the same field values as the result of *`struct-expr`*, except that the value for the field accessed by each *`accessor-id`* is replaced by the result of *`field-expr`*.

The result of *`struct-expr`* might be an instance of a sub-type of *`struct-id`*, but the result of the `copy-struct` expression is an immediate instance of *`struct-id`*. If *`struct-expr`* does not produce an instance of *`struct-id`*, the `exn:fail:contract` exception is raised.

If any *`accessor-id`* is not bound to an accessor of *`struct-id`* (according to the expansion-time information associated with *`struct-id`*), or if the same *`accessor-id`* is used twice, then a syntax error is raised.

`(define-struct/properties` *`id (field-id`* `···`*`) ((prop-expr val-expr)`* `···`*`) [inspector-expr]`*`)`
SYNTAX

Like `define-struct`, but properties (see §4.3 in *PLT MzScheme: Language Manual*) can be attached to the structure type. Each *`prop-expr`* should produce a structure-type property value, and each *`val-expr`* produces the corresponding value for the property.

Example:

```
(define-struct/properties point (x y)
  ([prop:custom-write (lambda (p port write?)
                        (fprintf port "(~a, ~a)"
                                 (point-x p)
                                 (point-y p)))]))

(display (make-point 1 2)) ; prints (1, 2)
```

`(make-->vector` *`struct-id`*`)` SYNTAX

This form builds a function that accepts a struct instance (matching *`struct-id`*) and provides a vector of the fields of the struct.

42. stxparam.ss: Syntax Parameters

To load: `(require (lib "stxparam.ss"))`

`(define-syntax-parameter` *`identifier expr`*`)` SYNTAX

Binds *`identifier`* 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 *`identifier`* 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 *`identifier`* can be used as a macro. If *`expr`* produces a `rename-transformer` result, then *`identifier`* can be used as a macro that expands to a use of the target identifier, but `syntax-local-value` of *`identifier`* does not produce the target's value.

`(syntax-parameterize` *`((identifier expr) ···) body-expr ···`*[1]`)` SYNTAX

Each *`identifier`* 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 *`identifier`*.

If an *`expr`* produces a procedure of one argument or a `make-set!-transformer` result, then its *`identifier`* can be used as a macro during the expansion of the *`body-expr`*s. If *`expr`* produces a `rename-transformer` result, then *`identifier`* can be used as a macro that expands to a use of the target identifier, but `syntax-local-value` of *`identifier`* does not produce the target's value.

`(syntax-parameter-value` *`id-stx`*`)` PROCEDURE

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.

`(make-parameter-rename-transformer` *`id-stx`*`)` PROCEDURE

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 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 MzScheme's `make-rename-transformer`.

43. surrogate.ss: Proxy-like Design Pattern

To load: `(require (lib "surrogate.ss"))`

This library provides an abstraction for building an instance of the proxy design pattern. The pattern consists of two objects, a *host* and a *surrogate* object. The host object delegates method calls to its surrogate object. Each host has a dynamically assigned surrogate, so an object can completely change its behavior merely by changing the surrogate.

The library provides a form, *`surrogate`*:

`(surrogate` *`method-spec ...`*`)` SYNTAX

where

```
method-spec ::== (method-name arg-spec ...)
 | (override method-name arg-spec ...)
 | (override-final method-name (lambda () default-expr) arg-spec ...)
arg-spec ::==
 | (id ...)
 | id
```

If neither `override` nor `override-final` is specified for a *`method-name`*, then `override` is assumed. Use `override`

The surrogate form produces four values: a host mixin (a procedure that accepts and returns a class), a host interface, a surrogate class, and a surrogate interface, in that order.

The host mixin adds one additional field, *`surrogate`*, to its argument and a getter method, *`get-surrogate`*, and a setter method, *`set-surrogate`*, for changing the field. The *`set-surrogate`* form accepts instances the class returned by the form or `#f`, and updates the field with its argument. Then, it calls the *`on-disable-surrogate`* on the previous value of the field and *`on-enable-surrogate`* for the new value of the field. The *`get-surrogate`* method returns the current value of the field.

The host mixin has a single overriding method for each *`method-name`* in the *`surrogate`* form. Each of these methods is defined with a case-lambda with one arm for each *`arg-spec`*. Each arm has the variables as arguments in the *`arg-spec`*. The body of each method tests the *`surrogate`* field. If it is `#f`, the method just returns the result of invoking the super or inner method. If the *`surrogate`* field is not `#f`, the corresponding method of the object in the field is invoked. This method receives the same arguments as the original method, plus two extras. The extra arguments come at the beginning of the argument list. The first is the original object. The second is a procedure that calls the super or inner method (*i.e.*, the method of the class that is passed to the mixin or an extension, or the method in an overriding class), with the arguments that the procedure receives.

The host interface has the names *`set-surrogate`*, *`get-surrogate`*, and each of the *`method-name`*s in the original form.

The surrogate class has a single public method for each *`method-name`* in the *`surrogate`* form. These methods are invoked by classes constructed by the mixin. Each has a corresponding method signature, as described in the above paragraph. Each method just passes its argument along to the super procedure it receives.

Note: if you derive a class from the surrogate class, do not both call the `super` argument and the super method of the surrogate class itself. Only call one or the other, since the default methods call the `super` argument.

Finally, the interface contains all of the names specified in surrogate's argument, plus *`on-enable-surrogate`* and *`on-disable-surrogate`*. The class returned by *`surrogate`* implements this interface.

44. tar.ss: Creating `tar` Files

To load: `(require (lib "tar.ss"))`

This library provides a facility for creating `tar` files. It creates tar files in USTAR format that are identical to files that the Unix utility `pax` generates. Note that the USTAR format imposes limits on path lengths. The resulting archives contain only directories and files (symbolic links are followed), and owner information is not preserved; the owner that is stored in the archive is always 'root'.

`(tar` *`tar-file path`* `...)` PROCEDURE

Creates *`tar-file`*, which holds the complete content of all *`path`*s. The given *`path`*s are all expected to be relative path names of existing directories and files (i.e., relative to the current directory). If a nested path is provided as a *`path`*, its ancestor directories are also added to the resulting tar file, up to the current directory (using *`pathlist-closure`*; see §11.3.3 in *PLT MzScheme: Language Manual*).

`(tar->output` *`paths`* [*`output-port`*]`)` PROCEDURE

Packages each of the given *`paths`* in a tar format archive that is written directly to the *`output-port`* or to the current output port if *`output-port`* is not given. Also, the specified *`paths`* are included as-is; if a directory is specified, its content is not automatically added, and nested directories are added without parent directories.

(See also §51.)

45. thread.ss: Thread Utilities

To load: `(require (lib "thread.ss"))`

`(coroutine` *`proc`*`)` PROCEDURE

Returns a coroutine object to encapsulate a thread that runs only when allowed. The *`proc`* procedure should accept one argument, and *`proc`* is run in the coroutine thread when `coroutine-run` is called. If `coroutine-run` returns due to a timeout, then the coroutine thread is suspended until a future call to `coroutine-run`. Thus, *`proc`* only executes during the dynamic extent of a `coroutine-run` call.

The argument to *`proc`* is a procedure that takes a boolean, and it can be used to disable suspends (in case *`proc`* has critical regions where it should not be suspended). A true value passed to the procedure enables suspends, and `#f` disables suspends. Initially, suspends are allowed.

`(coroutine?` *`v`*`)` PROCEDURE

Returns `#t` if *`v`* is a coroutine produced by `coroutine`, `#f` otherwise.

`(coroutine-run` *`timeout-secs coroutine`*`)` PROCEDURE

Allows the thread associated with *`coroutine`* to execute for up to *`timeout-secs`*. If *`coroutine`*'s procedure disables suspends, then the coroutine can run arbitrarily long until it re-enables suspends.

The `coroutine-run` procedure returns `#t` if *`coroutine`*'s procedure completes (or if it completed earlier), and the result is available via `coroutine-result`. The `coroutine-run` procedure returns returns `#f` if *`coroutine`*'s procedure does not complete before it is suspended after *`timeout-secs`*. If *`coroutine`*'s procedure raises an exception, then it is re-raised by `coroutine-run`.

`(coroutine-result` *`coroutine`*`)` PROCEDURE

Returns the result for *`coroutine`* if it has completed with a value (as opposed to an exception), `#f` otherwise.

`(coroutine-kill` *`coroutine`*`)` PROCEDURE

Forcibly terminates the thread associated with *`coroutine`* if it is still running, leaving the coroutine result unchanged.

`(consumer-thread` *`f`* [*`init`*]`)` PROCEDURE

Returns two values: a thread descriptor for a new thread, and a procedure with the same arity as *`f`*.[1] When the returned procedure is applied, its arguments are queued to be passed on to *`f`*, and void is immediately returned. The thread

[1] The returned procedure actually accepts any number of arguments, but immediately raises `exn:fail:contract:arity` if *`f`* cannot accept the provided number of arguments.

created by `consumer-thread` dequeues arguments and applies *f* to them, removing a new set of arguments from the queue only when the previous application of *f* has completed; if *f* escapes from a normal return (via an exception or a continuation), the *f*-applying thread terminates.

The *init* argument is a procedure of no arguments; if it is provided, *init* is called in the new thread immediately after the thread is created.

(run-server *port-k conn-proc conn-timeout* [*handler-proc listen-proc close-proc accept-proc accept/break-proc*]) PROCEDURE

Executes a TCP server on the port indicated by *port-k*. When a connection is made by a client, *conn-proc* is called with two values: an input port to receive from the client, and an output port to send to the client.

Each client connection is managed by a new custodian, and each call to *conn-proc* occurs in a new thread (managed by the connection's custodian). If the thread executing *conn-proc* terminates for any reason (e.g., *conn-proc* returns), the connection's custodian is shut down. Consequently, *conn-proc* need not close the ports provided to it. Breaks are enabled in the connection thread if breaks are enabled when *run-server* is called.

To facilitate capturing a continuation in one connection thread and invoking it in another, the parameterization of the `run-server` call is used for every call to *handler-proc*. In this parameterization and for the connection's thread, the `current-custodian` parameter is assigned to the connection's custodian.

If *conn-timeout* is not `#f`, then it must be a non-negative number specifying the time in seconds that a connection thread is allowed to run before it is sent a break signal. Then, if the thread runs longer than `(*` *conn-timeout* `2)` seconds, then the connection's custodian is shut down. If *conn-timeout* is `#f`, a connection thread can run indefinitely.

If *handler-proc* is provided, it is passed exceptions related to connections (i.e., exceptions not caught by *conn-proc*, or exceptions that occur when trying to accept a connection). The default handler ignores the exception and returns void.

The *listen-proc*, *close-proc*, *accept-proc* and *accept/break-proc* arguments default to the `tcp-listen`, `tcp-close`, `tcp-accept`, and `tcp-accept/enable-break` procedures, respectively. The `run-server` function calls these procedures without optional arguments. Provide alternate procedures to use an alternate communication protocol (such as SSL) or to supply optional arguments in the use of `tcp-listen`.

The *run-server* procedure loops to serve client connections, so it never returns. If a break occurs, the loop will cleanly shut down the server, but it will not terminate active connections.

46. trace.ss: Tracing Top-level Procedure Calls

To load: `(require (lib "trace.ss"))`

This library mimics the tracing facility available in Chez Scheme™.

`(trace variable ···)` SYNTAX

Each *variable* must be bound to a procedure in the environment of the `trace` expression. Each *variable* is `set!`ed to a new procedure that traces procedure calls and returns by printing the arguments and results of the call. If multiple values are returned, each value is displayed starting on a separate line.

When traced procedures invoke each other, nested invocations are shown by printing a nesting prefix. If the nesting depth grows to ten and beyond, a number is printed to show the actual nesting depth.

The `trace` form can be used on a variable that is already traced. In this case, assuming that the variable's value has not been changed, `trace` has no effect. If the variable has been changed to a different procedure, then a new trace is installed.

Tracing respects tail calls to preserve loops, but its effect may be visible through continuation marks. When a call to a traced procedure occurs in tail position with respect to a previous traced call, then the tailness of the call is preserved (and the result of the call is not printed for the tail call, because the same result will be printed for an enclosing call). Otherwise, however, the body of a traced procedure is not evaluated in tail position with respect to a call to the procedure.

The value of a `trace` expression is the list of names (as symbols) specified for tracing.

`(untrace variable ···)` SYNTAX

Undoes the effects of the `trace` form for each *variable*, `set!`ing each *variable* back to the untraced procedure, but only if the current value of *variable* is a traced procedure. If the current value of a *variable* is not a procedure installed by `trace`, then the variable is not changed.

The value of an `untrace` expression is the list of names (as symbols) restored to their untraced definitions.

47. traceld.ss: Tracing File Loads

To load: `(require (lib "traceld.ss"))`

This library does not define any procedures or syntax. Instead, **traceld.ss** is imported at the top-level for its side-effects. The trace library installs a new load handler and load extension handler to print information about the files that are loaded. These handlers chain to the current handlers to perform the actual loads. Trace output is printed to the port that is the current error port when the library is loaded.

Before a file is loaded, the tracer prints the file name and "time" (as reported by the procedure `current-process-milliseconds`) when the load starts. Trace information for nested loads is printed with indentation. After the file is loaded, the file name is printed with the "time" that the load completed.

If a **_loader** extension is loaded (see §14.1 in *PLT MzScheme: Language Manual*), the tracer wraps the returned loader procedure to print information about libraries requested from the loader. When a library is found in the loader, the thunk procedure that extracts the library is wrapped to print the start and end times of the extraction.

48. transcr.ss: Transcripts

To load: `(require (lib "transcr.ss"))`

MzScheme's built-in `transcript-on` and `transcript-off` always raise `exn:fail:unsupported`. The **transcr.ss** library provides working versions of `transcript-on` and `transcript-off`.

49. unit.ss: Core Units

To load: `(require (lib "unit.ss"))`

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

In some ways, a unit resembles a module (see Chapter 5 in *PLT MzScheme: Language Manual*), but units and modules serve different purposes overall. A unit encapsulates a pluggable component—code that relies, for example, on "some function *f* from a source to be determined later." In contrast, if a module imports a function, the import is "*the* function *f* provided by the specific module *m*." Moreover, a unit is a first-class value that can be multiply instantiated, each time with different imports, whereas a module's context is fixed. Finally, because a unit's interface is separate from its implementation, units naturally support mutually recursive references across unit boundaries, while module imports must be acyclic.

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

The core system is described in this chapter, and defined by the **unit.ss** library. The signature system is described in §50, and defined by **unitsig.ss**. Details about mixing core and signed units are presented in §50.9 (using procedures from **unitsig.ss**).

49.1 Creating Units

The `unit` form creates a unit:

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

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

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

The first *exportage* form exports the binding defined as *variable* in the unit body using the external name *variable*. The second form exports the binding defined as *internal-variable* using the external name *external-variable*. The external variables from an export clause must be distinct.

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

Examples

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

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

The unit defined below is similar, except that it exports the variable *x* instead of returning the value:

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

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

```
(define database@
  (unit
    (import show-message)
    (export insert lookup)

     (define table (list))
     (define insert
       (lambda (name info)
         (set! table (cons (cons name info) table))))
     (define lookup
       (lambda (name)
         (let ([data (assoc name table)])
           (if data
                (cdr data)
                (show-message "info not found")))))
     insert))
```

[1]The detection of unit definitions is the same as for internal definitions (see §2.8.5 in *PLT MzScheme: Language Manual*). Thus, the define and define-struct forms can be used for definitions.

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

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

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

49.2 Invoking Units

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

```
(invoke-unit unit-expr import-expr ···)
```

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

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

Examples

These examples use the definitions from the earlier unit examples in §49.1.

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

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

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

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

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

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

`(define-values/invoke-unit (`*`export-id`* `···)` *`unit-expr`* `[`*`prefix import-id`* `···])` SYNTAX

This form is similar to `invoke-unit`. However, instead of returning the value of the unit's initialization expression, `define-values/invoke-unit` expands to a `define-values` expression that binds each identifier *`export-id`* to the value of the corresponding variable exported by the unit. At run time, if the unit does not export all of the *`export-id`*s, the `exn:fail:unit` exception is raised.

If *`prefix`* is specified, it must be either `#f` or an identifier. If it is an identifier, the names defined by the expansion of `define-values/invoke-unit` are prefixed with *`prefix`*`:`.

Example:

```
(define x 3)
(define y 2)
(define-values/invoke-unit (c)
  (unit (import a b) (export c)
    (define c (- a b)))
  ex
  x y)
ex:c ; ⇒ 1
```

`(namespace-variable-bind/invoke-unit (`*`export-id`* `···)` *`unit-expr`* `[`*`prefix import-id`* `···])` SYNTAX

This form is like `define-values/invoke-unit`, but the expansion is a sequence of calls to `namespace-set-variable-value!` instead of a `define-values` expression. Thus, when it is evaluated, a *`namespace-variable-bind/invoke-unit`* expression binds top-level variables in the current namespace.

49.3 Linking Units and Creating Compound Units

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

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

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

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

tag is
  identifier
```

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

- The `import` clause imports variables into the compound unit. These imported variables are used as imports to the compound unit's sub-units.

- The `link` clause specifies how the compound unit is created from sub-units. A unique *`tag`* is associated with each sub-unit, which is specified using an arbitrary expression. Following the unit expression, each *`linkage`* specifies a variable using the *`variable`* form or the `(`*`tag variable`*`)` form. In the former case, the *`variable`* must occur in the `import` clause of the `compound-unit` expression; in the latter case, the *`tag`* must be defined in the same `compound-unit` expression. The `(`*`tag variable`* ···`)` form is a shorthand for multiple adjacent clauses of the second form with the same *`tag`*.

- The `export` clause re-exports variables from the compound unit that were originally exported from the sub-units. The *`tag`* part of each `export` sub-clause specifies the sub-unit from which the re-exported variable is drawn. The *`exportage`*s specify the names of variables exported by the sub-unit to be re-exported.

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

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

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

Examples

These examples use the definitions from the earlier unit examples in §49.1.

The following `compound-unit` expression creates a (probably useless) renaming wrapping around the unit bound to *`f2@`*:

```
(define f3@
  (compound-unit
    (import)
    (link [A (f2@)])
    (export (A (x A:x)))))
```

The only difference between *`f2@`* and *`f3@`* is that *`f2@`* exports a variable named *`x`*, while *`f3@`* exports a variable named *`A:x`*.

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

```
(define program@
  (compound-unit
    (import)
    (link (GRAPHICS (graphics@))
```

```
            (DATABASE (database@ (INTERFACE show-message)))
            (INTERFACE (interface@ (DATABASE insert lookup)
                                   (GRAPHICS make-window make-button))))
      (export)))
```

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

49.4 Unit Utilities

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

50. unitsig.ss: Units with Signatures

To load: `(require (lib "unitsig.ss"))`

The unit syntax presented in §49 poses a serious notational problem: each variable that is imported or exported must be separately enumerated in many `import`, `export`, and `link` clauses. Consider the phone book program example from §49.3: a realistic *`graphics@`* unit would contain many more procedures than *`make-window`* and *`make-button`*, and it would be unreasonable to enumerate the entire graphics toolbox in every client module. Future extensions to the graphics library are likely, and while the program must certainly be re-compiled to take advantage of the changes, the programmer should not be required to change the program text in every place that the graphics library is used.

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

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

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

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

50.1 Importing and Exporting with Signatures

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

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

Although the collection of variables to be exported from a `unit/sig` expression is specified by a signature rather than an immediate sequence of variables,[1] variables are exported in a `unit/sig` form in the same way as in the `unit` form. The *export signature* of a signed unit is the collection of names exported by the unit.

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

Example:

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

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

Suppose that the procedures in `graphics^` were named `add` and `remove` rather than `add-pixel` and `remove-pixel`. In this case, the `gravity@` unit cannot import both the `arithmetic^` and `graphics^` signatures as above, because the name `add` would be ambiguous in the unit body. To solve this naming problem, the imports of a signed unit can be distinguished by providing prefix tags:

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

Details for the syntax of signatures are in §50.2. The full `unit/sig` syntax is described in §50.3.

50.2 Signatures

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

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

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

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

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

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

  The actual structure type can contain additional fields; if a field identifier is omitted, the corresponding selector and setter names are not added to the signature. Optional *omission* specifications can omit other kinds of names: `-selectors` omits all field selector variables. `-setters` omits all field setter variables, and `(-variable)` omits a specific generated *variable*.

In a unit importing the signature, the *base-identifier* is also bound to expansion-time information about the structure type (see §12.6.4 in *PLT MzScheme: Language Manual*). The expansion-time information records the descriptor, constructor, predicate, field accessor, and field mutator bindings from the signature. It also indicates that the accessor and mutator sets are potentially incomplete (so `match` works with the structure type, but `shared` does not), either because the signature omits fields, or because the structure type is derived from a base type (which cannot be declared in a signature, currently).

- The `open` form copies all of the elements of another signature into the new signature description.
- The `unit` form creates a sub-signature within the new signature. A signature that includes a `unit` clause corresponds to a signed compound unit that exports an embedded unit. (Embedded units are described in §50.6 and §50.7.)

The names of all elements in a signature must be distinct.[2] Two signatures *match* when they contain the same element names, and when a name in both signatures is either a variable name in both signatures or a sub-signature name in both signatures such that the sub-signatures match. The order of elements within a signature is not important. A source signature *satisfies* a destination signature when the source signature has all of the elements of the destination signature, but the source signature may have additional elements.

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

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

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

```
(let-signature identifier signature body-expr ···¹)
```

For various purposes, signatures must be flattened into a linear sequence of variables. The flattening operation is defined as follows:

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

50.3 Signed Units

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

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

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

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

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

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

The *import-element*s specify imports for the signed unit. The names bound within the *signed-unit-body-expr*s to imported bindings are constructed by flattening the signatures according to the algorithm in §50.2:

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

50.4 Linking with Signatures

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

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

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

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

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

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

The export signature of a signed compound unit is implicitly specified by the `export` clause. In the above example, the *`model@`* compound unit exports a *`go`* variable, so its export signature is the same as *`gravity^`*. More forms for exporting are described in §50.6.

50.5 Restricting Signatures

As explained in §50.4, the signature checking for a linkage requires that a provided signature *exactly* matches the corresponding import signature. At first glance, this requirement appears to be overly strict; it might seem that the provided signature need only *satisfy* the imported signature. The reason for requiring an exact match at linkages is that a `compound-unit/sig` expression is expanded into a `compound-unit` expression. Thus, the number and order of the variables used for linking must be fully known at compile time.

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

In the example from §50.4, both *`calculus@`* and *`gravity@`* import numerical procedures, so both import the *`arithmetic^`* signature. However, *`calculus@`* does not actually need the *`power`* procedure to implement *`integrate`*; therefore, *`calculus@`* could be as effectively implemented in the following way:

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

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

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

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

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

50.6 Embedded Units

Signed compound units can re-export variables from linked units in the same way that core compound units can re-export variables. The difference in this case is that the collection of variables that are re-exported determines an export signature for the compound unit. Using certain export forms, such as the `open` form instead of the *var* form (see §50.7), makes it easier to export a number of variables at once, but these are simply shorthand notations.

Signed compound units can also export entire units as well as variables. Such an exported unit is an *embedded unit* of the compound unit. Extending the example from §50.5, the entire *gravity@* unit can be exported from *model@* using the `unit` export form:

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

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

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

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

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

Example:

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

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

50.7 Signed Compound Units

The `compound-unit/sig` form links multiple signed units into a new signed compound unit:

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

```
linkage is
  unit-path

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

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

export-element is one of
  (var (simple-unit-path variable))
  (var (simple-unit-path variable) external-variable)
  (open unit-path)
  (unit unit-path)
  (unit unit-path variable)

tag is
  identifier
```

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

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

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

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

There are five different forms for specifying exports:

- The `(var (`*`unit-path variable`*`))` form exports *`variable`* from the unit referenced by *`unit-path`*. The export signature for the signed compound unit includes a *`variable`* element.
- The `(var (`*`unit-path variable`*`) `*`external-variable`*`)` form exports *`variable`* from the unit referenced by *`unit-path`*. The export signature for the signed compound unit includes an *`external-variable`* element.

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

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

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

- The `(unit` *`unit-path identifier`*`)` form exports an embedded unit like `(unit` *`unit-path`*`)` form, but *`identifier`* is used for the name of the sub-signature in the compound unit's export signature.

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

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

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

50.8 Invoking Signed Units

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

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

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

If the invoked unit requires no imports, the `invoke-unit/sig` form is used in the same way as `invoke-unit`. Otherwise, the *`invoke-linkage`* signatures must match the import signatures of the signed unit to be invoked. If the signatures match, then variables in the environment of the `invoke-unit/sig` expression are used for immediate linking; the variables used for linking are the ones with names corresponding to the flattened signatures. The signature flattening algorithm is specified in §50.2; when the `(`*`identifier`*` : `*`signature`*`)` form is used, *`identifier`*`:` is prefixed onto each variable name in the flattened signature and the prefixed name is used.

`(define-values/invoke-unit/sig` *`signature unit/sig-expr`* [*`prefix invoke-linkage`* ···]`)` SYNTAX

This form is the signed-unit version of `define-values/invoke-unit`. The names defined by the expansion of `define-values/invoke-unit/sig` are determined by flattening the *`signature`* specified before *`unit-expr`*, then adding the *`prefix`* (if any). See §50.2 for more information about signature flattening.

Each *`invoke-linkage`* is either *`signature`* or `(`*`identifier`*` : `*`signature`*`)`, as in `invoke-unit/sig`.

`(namespace-variable-bind/invoke-unit/sig` *`signature unit/sig-expr`* [*`prefix invoke-linkage`* ···]`)` SYNTAX

This form is the signed-unit version of *`namespace-variable-bind/invoke-unit`*. See also `define-values/invoke-unit/sig`.

`(provide-signature-elements` *`signature`*`)` SYNTAX

Exports from a module every name in the flattened form of *`signature`*.

50.9 Extracting a Primitive Unit from a Signed Unit

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

The names exported by the primitive unit correspond to the flattened export signature of the signed unit; see §50.2 for the flattening algorithm.

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

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

50.10 Adding a Signature to Primitive Units

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

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

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

50.11 Expanding Signed Unit Expressions

The `unit/sig`, `compound-unit/sig`, and `invoke-unit/sig` forms expand into expressions using the `unit`, `compound-unit`, and `invoke-unit` forms, respectively.

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

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

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

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

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

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

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

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

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

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

`(signature->symbols` *`name`*`)` SYNTAX

Expands to the “exploded” version (see §50.11) of the signature bound to *`name`* (where *`name`* is an unevaluated identifier).

51. zip.ss: Creating `zip` Files

To load: `(require (lib "zip.ss"))`

This library provides a facility for creating zip files, which are compatible with both Windows and Unix and Mac OS X. The actual compression is implemented by `deflate` (see Chapter 14). The most useful entry point for this library is `zip`.

`(zip` *`zip-file path`* `···)` PROCEDURE

Creates *`zip-file`*, which holds the complete content of all *`path`*s. The given *`path`*s are all expected to be relative path names of existing directories and files (i.e., relative to the current directory). If a nested path is provided as a *`path`*, its ancestor directories are also added to the resulting zip file, up to the current directory (using *`pathlist-closure`*; see §11.3.3 in *PLT MzScheme: Language Manual*). Files are packaged as usual for zip files, including permission bits for both Windows and Unix and Mac OS X. The permission bits are determined by `file-or-directory-permissions` (§11.3.3 in *PLT MzScheme: Language Manual*), so it does not preserve the distinction between owner/group/other permissions; also, symbolic links are always followed.

`(zip->output` *`paths`* [*`output-port`*]`)` PROCEDURE

Zips each of the given *`paths`*, and packages it as a zip "file" that is written directly to the *`output-port`* or to the current output port if *`output-port`* is not given. Also, the specified *`paths`* are included as-is; if a directory is specified, its content is not automatically added, and nested directories are added without parent directories.

`(zip-verbose` [*`on?`*]`)` PROCEDURE

A parameter that controls output during a zip operation. Setting this parameter to a true value will cause `zip` to display (on the current error port) the filename that is currently being compressed.

(See also §44.)

License

GNU Library General Public License

Version 2, June 1991

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

Preamble

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

   a) The modified work must itself be a software library.
   b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.
   c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.
   d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.
   (For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Library General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Index