PLT MzLib: Libraries Manual

PLT (scheme@plt-scheme.org)

Version 203
December 2002

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, 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 typest 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 December 19, 2002.

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:

- **awk.ss** — AWK-like syntax
- **class.ss** — object system
- **cm.ss** — compilation manager
- **cmdline.ss** — command-line parsing
- **compat.ss** — compatibility procedures and syntax
- **compile.ss** — bytecode compilation
- **contracts.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`
- **list.ss** — list-processing procedures
- **match.ss** — pattern matching
- **math.ss** — arithmetic procedures and constants
- **pconvert.ss** — print values as expressions
- **pregexp.ss** – Perl-style regular expressions
- **pretty.ss** — pretty-printer
- **restart.ss** — stand-alone MzScheme emulator
- **sendevent.ss** — AppleEvents
- **shared.ss** — graph constructor syntax
- **spidey.ss** — MrSpidey annotation syntax
- **string.ss** — string-processing procedures
- **thread.ss** — thread utilities
- **trace.ss** — function tracing
- **traceld.ss** — file-load tracing
- **transcr.ss** — transcripts
- **unit.ss** — component system
- **unitsig.ss** — component system with signatures

2. awk.ss: Awk-like Syntax

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.

3. class.ss: Classes and Objects

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

- *inheritance*: An object of a derived class 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*: A method declared in a superclass can be redeclared in the derived class. References to the overridden method in the superclass use the implementation 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").

3.1 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:
    (public push! pop! none? 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) (newline))

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

(define fancy-stack%
  (class stack%
    ; Declare override
    (override print-name)

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

    (define (print-name)
      (display name)
      (display ", Esq.")
      (newline))

    (super-instantiate ())))

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

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

    ; Always supply name
    (super-instantiate () (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?)
        (rename [std-pop! pop!])
        (override pop!)
        (define (pop!) (if (none?) #f (std-pop!)))
        (super-instantiate ())))
    (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 *print-name* to add an "Esq." suffix.
- 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 call (**super-instantiate** ...) 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 *named-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 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 *named-stack%*, *double-stack%*, and *safe-stack%* inherit the *push!* method of *stack%*, it is declared with **inherit** only in *double-stack%*; new declarations in *named-stack%* and *safe-stack%* do not need to refer to *push!*, so the inheritance does not need to be declared. Similarly, only *safe-stack%* needs to declare (**inherit** *none?*).

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

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

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

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

3.2 Creating Interfaces

The **interface** form creates a new interface:

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

All of the *variable*s must be distinct.

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

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

3.3 Creating Classes

The built-in class `object%` has no methods fields, implements only its own interface, `(class->interface object%)`. All other classes are derived from `object%`.

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

(**class***/**names** *local-names superclass-expr* (*interface-expr* ···)

class-clause
$\cdots$)

local-names is one of
(*this-variable*)
(*this-variable super-instantiate-variable*)
(*this-variable super-instantiate-variable super-make-object-variable*)

class-clause is one of
(**init** *init-declaration* $\cdots$)
(**init-field** *init-declaration* $\cdots$)
(**field** *field-declaration* $\cdots$)
(**inherit-field** *optionally-renamed-variable* $\cdots$)
(**init-rest** *variable*)
(**init-rest**)
(**public** *optionally-renamed-variable* $\cdots$)
(**override** *optionally-renamed-variable* $\cdots$)
(**public-final** *optionally-renamed-variable* $\cdots$)
(**override-final** *optionally-renamed-variable* $\cdots$)
(**private** *variable* $\cdots$)
(**inherit** *optionally-renamed-variable* $\cdots$)
(**rename** *renamed-variable* $\cdots$)
method-definition
definition
expr
(**begin** *class-clause* $\cdots$)

init-declaration is one of
variable
(*optionally-renamed-variable*)
(*optionally-renamed-variable default-value-expr*)

field-declaration is
(*optionally-renamed-variable default-value-expr*)

optionally-renamed-variable is one of
variable
renamed-variable

renamed-variable is
(*internal-variable external-variable*)

method-definition is
(**define-values** (*variable*) *method-procedure*)

method-procedure is
(**lambda** *formals expr* $\cdots^1$)
(**case-lambda** (*formals expr* $\cdots^1$) $\cdots$)
(**let-values** (((*variable*) *method-procedure*) $\cdots$) *method-procedure*)
(**letrec-values** (((*variable*) *method-procedure*) $\cdots$) *method-procedure*)
(**let-values** (((*variable*) *method-procedure*) $\cdots^1$) *variable*)
(**letrec-values** (((*variable*) *method-procedure*) $\cdots^1$) *variable*)

The *this-variable*, *super-instantiate-variable*, and *super-make-object-variable* variables (usually `this`, **super-instantiate**, and `super-make-object`) are bound in the rest of the `class*/names` expression, excluding *superclass-expr* and the *interface-expr*s. In instances of the new class, *this-variable* (i.e., `this`) is bound to the object itself; *super-instantiate-variable* (i.e., **super-instantiate**) is bound to a form that must be used (once) to initialize fields in the superclass (see §3.4); *super-make-object-variable* (i.e., `super-make-object`) can be used instead of *super-instantiate-variable* to initialize superclass fields. See §3.4 for more information about *super-instantiate-variable* and *super-make-object-variable*.

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

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

The *class-clause*s define initialization arguments, public and private fields, and public and private methods. For each *variable* or *optionally-renamed-variable* in a **public**, **override**, **public-final**, **override-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 §3.4).

The result of a `class*/names` expression is a new class, derived from the specified superclass and implementing the specified interfaces. Instances of the class are created with the **instantiate** form or `make-object` procedure, as described in §3.4.

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.

The **class*** form is like **class*/names**, but omits *local-names* and always uses the name `this`, **super-instantiate**, and `super-make-object`:

(**class*** *superclass-expr* (*interface-expr* ...)
  class-clause
  ...)

The **class** form further omits the *interface-expr*s, for the case that none are needed:

(**class** *superclass-expr*
  class-clause
  ...)

The **public***, **public-final***, **override***, **override-final***, and **private*** forms abbreviate a **public**, **public-final**, **override**, **override-final**, or **private** declaration and a sequence of definitions:

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

etc.

The **define/public**, **define/public-final**, **define/override**, **define/override-final**, and **define/private** forms similarly abbreviate a **public**, **override**, or **private** declaration with a definition:

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

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

etc.

3.3.1 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 a 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 §3.4.

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: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 §3.4.)

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:object` exception is raised. Similarly, if extra by-position arguments are provided to a class with an **init-rest** clause, the `exn:object` exception is raised.

Unused (by-name) arguments are be propagated to the superclass, as described in §3.4. 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 §3.4 for further details.

See also §3.3.3.3 for information about internal and external names.

3.3.2 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 §3.5). Fields declared with **define-values** are accessible only within the class.

A field declared with **init-field** is both a public field an an initialization variable. See §3.3.1 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: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 §3.4 for more information.

See also §3.3.3.3 for information about internal and external names.

3.3.3 Methods

3.3.3.1 Method Definitions

Each **public**, **override**, **public-final**, **override-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**, **override**, **public-final**, **override-final**, **private** clauses and their corresponding definitions (among themselves, and with respect to other clauses in the class) does not matter.

As shown in §3.3, 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 *variable* must be one of the *variable*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** or **public-final** introduces a new method into a class. The method must not be present already in the superclass, otherwise the `exn:object` exception is raised when the class expression is evaluated. A method declared with **public-final** cannot be overridden in a subclass.

A method declared with **override** or **override-final** overrides a definition already present in the superclass. If the method is not already present, the `exn:object` exception is raised when the class expression is evaluated. A method declared with **override-final** cannot be overridden 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.

3.3.3.2 Inherited and Superclass Methods

Each **inherit** and **rename** clause declares one or more methods that are not defined in the class, but must be present in the superclass. Methods declared with **inherit** are subject to overriding, while methods declared with **rename** are not. Methods that are present in the superclass but not declared with **inherit** or **rename** 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** is not present in the superclass, the `exn:object` exception is raised when the class expression is evaluated.

3.3.3.3 Internal and External Names

Each method declared with **public**, **override**, **public-final**, **override-final**, **inherit**, and **rename** can have separate internal and external names when (*internal-variable external-variable*) is used for declaring the method. The internal name is used to access the method directly within the class expression, while the external name is used with **send** and **generic** (see §3.5). If a single *variable* is provided for a method declaration, the variable is used for both the internal and external names.

Method inheritance and overriding are based external names, only. Separate internal and external names are *required* for **rename**, because its purpose is to provide access to the superclass's version of an overridden method.

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 variable and an external variable, and it is possible to use the same identifier as internal and external variables 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, the set of all internal variables must be distinct, and set of of external variables must be distinct for each of the method, field, and initialization-argument categories.

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** *variable* ···)

This form binds each *variable* so that, within the scope of the definition, each use of each *variable* 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 *variable*s are accessible only in the scope of the **define-local-member-name** declaration.

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. The `interface->method-names` procedure (see §3.6) does not expose hidden names.

Example:

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

              (send o m) ; ⇒ 10
              o))

(send o m) ; ⇒ error: no method m
```

3.4 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 §3.3.1. If *class* is not a class, the `exn:application:type` exception is raised.

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

(**instantiate** *class-expr* (*by-pos-expr* ···) (*variable 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 *variable*.

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 invoking the form bound to *super-instantiate-variable* (usually **super-instantiate**):

(*super-instantiate-variable* (*by-position-super-init-expr* ···) (*variable by-name-super-init-expr* ···) ···)

or by calling the procedure bound to *super-make-object-variable* (usually `super-make-object`):

(*super-make-object-variable super-init-v* ···)

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** and `make-object`.

By-name initialization arguments to a class that have no matching initialization variable are implicitly added as by-name arguments to a *super-instantiate-variable* or *super-make-object-variable* 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: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:object` exception is raised in such a case. Also, if superclass initialization is invoked more than once, the `exn:object` exception is raised.

3.5 Field and Method Access

In expressions within a class definition, the initialization variables, fields, and methods of the class all part of the environment, as are the names bound to *super-instantiate-variable* and *super-make-object-variable*. 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!**.

3.5.1 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-variable arg-expr* ··· . *arg-list-expr*)

which calls the method in a way analogous to (`apply` *method-variable 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:application:type` exception is raised. If the object has no public method *method-name*, the `exn: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** ((*variable* (*object-expr method-name*)) · · ·)
expr · · ·[1])

Example:

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

which is the same as

```
(let ([s (make-object stack%)])
  (send s push! 10)
  (send s push! 9)
  (send s pop!))
```

3.5.2 Fields

Fields are accessed from outside an object through a field accessor or mutator procedure produced by **class-field-accessor** or **class-field-mutator**:

- (**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 an 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.

3.5.3 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: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 §3.3.3.3).

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.

3.6 Object Utilities

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

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

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

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

(`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) defines an instance variable with the name *symbol*, #`f` otherwise.

`(interface->method-names` *interface*`)` returns a list of symbols for the instance variable names in *interface* (including instance variables inherited from superinterfaces).

4. class100.ss: Version-100-Style Classes

The **class100**, **class100**∗, and **class100**∗**/names** forms provide a syntax close to that of **class**, **class**∗, and **class**∗**/names** in MzScheme versions 100 through 103, but with the semantics of the current **class.ss** system (see Chapter 3). 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**∗**/names** form creates a new class:

(**class100**∗**/names** *local-names superclass-expr* (*interface-expr* · · ·) *initialization-variables*
class100-clause
· · ·)

local-names is one of
(*this-variable super-make-object-variable*)
(*this-variable super-make-object-variable super-instanitate-variable*)

initialization-variables 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* · · ·)
(**private** *private-method-declaration* · · ·)
(**private-field** *private-var-declaration* · · ·)
(**inherit** *inherit-method-declaration* · · ·)
(**rename** *rename-method-declaration* · · ·)

public-method-declaration is one of
((*internal-variable external-variable*) *method-procedure*)
(*variable method-procedure*)

private-method-declaration is one of
(*variable method-procedure*)

private-var-declaration is one of
(*variable initial-value-expr*)
(*variable*)
variable

inherit-method-declaration is one of
  variable
  (*internal-instance-variable external-inherited-variable*)

rename-method-declaration is
  (*internal-variable external-variable*)

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

The **class100∗** macro avoids specifying *local-names*, instead implicitly binding `this` and `super-init` (and nothing for *super-instaniate-variable*).

(**class100∗** *superclass-expr* (*interface-expr* · · ·) *initialization-variables*
  class100-clause
  · · ·)

The **class100** macro omits both *local-names* and the *interface-expr*s:

(**class100** *superclass-expr initialization-variables*
  class100-clause
  · · ·)

(class100-asi *superclass instance-variable-clause* · · ·**)** SYNTAX

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

(class100∗-asi *superclass interfaces instance-variable-clause* · · ·**)** SYNTAX

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

5. class-old.ss: Version-100 Classes

This library provides the class system of MzScheme version 103; consult old MzScheme documentation for details.. It is not compatible with the newer class system implemented by **class.ss** and **class100.ss**.

6. cm.ss: Compilation Manager

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

Returns a procedure suitable as a value for the `current-load/use-compiled` parameter (see §7.4.1.6 in *PLT MzScheme: Language Manual*). The returned procedure 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 `current-eval`, `current-load`, and `current-namespace` is the same as when `make-compilation-manager-load/use-compiled-handler` was called; and
- either the source file is newer than the **.zo** file in the **compiled** subdirectory, or no **.dep** file exists next to the **.zo** file, or the version in the **.dep** does not match the result of (`version`), or one of the files listed in the **.dep** file has a timestamp newer than the one recorded in the **.dep** file.

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

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*).

(managed-compile-zo *file*) PROCEDURE

Compiles the given module source file to a **.zo**, installing a compilation-manager handler while the file is 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).

(manager-trace-handler *proc* [*procedure*])

A parameter whose value is a procedure to return a trace string for compilation-manager actions. The procedure receives a single string argument, and its result is ignored.

(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.

7. cmdline.ss: Command-line Parsing

(**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 vector of strings; typically, it is (`current-command-line-arguments`).

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 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:user` exception is raised.

- **once-any** — Only one flag specified in the set can be represented on the command line; i.e., the flags in the set are mutually exclusive. If the set is represented in the command line more than once, the `exn:user` exception is raised.
- **final** — Like **multi**, except that no other argument after the flag is treated as a flag.

A normal flag specification has four parts:

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

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

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:user` 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 leftover arguments. In this case, the values of the expressions with `=>` are passed on directly as arguments to `parse-command-line`. The *help-proc-expr* and *unknown-proc-expr* 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"        "Compile with optimization level 2"
                         (optimize-level 2)])
(multi
   [("-l" "–link-flags") lf ; flag takes one argument
                         "Add a flag for the linker" "flag"
                         (link-flags (cons lf (link-flags)))])
(args (filename) ; expects one command-line argument: a filename
  filename)) ; return a single filename to compile
```

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

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

When there are no more flags, the *finish-proc* procedure is called with a list of information accumulated for command-line flags (see below) and the remaining non-flag arguments from the command-line. The arity of the *finish-proc* procedure determines the number of non-flag arguments accepted and required from the command-line. For example, if *finish-proc* accepts either two or three arguments, then either one or two non-flag arguments must be provided on the command-line. The *finish-proc* procedure can have any arity (see §3.10.1 in *PLT MzScheme: Language Manual*) except `0` or a list of `0`s (i.e., the procedure must at least accept one or more arguments).

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

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

A *table* is a list of flag specification sets. Each set is represented as a list of two items: a mode symbol and a list of 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 §3.10.1 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 of strings used for constructing help information for the spec. The first string in the list describes the flag, and additional strings name the expected arguments for the flag. The number of extra help strings provided for a spec must match the number of arguments accepted by the spec's handler procedure.

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

```
(parse-command-line "compile" (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")])
    (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
```

8. compat.ss: Compatibility

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.

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

This is the same as `mergesort` (see §18) with the arguments reversed.

9. compile.ss: Compiling Files

(`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 `compile-file` function 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` function provided by the **compiler** collection's **compiler.ss** module.

10. contracts.ss: Contracts

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

This section describes two classes of contracts: contracts for flat values (described in section 10.1) and contracts for functions (described in section 10.2).

In addition, this section describes two forms for establishing a contract on a value (described in section 10.3).

10.1 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-named-contract` *type-name predicate*`)` PROCEDURE

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.

`(flat-named-contract-type-name` *flat-named-contract*`)` PROCEDURE

Extracts the type name from a *flat-named-contract*.

`(flat-named-contract-predicate` *flat-named-contract*`)` PROCEDURE

Extracts the predicate from a *flat-named-contract*.

In addition, this library provides many helper functions for constructing contracts.

`(union` *contract ...*`)` PROCEDURE

union accepts any number of predicates and at most one function contract and returns a contract that corresponds to the union of them all.

`(and/f` *predicate*`)` PROCEDURE

and/f accepts a list of predicates and returns a predicate that is the conjunction of those predicates.

`(or/f` *predicate ...*`)` PROCEDURE

or/f accepts a list of predicates and returns a predicate that is the disjuction of those predicates.

`(>=/c` *number*`)` PROCEDURE

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

`(<=/c` *number*`)` PROCEDURE

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

`(>/c` *number*`)` PROCEDURE

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

`(</c` *number*`)` PROCEDURE

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

`natural-number?` FLAT-CONTRACT

natural-number? returns #`t` if the input is a natural number and #`f` otherwise.

`false?` FLAT-CONTRACT

false? returns true if the input is #`f` and #`t` otherwise.

`printable?` FLAT-CONTRACT

printable? returns #`t` for any value that can be written out and read back in.

`any?` FLAT-CONTRACT

any? always returns #`t`.

`(symbols` *symbol ...*`)` PROCEDURE

symbols accepts any number of symbols and returns a predicate that checks for those symbols.

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

is-a?/c accepts a class or interface and returns a predicate that checks if objects are subclasses of the class or implement the interface.

`(implementation?/c` *interface*`)` PROCEDURE

implementation?/c accepts an interface and returns a predicate that checks if classes are implement the given interface.

`(subclass?/c` *class*`)` PROCEDURE

subclass?/c accepts a class and returns a predicate that checks if classes are subclasses of the original class.

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

listof accepts a flat contract and returns a predicate that checks for lists whose elements match the original predicate.

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

vectorof accepts a flat contract and returns a predicate that checks for vectors whose elements match the original predicate.

`(vector/p` *flat-contract ...*`)` FLAT-CONTRACT

vector/p accepts any number of flat contract and returns a predicate that checks for vectors. The number of elements in the vector must match the number of arguments supplied to *vector/p* and the elements of the vector must match the corresponding flat contract.

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

box/p accepts a flat contract and returns a flat contract that checks for boxes whose contents match *box/p*'s argument.

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

cons/p accepts two predicates and returns a predicate that checks for cons cells whose car and cdr correspond to *cons/p*'s two arguments.

`(list/p` *flat-contract ...*`)` PROCEDURE

list/p accepts an arbitrary number of arguments and returns a predicate that checks for lists whose length is the same as the number of arguments to *list/p* and whose elements match those arguments.

`mixin-contract` CONTRACT

mixin-contract is a contract that matches 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 ...*`)` PROCEDURE

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.

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

where *expr* is any Scheme 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 §14.3 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.

(->* *(expr ...) (expr ...)*) SYNTAX

(->* *(expr ...) expr (expr ...)*) 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. 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.

(->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.

`(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

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. 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.

10.3 Attaching Contracts to Scheme 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*) ...))
(*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 *expr*.

The **provide/contract** form treats modules as units of blame. The module that defines the provided variable is expected to meet the position (co-variant) positions of the contract. Each module that imports the provided variable must obey the negative (contract-variant) positions of the contract.

Only uses of the contracted variable outside the module are checked.

The **struct** form of a **provide/contract** clause provides a structure definition. Each field has a contract that dictates the contents of the fields.

`(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.

10.4 Contract Utility

`contract?` PREDICATE

The procedure *contract?* returns `#t` if its argument was constructed with one of the arrow constructors described earlier in this section, or if its argument is a procedure of arity `1`.

11. date.ss: Dates

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, 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.

12. deflate.ss: Deflating (Compressing) Data

(`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 return value is void.

13. defmacro.ss: Non-Hygienic Macros

(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 §3.10.1 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]).

14. etc.ss: Useful Procedures and Syntax

`(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:application:type` exception is raised.

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

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

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

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

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

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

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

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

(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 *target*ss are bound to the *expr*s. The environment of the *expr*s includes all of the *target*ss.
- (_ *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 an variable has a default value expression, then all (non-aggregate) variables after it must have default value expressions. A final aggregate variable can be used as in `lambda`, but it cannot be given a default value. Each default value expression is evaluated only if it is needed. The environment of each default value expression includes the preceding arguments.

For example:

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

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*.

(**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:application:type` exception is raised.

(this-expression-source-directory) SYNTAX

Expands to a string that names 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.

`true` BOOLEAN

Boolean true.

15. file.ss: Filesystem Utilities

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:i/o: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:i/o: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 (except for letter case; see `normalize-path` below).

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

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

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

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

`(find-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 §7.4.1.1 in *PLT MzScheme: Language Manual*).

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.

`(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 (except for letter case; see `normalize-path` 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 §7.4.1.1 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 *predicate* 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*]) 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 string, the temporary file is created as a copy of the named file,. If *copy-from-filename* is #f or not provided, the temporary file is created as empty.

The temporary file is not opened for reading or writing when the pathname is returned. The client program calling `make-temporary-file` is expected to open the file with the desired access and flags (probably using the 'truncate flag; see §11.1.2 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). The default *format-string* is `"mztmp~a"`.

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

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

Letter case is *not* normalized by `normalize-path`, so combine `normalize-path` with `normal-case-path` to

get strings for path comparison.

An error is signaled by `normalize-path` if the input path contains an embedded path for a non-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 *fielname* 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*.

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 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*.

16. include.ss: Textually Including Source

(include *path-spec***)** SYNTAX

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

The *path-spec* can be either a literal string (parsed according to the platform's conventions) or 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*).

If *path-spec* specifies a relative path, it 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.

17. inflate.ss: Inflating Compressed Data

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

Extracts data that was compressed using the GNU `gzip` utility (or `gzip` in the **deflate.ss** library; see §12), writing the uncompressed data directly to a file. The *file* argument is the name of the file containing compressed data. The default output file name is the original name of the compressed file as stored in *file*. If a file by this name exists, it will be overwritten. If no original name is stored in the source file, `"unzipped"` is used as the default output file name.

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

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

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

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

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

(inflate *in out*) PROCEDURE

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

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

18. list.ss: List Utilities

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.

`(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 `foldr`, `foldl` 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.

`(mergesort` *list less-than?*`)` PROCEDURE

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

`(quicksort` *list less-than?*`)` PROCEDURE

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

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

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

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

Like `remove`, except that the first argument is a list of items to remove, instead of a single item.

`(remq` *item list*`)` PROCEDURE

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

`(remq*` *items list*`)` PROCEDURE

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

`(remv` *item list*`)` PROCEDURE

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

`(remv*` *items list*`)` PROCEDURE

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

`(rest` *l*`)` PROCEDURE

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

`(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!`.)

19. match.ss: Pattern Matching

This library provides functions for pattern-matching Scheme values. (This chapter written by Andrew K. Wright, originally titled *Pattern Matching for Scheme*.) 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])

Figures 19.1 and 19.2 give 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 schemematch-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 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.

		Pattern :	*Matches* :
pat	::=	*identifier*	anything, and binds *identifier* as a variable
	|	_	anything
	|	()	itself (the empty list)
	|	#t	itself
	|	#f	itself
	|	*string*	an `equal?` string
	|	*number*	an `equal?` number
	|	*character*	an `equal?` character
	|	'*s-expression*	an `equal?` s-expression
	|	'*symbol*	an `equal?` symbol (special case of s-expression)
	|	$(pat_1 \ldots pat_n)$	a proper list of n elements
	|	$(pat_1 \ldots pat_n \ . \ pat_{n+1})$	a list of n or more elements
	|	$(pat_1 \ldots pat_n \ pat_{n+1} \ ...)$	a proper list of n or more elements[1]
	|	$(pat_1 \ldots pat_n \ pat_{n+1} \ ..k)$	a proper list of $n+k$ or more elements
	|	$\#(pat_1 \ldots pat_n)$	a vector of n elements
	|	#&*pat*	a box
	|	($ *struct* $pat_1 \ldots pat_n$)	a structure
	|	(**and** $pat_1 \ldots pat_n$)	if all of pat_1 through pat_n match
	|	(**or** $pat_1 \ldots pat_n$)	if any of pat_1 through pat_n match
	|	(**not** $pat_1 \ldots pat_n$)	if none of pat_1 through pat_n match
	|	(? *predicate* $pat_1 \ldots pat_n$)	if *predicate* true and pat_1 through pat_n all match
	|	(**set!** *identifier*)	anything, and binds *identifier* as a setter
	|	(**get!** *identifier*)	anything, and binds *identifier* as a getter
	|	‘*qp*	a quasipattern

Figure 19.1: Pattern Syntax

		Quasipattern:	*Matches* :
qp	::=	()	itself (the empty list)
	|	#t	itself
	|	#f	itself
	|	*string*	an `equal?` string
	|	*number*	an `equal?` number
	|	*character*	an `equal?` character
	|	*identifier*	an `equal?` symbol
	|	$(qp_1 \ldots qp_n)$	a proper list of n elements
	|	$(qp_1 \ldots qp_n \ . \ qp_{n+1})$	a list of n or more elements
	|	$(qp_1 \ldots qp_n \ qp_{n+1} \ ...)$	a proper list of n or more elements
	|	$(qp_1 \ldots qp_n \ qp_{n+1} \ ..k)$	a proper list of $n+k$ or more elements
	|	$\#(qp_1 \ldots qp_n)$	a vector of n elements
	|	#&*qp*	a box
	|	,*pat*	a pattern
	|	,@*pat*	a pattern, spliced

Figure 19.2: Quasipattern Syntax

19.1 Patterns

Figure 19.1 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*$_n$.
- (*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.
- (*pat*$_1$ ··· *pat*$_n$ *pat*$_{n+1}$ **...**) — matches a proper list of *n* or more elements, where each element of the tail 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.
- (*pat*$_1$ ··· *pat*$_n$ *pat*$_{n+1}$ **..***k*) — similar to the previous pattern, but the tail must be at least *k* elements long. The pattern keywords **..0** and **...** are equivalent.
- #(*pat*$_1$ ··· *pat*$_n$) — matches a vector of length *n*, whose elements match *pat*$_1$ through *pat*$_n$.
- #&*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 §12.6.3 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 §35.2) can provide the structure type information.
- (**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 scmkset!, 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, the reult is void.

19.2 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.

20. math.ss: Math

`(conjugate` z`)` PROCEDURE

Returns the complex conjugate of z.

`(cosh` z`)` PROCEDURE

Returns the hyperbolic cosine of z.

`e` NUMBER

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

`pi` NUMBER

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

`(sinh` z`)` PROCEDURE

Returns the hyperbolic sine of z.

`(sgn` n`)` PROCEDURE

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

`(sqr` z`)` PROCEDURE

Returns $(* \ z \ z)$).

21. pconvert.ss: Converted Printing

This library defines routines for printing Scheme values as `eval`uatable S-expressions rather than `read`able S-expressions. The `print-convert` procedure does not print values; rather, it converts a Scheme value into another Scheme value such that the new value pretty-prints as a Scheme expression that evaluates to the original value. For example, `(pretty-print (print-convert '(9 ,(box 5) #(6 7)))` prints the literal expression `(list 9 (box 5) (vector 6 7))` to the current output port.

To install print converting into the `read-eval-print` loop, require **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).

`(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

This parameter that controls how values that have inferred names are represented. This parameter is passed a value. If the parameter returns `#t`, the *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).

`(use-named/undefined-handler` [*use-handler*]`)` PROCEDURE

This parameter that controls how values that have inferred names are represented. This parameter is only called if *use-named/undefined-handler* returned true for some value. This parameter 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`.

`(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 *shared-info* value must be a result from `build-share`. The procedure returns a list matching variables to shared values within the value passed to `build-share`. For example,

```
(get-shared (build-share (shared ([a (cons 1 b)][b (cons 2 a)]) a)))
```

might return the list

```
((-1- (cons 1 -2-)) (-2- (cons 2 -1-)))
```

The default value for *cycles-only?* is #f; if it is not #f, `get-shared` returns only information about cycles.

`(install-converting-printer)` PROCEDURE

Sets the current print handler to print values using `print-convert`. The current read handler is also set to use the prompt returned by `current-read-eval-convert-print-prompt`.

`(print-convert` *v* [*cycles-only?*]`)` PROCEDURE

Converts the value *v*. If *cycles-only?* is not #f, then only circular objects are included in the output. The default value of *cycles-only?* is the value of `(show-sharing)`.

`(print-convert-expr` *share-info v unroll-once?*`)` PROCEDURE

Converts the value *v* using sharing information *share-info* previously returned by `build-share` for a value containing *v*. If the most recent call to `get-shared` with *share-info* requested information only for cycles, then `print-convert-expr` will only display sharing among values for cycles, rather than showing all value sharing.

The *unroll-once?* argument is used if *v* is a shared value in *share-info*. In this case, if *unroll-once?* is #f, then the return value will be a shared-value identifier; otherwise, the returned value shows the internal structure of *v* (using shared value identifiers within *v*'s immediate structure as appropriate).

`(quasi-read-style-printing` [*on?*]`)` PROCEDURE

Parameter that controls how vectors and boxes are represented after conversion when the value of `constructor-style-printing` is #f. If `quasi-read-style-printing` is set to #f, then boxes and vectors are unquoted and represented using 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.

22. pregexp.ss: Perl-Style Regular Expressions

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.

22.1 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.

22.2 Regexp procedures

This library provides the procedures `pregexp`, `pregexp-match-positions`, `pregexp-match`, `pregexp-split`, `pregexp-replace`, and `pregexp-replace*`.

[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 '`"`'.

22.2.1 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`.

22.2.2 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.

22.2.3 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.

22.2.4 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")
```

22.2.5 pregexp-replace

(pregexp-replace *regexp text-string 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 the *insert string* (string to be inserted).

```
(pregexp-replace "te" "liberte" "ty")
=> "liberty"
```

If the pattern doesn't occur in the text string, the returned string is identical (eq?) to the text string.

22.2.6 pregexp-replace*

(pregexp-replace* *regexp text-string insert-string*) PROCEDURE

Replaces *all* matches in the text string by the insert string:

```
(pregexp-replace* "te" "liberte egalite fraternite" "ty")
=> "liberty egality fratyrnity"
```

As with pregexp-replace, if the pattern doesn't occur in the text string, the returned string is identical (eq?) to the text string.

22.2.7 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.

22.3 The regexp pattern language

Here is a complete description of the regexp pattern language recognized by the pregexp procedures.

22.3.1 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.

22.3.2 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 `pffft`.

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`.

22.3.2.1 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 be part of a "word".[3]

[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.

[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".

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.

22.3.2.2 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")

(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.

22.3.3 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))
```

22.3.3.1 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")
```

22.3.3.2 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 `?`.

22.3.4 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")
```

22.3.4.1 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 nth 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"
  "12334098324243242098098O234"
  "{\\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.

22.3.4.2 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")
```

22.3.4.3 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")
```

[5] A useful, if terminally cute, coinage from the abbots of Perl .

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.

```
(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.

22.3.5 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")
```

22.3.6 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.

22.3.6.1 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`.

22.3.7 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.

22.3.7.1 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.

22.3.7.2 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.

22.4 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.

[6]Note that *n0-255* lists prefixes as preferred alternates, something we cautioned against in sec 22.3.5. However, since we intend to anchor this subregexp explicitly to force an overall match, the order of the alternates does not matter.

```
(define ip-re1
  (string-append
    "^"        ;nothing before
    n0-255     ;the first n0-255,
    "(?x:"     ;then the subpattern of
    "\\."      ;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
```

23. pretty.ss: Pretty Printing

(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.

(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 sets the default depth for recursive pretty printing to *depth* and returns void. If no *depth* argument is provided, the current value is returned instead. A depth of 0 indicates that only simple values are printed; Scheme values within other values (e.g. the elements of a list) are replaced with "...".

(pretty-print-display-string-handler [*f*]) PROCEDURE

Parameter that sets the procedure for displaying final strings to a port to output pretty-printed values. The default handler is the default port display handler (see §11.2.5 in *PLT MzScheme: Language Manual*).

(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 require require-for-syntax provide link
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-print-hook` [*hook*]`)` PROCEDURE

Parameter that sets the print hook for pretty-printing to *hook*. If *hook* is not provided, the current hook is returned.

The print hook is applied to a value for printing when the sizing hook (see `pretty-print-size-hook`) returns an integer size for the value.

The print hook receives three arguments. The first argument is the value to print. The second argument is a Boolean: #t for printing like `display` and #f for printing like `write`. The third argument is the destination port.

`(pretty-print-print-line` [*liner*]`)` PROCEDURE

Parameter that sets a procedure for printing the newline separator between lines of a pretty-printed value. The *liner* procedure is called with four arguments: a new line number, an output port, the old line's length,

and the number of destination columns. The return value from *liner* is the number of extra characters it printed at the beginning of the new line.

The *liner* procedure is called before any characters are printed with `0` as the line number and `0` as the old line length; *liner* is called after the last character for a value is printed with #f as the line number and with the length of the last line. Whenever the pretty-printer starts a new line, *liner* is called with the new line's number (where the first new line is numbered `1`) and the just-finished line's length. The destination columns argument to *liner* is always the total width of the destination printing area, or 'infinity if pretty-printed values are not broken into lines.

The default *liner* procedure prints a newline whenever the line number is not `0` and the column count is not 'infinity, always returning `0`. A custom *liner* procedure can be used to print extra text before each line of pretty-printed output; the number of characters printed before each line should be returned by *liner* so that the next line break can be chosen correctly.

`(pretty-print-show-inexactness` [*explicit?*]`)` PROCEDURE

Parameter that determines how inexact numbers are printed. If the parameter's value is #t, then inexact numbers are always printed with a leading `#i`. The initial value is #f.

`(pretty-print-style-table?` *v*`)` PROCEDURE

Returns #t if *v* is a style table,#f otherwise.

`(pretty-print-post-print-hook` [*hook*]`)` PROCEDURE

Parameter that sets a procedure to be called just after an object is printed. The hook receives two arguments: the object and the output port.

`(pretty-print-pre-print-hook` [*hook*]`)` PROCEDURE

Parameter that sets a procedure to be called just before an object is printed. The hook receives two arguments: the object and the output port.

`(pretty-print-size-hook` [*hook*]`)` PROCEDURE

Parameter that sets the sizing hook for pretty-printing to *hook*. If *hook* is not provided, the current hook is returned.

The sizing hook is applied to each value to be printed. If the hook returns #f, then printing is handled internally by the pretty-printer. Otherwise, the value should be an integer specifying the length of the printed value in characters; the print hook will be called to actually print the value (see `pretty-print-print-hook`).

The sizing hook receives three arguments. The first argument is the value to print. The second argument is a Boolean: #t for printing like `display` and #f for printing like `write`. The third argument is the destination port. The sizing hook may be applied to a single value 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 surronding it.

24. process.ss: Process and Shell-Command Execution

This library builds on MzScheme's `subprocess` procedure; see also §15.2 in *PLT MzScheme: Language Manual*.

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

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

(`process` *command-string*`)` executes a shell command asynchronously under Unix, Windows, and BeOS. (This procedure is not supported for MacOS.) 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.
  - '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` under Unix for all of Unix, Windows, and BeOS, except that *command-string* is a filename that is executed directly, and the *arg-string*s are the arguments. (This procedure is not supported for MacOS.) 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. All provided ports must be file-stream ports. Any of the ports can be #f, in which case a system pipe is created and returned, 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`.

25. restart.ss: Simulating Stand-alone MzScheme

(`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 §7). 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.)

26. sendevent.ss: AppleEvents

26.1 AppleEvents

(`send-event` *receiver-string event-class-string event-id-string* [*direct-argument-v argument-list*]) PROCEDURE

Sends an AppleEvent or raises `exn:misc:unsupported`. Currently AppleEvents are supported only within MrEd under Mac OS Classic and Mac OS X.

The *receiver-string*, *event-class-string*, and *event-id-string* arguments specify the signature of the receiving application, the class of the AppleEvent, and the ID of the AppleEvent. Each of these must be a four-character string, otherwise the `exn:application:type` exception is raised.

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

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

#t or #f	⇒ Boolean
small integer	⇒ Long Integer
inexact real number	⇒ Double
string	⇒ Characters
list of 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:misc:unsupported` exception is raised.

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

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

Boolean ⇒ #t or #f
Signed Integer ⇒ integer
Float, Double, or Extended ⇒ inexact real number
Characters ⇒ string
list of reverse-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:misc` exception is raised.

27. shared.ss: Graph Constructor Syntax

(**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)
```

28. spidey.ss: MrSpidey Annotations

This library defines syntax for annotations that used to be understood by MrSpidey. The annotations are associated to syntax objects via properties (see §12.6.2 in *PLT MzScheme: Language Manual*), and the syntax forms below otherwise expand away. The following macros are defined:

- `:` — expands to the first expression
- `polymorphic` — expands to the first expression
- `define-constructor` — expands to `(void)`
- `define-type` — expands to `(void)`
- `mrspidey:control` — expands to `(void)`
- `type:` — expands to `(void)`

29. string.ss: String Utilities

`(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 str* [*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 *str*. (Unlike `regexp-match`, results for parenthesized subpatterns in *pattern* are not returned.) If *pattern* matches a zero-length string along the way, the `exn:user` exception is raised.

If *str* contains no matches (in the range *start* to *end*), `null` is returned. Otherwise, each string in the resulting list is a distinct substring of *str* that matches *pattern*.

`(regexp-match-exact?` *pattern str*`)` 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 string *str* matches *pattern*.

`(regexp-match-positions*` *pattern str* [*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 *str*. (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:user` exception is raised.

If *str* contains no matches, `null` is returned.

`(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 in *str* is protected by a quoting backslash.

`(regexp-split` *pattern str* [*start-k end-k*]`)` PROCEDURE

The complement of `regexp-match*` (see above): the result is a list of sub-strings in *str* that are separated by matches to *pattern*; adjacent matches are separated with `""`. If *pattern* matches a zero-length string along the way, the `exn:user` exception is raised.

If *str* contains no matches (in the range *start* to *end*), the result will be a list containing *str* (from *start* to *end*). If a match occurs at the beginning of *str* (at *start*), the resulting list will start with an empty string, and if a match occurs at the end (at *end*), the list will end with an empty string.

`(string-lowercase!` *str*`)` PROCEDURE

Destructively changes *str* to contain only lowercase characters.

`(string-uppercase!` *str*`)` PROCEDURE

Destructively changes *str* to contain only uppercase characters.

30. thread.ss: Thread Utilities

`(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 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.

`(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 without significant buffer delays (i.e., a character that becomes available on *input-port* is immediately transferred to *output-port*, even if future reads on *input-port* must block).

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.

`(dynamic-disable-break` *thunk*`)` PROCEDURE

Invokes *thunk* and returns the result. During the application of *thunk*, breaks are disabled.

`(dynamic-enable-break` *thunk*`)` PROCEDURE

Invokes *thunk* and returns the result. During the application of *thunk*, breaks are enabled.

`(make-single-threader)` PROCEDURE

Returns a new procedure that takes any thunk and applies it. When this procedure is applied to any collection of thunks by any collection of threads, the thunks are applied sequentially across all threads.

[1] The returned procedure actually accepts any number of arguments, but immediately raises `exn:application:arity` if *f* cannot accept the provided number of arguments.

(`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 characters 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.

(`run-server` *port-k session-proc session-timeout*) PROCEDURE

Executes a TCP server on the port inidicated by *port-k*. When a connection is made by a client, *session-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, or *session*, is managed by a new custodian, and each call to *session-proc* occurs in a new thread (managed by the session's custodian). If the thread executing *session-proc* terminates for any reason (e.g., *sesson-proc* returns), the session's custodian is shut down. Consequently, *session-proc* need not close the ports provded to it. Breaks are enabled in the session thread if breaks are enabled when *run-server* is called.

If *session-timeout* is not #f, then it must be a non-negative number specifying the time in seconds that a session thread is allowed to run before it is sent a break signal. Then, if the thread runs longer than (∗ *session-timeout* 2) seconds, then the session's custodian is shut down. If *session-timeout* is #f, a session thread can run indefinitely.

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 sessions.

(`with-semaphore` *s thunk*) PROCEDURE

Calls `semaphore-wait` on *s*, then invokes *thunk* with no arguments, and then calls `semaphore-post` on *s*. The return value is the result of calling *thunk*.

31. trace.ss: Tracing Top-level Procedure Calls

This library mimics the tracing facility available in Chez Scheme™.

Tracing does not respect tail calls; i.e., tracing a procedure that ends with a tail call checks the call so that it executes (and prints) as a non-tail call. Untracing a procedure restores its tail call behavior. Only one procedure can be traced for any single name across all namespaces.

`(trace` *name* ···`)` SYNTAX

This form takes a sequence of global variables names; each name must be defined as as procedure in the current namespace when the `trace` expression is evaluated. Each *name* provided to `trace` is then redefined to a new procedure. This new procedure traces procedure-calls and procedure-returns by printing the arguments and results of the call. If multiple values are returned, each value is displayed starting on a separate line.

When traced procedures invoke each other, this is shown by printing a nesting prefix. If the nesting depth grows to ten and beyond, a number is printed to show the actual nesting depth.

The `trace` macro can be used on a name that is already traced in the current namespace. In this case, assuming that the name has not been redefined, `trace` has no effect. If the name *has* been redefined, then a new trace is installed. If `trace` is used on the same name in two different eventspaces, then the first installed trace will remain intact but it will no longer be recognized by the `trace` and `untrace` forms.

The value of a `trace` expression is the list of names specified for tracing.

`(untrace` *name* ···`)` SYNTAX

This form undoes the effects of the `trace` form for each *name*, but only if the current definition of *name* is the one previously installed by `trace`. If the current definition for *name* is not the procedure installed by `trace`, then the definition is not changed.

The value of an `untrace` expression is the list of names restored to their untraced definitions.

32. traceld.ss: Tracing File Loads

This library does not define any procedures or syntax. Instead, **trace.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.

33. transcr.ss: Transcripts

MzScheme's built-in `transcript-on` and `transcript-off` always raise `exn:misc:unsupported`. The **transcr.ss** library provides working versions of `transcript-on` and `transcript-off`.

34. unit.ss: Core Units

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

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 plugable 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 §35, and defined by **unitsig.ss**. Details about mixing core and signed units are presented in §35.9 (using procedures from **unitsig.ss**).

34.1 Creating Units

The **unit** form creates a unit:

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

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

The *variable*s in the **import** clause are bound within the *unit-body-expr* expressions. The variables for

exportages in the **export** clause must be defined in the *unit-body-expr*s as described below; additional private variables can be defined as well. The imported 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 an platform-specific graphics toolbox.

34.2 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 §34.1.

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

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

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

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

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

34.3 Linking Units and Creating Compound Units

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

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

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

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

tag is
  identifier

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

- The **import** clause imports variables into the compound unit. These imported variables are used as imports to the compound unit's sub-units.

- The **link** clause specifies how the compound unit is created from sub-units. A unique *tag* is associated with each sub-unit, which is specified using an arbitrary expression. Following the unit expression, each *linkage* specifies a variable using the *variable* form or the (*tag variable*) form. In the former case, the *variable* must occur in the **import** clause of the **compound-unit** expression; in the latter case, the *tag* must be defined in the same **compound-unit** expression. The (*tag variable* ···) form is a shorthand for multiple adjacent clauses of the second form with the same *tag*.

- The **export** clause re-exports variables from the compound unit that were originally exported from the sub-units. The *tag* part of each **export** sub-clause specifies the sub-unit from which the re-exported variable is drawn. The *exportage*s specify the names of variables exported by the sub-unit to be re-exported.

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

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

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

Examples

These examples use the definitions from the earlier unit examples in §34.1.

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.

34.4 Unit Utilities

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

35. unitsig.ss: Units with Signatures

The unit syntax presented in §34 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 §34.3: 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.

35.1 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 §35.2. The full **unit/sig** syntax is described in §35.3.

35.2 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 §12.6.3 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 strcuture 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 §35.6 and §35.7.)

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.

35.3 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 §35.2:

- 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.

35.4 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 §35.6.

35.5 Restricting Signatures

As explained in §35.4, 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 §35.4, 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.

35.6 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 §35.7), 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 §35.5, 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*).

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

35.8 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 §35.2; 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 §35.2 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*.

35.9 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 §35.2 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.

35.10 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.

35.11 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:unit` exception is raised for one of the following reasons:

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

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

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

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

  - A value in the *units* list is not a signed unit.

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

(signature->symbols *name***)** SYNTAX

Expands to the "exploded" version (see §35.11) of the signature bound to *name* (where *name* is an unevaluated identifier).

Index