Guide: PLT Scheme

Version 3.99.0.23

April 30, 2008

This guide is intended for programmers who are new to Scheme, new to PLT Scheme, or new to some part of PLT Scheme. It assumes programming experience, so if you are new to programming, consider instead reading *How to Design Programs*. If you want a brief introduction to PLT Scheme, start with §"**Quick**: An Introduction to PLT Scheme with Pictures".

Chapter 2 provides a brief introduction to Scheme. From Chapter 3 on, this guide dives into details—covering much of the PLT Scheme toolbox, but leaving precise details to §"**Reference**: PLT Scheme" and other reference manuals.

Contents

1 Welcome to PLT Scheme

Depending on how you look at it, **PLT Scheme** is

- a *programming language*—a descendant of Scheme, which is a dialect of Lisp;
- a *family* of programming languages—variants of Scheme, and more; or
- a set of *tools*—for using a family of programming languages.

Where there is no room for confusion, we use simply *Scheme* to refer to any of these facets of PLT Scheme.

PLT Scheme's two main tools are

- **MzScheme**, the core compiler, interpreter, and run-time system; and
- **DrScheme**, the programming environment (which runs on top of MzScheme).

Most likely, you'll want to explore PLT Scheme using DrScheme, especially at the beginning. If you prefer, you can also work with the command-line `mzscheme` interpreter and your favorite text editor. The rest of this guide presents the language mostly independent of your choice of editor.

If you're using DrScheme, you'll need to choose the proper language, because DrScheme accommodates many different variants of Scheme. Assuming that you've never used DrScheme before, start it up, type the line

```
#lang scheme
```

in DrScheme's top text area, and then click the Run button that's above the text area. DrScheme then understands that you mean to work in the normal variant of Scheme (as opposed to the smaller `scheme/base`, or many other possibilities).

If you've used DrScheme before with something other than a program that starts `#lang`, DrScheme will remember the last language that you used, instead of inferring the language from the `#lang` line. In that case, use the Language|Choose Language... menu item. In the the dialog that appears, select the first item, which is Module. Put the `#lang` line above in the top text area, still.

1.1 Interacting with Scheme

DrScheme's bottom text area and the `mzscheme` command-line program (when started with no options) both act as a kind of calculator. You type a Scheme expression, hit return, and

the answer is printed. In the terminology of Scheme, this kind of calculator is called a *read-eval-print loop* or *REPL*.

A number by itself is an expression, and the answer is just the number:

```
> 5
5
```

A string is also an expression that evaluates to itself. A string is written with double quotes at the start and end of the string:

```
> "hello world"
"hello world"
```

Scheme uses parentheses to wrap larger expressions—almost any kind of expression, other than simple constants. For example, a function call is written: open parenthesis, function name, argument expression, and closing parenthesis. The following expression calls the built-in function `substring` with the arguments `"hello world"`, `0`, and `5`:

```
> (substring "hello world" 0 5)
"hello"
```

1.2 Definitions and Interactions

You can define your own functions that work like `substring` by using the `define` form, like this:

```
(define (piece str)
  (substring str 0 5))

> (piece "howdy universe")
"howdy"
```

Although you can evaluate the `define` form in the REPL, definitions are normally a part of a program that you want to keep and use later. So, in DrScheme, you'd normally put the definition in the top text area—called the *definitions area*—along with the `#lang` prefix:

```
#lang scheme

(define (piece str)
  (substring str 0 5))
```

If calling `(piece "howdy universe")` is part of the main action of your program, that would go in the definitions area, too. But if it was just an example expression that you were using to explore `piece`, then you'd more likely leave the definitions area as above, click Run, and then evaluate `(piece "howdy universe")` in the REPL.

With `mzscheme`, you'd save the above text in a file using your favorite editor. If you save it as `"piece.ss"`, then after starting `mzscheme` in the same directory, you'd evaluate the following sequence:

```
> (enter! "piece.ss")
> (piece "howdy universe")
"howdy"
```

The `enter!` function both loads the code and switches the evaluation context to the inside of the module, just like DrScheme's Run button.

1.3 A Note to Readers with Scheme/Lisp Experience

If you already know something about Scheme or Lisp, you might be tempted to put just

```
(define (piece str)
  (substring str 0 5))
```

into `"piece.ss"` and run `mzscheme` with

```
> (load "piece.ss")
> (piece "howdy universe")
"howdy"
```

That will work, because `mzscheme` is willing to imitate a traditional Scheme environment, but we strongly recommend against using `load` or writing programs outside of a module.

Writing definitions outside of a module leads to bad error messages, bad performance, and awkward scripting to combine and run programs. The problems are not specific to `mzscheme`; they're fundamental limitations of the traditional top-level environment, which Scheme and Lisp implementations have historically fought with ad hoc command-line flags, compiler directives, and build tools. The module system is to designed to avoid the problems, so start with `#lang`, and you'll be happier with PLT Scheme in the long run.

2 Scheme Essentials

This chapter provides a quick introduction to Scheme as background for the rest of the guide. Readers with some Scheme experience can safely skip to §3 “Built-In Datatypes”.

2.1 Simple Values

Scheme values include numbers, booleans, strings, and byte strings. In DrScheme and documentation examples (when you read the documentation in color), value expressions are shown in green.

Numbers are written in the usual way, including fractions and imaginary numbers:

§3.2 “Numbers” (later in this guide) explains more about numbers.

```
1                        1.0
1/2                      0.5
99999999999999999999     1e+22
1+2i                     1.0+2.0i
```

Booleans are `#t` for true and `#f` for false. In conditionals, however, all non-`#f` values are treated as true.

§3.1 “Booleans” (later in this guide) explains more about booleans.

Strings are written between doublequotes. Within a string, backslash is an escaping character; for example, a backslash followed by a doublequote includes a literal doublequote in the string. Except for an unescaped doublequote or backslash, any Unicode character can appear in a string constant.

§3.4 “Strings (Unicode)” (later in this guide) explains more about strings.

```
"hello world"
"A \"fancy\" string"
"λx:(μα.αα).xx"
```

When a constant is evaluated in the REPL, it typically prints the same as its input syntax. In some cases, the printed form is a normalized version of the input syntax. In documentation and in DrScheme’s REPL, results are printed in blue instead of green to highlight the difference between an input expression and a printed result.

Examples:

```
> 1.0000
1.0
> "A \u0022fancy\u0022 string"
"A \"fancy\" string"
```

2.2 Simple Definitions and Expressions

A program module is written as

`#lang` ⟨*langname*⟩ ⟨*topform*⟩*

where a ⟨*topform*⟩ is either a ⟨*definition*⟩ or an ⟨*expr*⟩. The REPL also evaluates ⟨*topform*⟩s.

In syntax specifications, text with a gray background, such as `#lang`, represents literal text. Whitespace must appear between such literals and nonterminals like ⟨*id*⟩, except that whitespace is not required before or after `(`, `)`, `[`, or `]`. A comment, which starts with `;` and runs until the end of the line, is treated the same as whitespace.

Following the usual conventions, * in a grammar means zero or more repetitions of the preceding element, $^+$ means one or more repetitions of the preceding element, and {} groups a sequence as an element for repetition.

2.2.1 Definitions

A definition of the form

`(` `define` ⟨*id*⟩ ⟨*expr*⟩ `)`

binds ⟨*id*⟩ to the result of ⟨*expr*⟩, while

`(` `define` `(` ⟨*id*⟩ ⟨*id*⟩* `)` ⟨*expr*⟩$^+$ `)`

binds the first ⟨*id*⟩ to a function (also called a *procedure*) that takes arguments as named by the remaining ⟨*id*⟩s. In the function case, the ⟨*expr*⟩s are the body of the function. When the function is called, it returns the result of the last ⟨*expr*⟩.

§4.5 "Definitions: `define`" (later in this guide) explains more about definitions.

Examples:

```
(define five 5)             ; defines five to be 5

(define (piece str)         ; defines piece as a function
  (substring str 0 five))   ; of one argument

> five
5
> (piece "hello world")
"hello"
```

Under the hood, a function definition is really the same as a non-function definition, and a function name does not have to be used in a function call. A function is just another kind of value, though the printed form is necessarily less complete than the printed form of a number or string.

Examples:

```
> piece
#<procedure:piece>
> substring
#<procedure:substring>
```

A function definition can include multiple expressions for the function's body. In that case, only the value of the last expression is returned when the function is called. The other expressions are evaluated only for some side-effect, such as printing.

Examples:

```
(define (greet name)
  (printf "returning a greeting for ~a...\n" name)
  (string-append "hello " name))

> (greet "universe")
returning a greeting for universe...
"hello universe"
```

Scheme programmers prefer to avoid assignment statements. It's important, though, to understand that multiple expressions are allowed in a definition body, because it explains why the following `nogreet` function simply returns its argument:

```
(define (nogreet name)
  string-append "hello " name)

> (nogreet "world")
"world"
```

Within `nogreet`, there are no parentheses around `string-append "hello " name`, so they are three separate expressions instead of one function-call expression. The expressions `string-append` and `"hello "` are evaluated, but the results are never used. Instead, the result of the function is just the result of the expression `name`.

2.2.2 An Aside on Indenting Code

Line breaks and indentation are not significant for parsing Scheme programs, but most Scheme programmers use a standard set of conventions to make code more readable. For example, the body of a definition is typically indented under the first line of the definition. Identifiers are written immediately after an open parenthesis with no extra space, and closing parentheses never go on their own line.

DrScheme automatically indents according to the standard style when you type Enter in a program or REPL expression. For example, if you hit Enter after typing `(define (greet name)`, then DrScheme automatically inserts two spaces for the next line.

If you change a region of code, you can select it in DrScheme and hit Tab, and DrScheme will re-indent the code (without inserting any line breaks). Editors like Emacs offer a Scheme mode with similar indentation support.

Re-indenting not only makes the code easier to read, it gives you extra feedback that your parentheses are matched in the way that you intended. For example, if you leave out a closing parenthesis after the last argument to a function, automatic indentation starts the next line under the first argument, instead of under the `define` keyword:

```
(define (nogreet name
                (string-append "hello " name)))
```

Furthermore, when an open parenthesis has no matching close parenthesis in a program, both `mzscheme` and DrScheme use the source's indentation to suggest where it might be missing.

2.2.3 Identifiers

Scheme's syntax for identifiers is especially liberal. Excluding the special characters

```
( ) [ ] { } " , ' ` ; # | \
```

and except for the sequences of characters that make number constants, almost any sequence of non-whitespace characters forms an ⟨*id*⟩. For example `substring` is an identifier. Also, `string-append` and `a+b` are identifiers, as opposed to arithmetic expressions. Here are several more examples:

§4.2 "Identifiers and Binding" (later in this guide) explains more about identifiers.

```
+
Apple
integer?
call/cc
call-with-composable-continuation
x-1+3i
```

2.2.4 Function Calls (Procedure Applications)

We have already seen many function calls, which are called *procedure applications* in more traditional Scheme terminology. The syntax of a function call is

```
( ⟨id⟩ ⟨expr⟩* )
```

where the number of ⟨*expr*⟩s determines the number of arguments supplied to the function named by ⟨*id*⟩.

§4.3 "Function Calls" (later in this guide) explains more about function calls.

The `scheme` language pre-defines many function identifiers, such as `substring` and

string-append. More examples are below.

In example Scheme code throughout the documentation, uses of pre-defined names are hyperlinked to the reference manual. So, you can click on an identifier to get full details about its use.

```
> (string-append "hello"  " "  "scheme") ; append strings
"hello scheme"
> (substring "hello scheme" 6 12)        ; extract a substring
"scheme"
> (string-length "scheme")               ; get a string's length
6
> (string? "hello scheme")               ; recognize strings
#t
> (string? 1)
#f
> (sqrt 16)                              ; find a square root
4
> (sqrt -16)
0+4i
> (+ 1 2)                                ; add numbers
3
> (- 2 1)                                ; subtract numbers
1
> (< 2 1)                                ; compare numbers
#f
> (>= 2 1)
#t
> (number? "hello scheme")               ; recognize numbers
#f
> (number? 1)
#t
> (equal? 1 "hello")                     ; compare anything
#f
> (equal? 1 1)
#t
```

2.2.5 Conditionals with `if`, `and`, `or`, and `cond`

The next simplest kind of expression is an `if` conditional:

`(` `if` ⟨*expr*⟩ ⟨*expr*⟩ ⟨*expr*⟩ `)`

§4.7 "Conditionals" (later in this guide) explains more about conditionals.

The first ⟨*expr*⟩ is always evaluated. If it produces a non-`#f` value, then the second ⟨*expr*⟩ is evaluated for the result of the whole `if` expression, otherwise the third ⟨*expr*⟩ is evaluated

for the result.

Examples:

```
> (if (> 2 3)
      "bigger"
      "smaller")
"smaller"

(define (reply s)
  (if (equal? "hello" (substring s 0 5))
      "hi!"
      "huh?"))

> (reply "hello scheme")
"hi!"
> (reply "λx:(μα.αα).xx")
"huh?"
```

Complex conditionals can be formed by nesting `if` expressions. For example, you could make the `reply` function work when given non-strings:

```
(define (reply s)
  (if (string? s)
      (if (equal? "hello" (substring s 0 5)
          "hi!"
          "huh?"))
      "huh?"))
```

Instead of duplicating the `"huh?"` case, this function is better written as

```
(define (reply s)
  (if (if (string? s)
          (equal? "hello" (substring s 0 5)
          #f))
      "hi!"
      "huh?"))
```

but these kinds of nested `if`s are difficult to read. Scheme provides more readable shortcuts through the `and` and `or` forms, which work with any number of expressions:

§4.7.2 "Combining Tests: `and` and `or`" (later in this guide) explains more about `and` and `or`.

```
( and ⟨expr⟩* )
( or ⟨expr⟩* )
```

The `and` form short-circuits: it stops and returns `#f` when an expression produces `#f`, otherwise it keeps going. The `or` form similarly short-circuits when it encounters a true result.

Examples:

```
(define (reply s)
```

```
  (if (and (string? s)
           (>= (string-length s) 5)
           (equal? "hello" (substring s 0 5)))
      "hi!"
      "huh?"))

> (reply "hello scheme")
"hi!"
> (reply 17)
"huh?"
```

Another common pattern of nested `if`s involves a sequence of tests, each with its own result:

```
(define (reply-more s)
  (if (equal? "hello" (substring s 0 5))
      "hi!"
      (if (equal? "goodbye" (substring s 0 7))
          "bye!"
          (if (equal? "?" (substring s (- (string-length s) 1)))
              "I don't know"
              "huh?"))))
```

The shorthand for a sequence of tests is the `cond` form:

```
( cond {[ ⟨expr⟩ ⟨expr⟩* ]}* )
```

§4.7.3 "Chaining Tests: `cond`" (later in this guide) explains more about `cond`.

A `cond` form contains a sequence of clauses between square brackets. In each clause, the first ⟨*expr*⟩ is a test expression. If it produces true, then the clause's remaining ⟨*expr*⟩s are evaluated, and the last one in the clause provides the answer for the entire `cond` expression; the rest of the clauses are ignored. If the test ⟨*expr*⟩ produces `#f`, then the clause's remaining ⟨*expr*⟩s are ignored, and evaluation continues with the next clause. The last clause can use `else` as a synonym for a `#t` test expression.

Using `cond`, the `reply-more` function can be more clearly written as follows:

```
(define (reply-more s)
  (cond
   [(equal? "hello" (substring s 0 5))
    "hi!"]
   [(equal? "goodbye" (substring s 0 7))
    "bye!"]
   [(equal? "?" (substring s (- (string-length s) 1)))
    "I don't know"]
   [else "huh?"]))

> (reply-more "hello scheme")
"hi!"
```

```
> (reply-more "goodbye cruel world")
"bye!"
> (reply-more "what is your favorite color?")
"I don't know"
> (reply-more "mine is lime green")
"huh?"
```

The use of square brackets for `cond` clauses is a convention. In Scheme, parentheses and square brackets are actually interchangable, as long as `(` is matched with `)` and `[` is matched with `]`. Using square brackets in a few key places makes Scheme code even more readable.

2.2.6 Function Calls, Again

In our earlier grammar of function calls, we oversimplified. The actual syntax of a function call allows an arbitrary expression for the function, instead of just an ⟨*id*⟩:

§4.3 "Function Calls" (later in this guide) explains more about function calls.

`(` ⟨*expr*⟩ ⟨*expr*⟩* `)`

The first ⟨*expr*⟩ is often an ⟨*id*⟩, such as `string-append` or `+`, but it can be anything that evaluates to a function. For example, it can be a conditional expression:

```
(define (double v)
  ((if (string? v) string-append +) v v))
```

```
> (double "hello")
"hellohello"
> (double 5)
10
```

Syntactically, the first expression in a function call could even be a number—but that leads to an error, since a number is not a function.

```
> (1 2 3 4)
procedure application: expected procedure, given: 1;
arguments were: 2 3 4
```

When you accidentally omit a function name or when you use parentheses around an expression, you'll most often get an "expected a procedure" error like this one.

2.2.7 Anonymous Functions with `lambda`

Programming in Scheme would be tedious if you had to name all of your numbers. Instead of writing `(+ 1 2)`, you'd have to write

§4.4 "Functions: `lambda`" (later in this guide) explains more about `lambda`.

```
> (define a 1)
```

```
> (define b 2)
> (+ a b)
3
```

It turns out that having to name all your functions can be tedious, too. For example, you might have a function `twice` that takes a function and an argument. Using `twice` is convenient if you already have a name for the function, such as `sqrt`:

```
(define (twice f v)
  (f (f v)))

> (twice sqrt 16)
2
```

If you want to call a function that is not yet defined, you could define it, and then pass it to `twice`:

```
(define (louder s)
  (string-append s "!"))

> (twice louder "hello")
"hello!!"
```

But if the call to `twice` is the only place where `louder` is used, it's a shame to have to write a whole definition. In Scheme, you can use a `lambda` expression to produce a function directly. The `lambda` form is followed by identifiers for the function's arguments, and then the function's body expressions:

`(` `lambda` `(` ⟨*id*⟩* `)` ⟨*expr*⟩$^+$ `)`

Evaluating a `lambda` form by itself produces a function:

```
> (lambda (s) (string-append s "!"))
#<procedure>
```

Using `lambda`, the above call to `twice` can be re-written as

```
> (twice (lambda (s) (string-append s "!"))
         "hello")
"hello!!"
> (twice (lambda (s) (string-append s "?!"))
         "hello")
"hello?!?!"
```

Another use of `lambda` is as a result for a function that generates functions:

```
(define (make-add-suffix s2)
  (lambda (s) (string-append s s2)))
```

```
> (twice (make-add-suffix "!") "hello")
"hello!!"
> (twice (make-add-suffix "?!") "hello")
"hello?!?!"
> (twice (make-add-suffix "...") "hello")
"hello......"
```

Scheme is a *lexically scoped* language, which means that `s2` in the function returned by `make-add-suffix` always refers to the argument for the call that created the function. In other words, the `lambda`-generated function "remembers" the right `s2`:

```
> (define louder (make-add-suffix "!"))
> (define less-sure (make-add-suffix "?"))
> (twice less-sure "really")
"really??"
> (twice louder "really")
"really!!"
```

We have so far referred to definitions of the form `(define` ⟨*id*⟩ ⟨*expr*⟩`)` as "non-function definitions." This characterization is misleading, because the ⟨*expr*⟩ could be a `lambda` form, in which case the definition is equivalent to using the "function" definition form. For example, the following two definitions of `louder` are equivalent:

```
(define (louder s)
  (string-append s "!"))

(define louder
  (lambda (s)
    (string-append s "!")))

> louder
#<procedure:louder>
```

Note that the expression for `louder` in the second case is an "anonymous" function written with `lambda`, but, if possible, the compiler infers a name, anyway, to make printing and error reporting as informative as possible.

2.2.8 Local Binding with `define`, `let`, and `let*`

It's time to retract another simplification in our grammar of Scheme. In the body of a function, definitions can appear before the body expressions:

`(` `define` `(` ⟨*id*⟩ ⟨*id*⟩* `)` ⟨*definition*⟩* ⟨*expr*⟩$^+$ `)`
`(` `lambda` `(` ⟨*id*⟩* `)` ⟨*definition*⟩* ⟨*expr*⟩$^+$ `)`

Definitions at the start of a function body are local to the function body.

§4.5.4 "Internal Definitions" (later in this guide) explains more about local (internal) definitions.

Examples:

```
(define (converse s)
  (define (starts? s2) ; local to converse
    (define len2 (string-length s2))  ; local to starts?
    (and (>= (string-length s) len2)
         (equal? s2 (substring s 0 len2))))
  (cond
   [(starts? "hello") "hi!"]
   [(starts? "goodbye") "bye!"]
   [else "huh?"]))

> (converse "hello!")
"hi!"
> (converse "urp")
"huh?"
> starts? ; outside of converse, so...
reference to undefined identifier: starts?
```

Another way to create local bindings is the `let` form. An advantage of `let` is that it can be used in any expression position. Also, `let` binds many identifiers at once, instead of requiring a separate `define` for each identifier.

§4.5.4 "Internal Definitions" (later in this guide) explains more about `let` and `let*`.

`(` `let` `(` {`[` ⟨*id*⟩ ⟨*expr*⟩ `]`}* `)` ⟨*expr*⟩+ `)`

Each binding clause is an ⟨*id*⟩ and a ⟨*expr*⟩ surrounded by square brackets, and the expressions after the clauses are the body of the `let`. In each clause, the ⟨*id*⟩ is bound to the result of the ⟨*expr*⟩ for use in the body.

```
> (let ([x 1]
        [y 2])
    (format "adding ~s and ~s produces ~s" x y (+ x y)))
"adding 1 and 2 produces 3"
```

The bindings of a `let` form are available only in the body of the `let`, so the binding clauses cannot refer to each other. The `let*` form, in contrast, allows later clauses to use earlier bindings:

```
> (let* ([x 1]
         [y 2]
         [z (+ x y)])
    (format "adding ~s and ~s produces ~s" x y z))
"adding 1 and 2 produces 3"
```

2.3 Lists, Iteration, and Recursion

Scheme is a dialect of the language Lisp, whose name originally stood for "LISt Processor." The built-in list datatype remains a prominent feature of the language.

The `list` function takes any number of values and returns a list containing the values:

```
> (list "red" "green" "blue")
("red" "green" "blue")
> (list 1 2 3 4 5)
(1 2 3 4 5)
```

As you can see, a list result prints in the REPL as a pair of parentheses wrapped around the printed form of the list elements. There's an opportunity for confusion here, because parentheses are used for both expressions, such as `(list "red" "green" "blue")`, and printed results, such as `("red" "green" "blue")`. Remember that, in the documentation and in DrScheme, parentheses for results are printed in blue, whereas parentheses for expressions are brown.

Many predefined functions operate on lists. Here are a few examples:

```
> (length (list "a" "b" "c"))         ; count the elements
3
> (list-ref (list "a" "b" "c") 0)     ; extract by position
"a"
> (list-ref (list "a" "b" "c") 1)
"b"
> (append (list "a" "b") (list "c")) ; combine lists
("a" "b" "c")
> (reverse (list "a" "b" "c"))        ; reverse order
("c" "b" "a")
> (member "d" (list "a" "b" "c"))     ; check for an element
#f
```

2.3.1 Predefined List Loops

In addition to simple operations like `append`, Scheme includes functions that iterate over the elements of a list. These iteration functions play much the same role as `for` in Java and other languages. The body of a Scheme iteration is packaged into a function to be applied to each element, so the `lambda` form becomes particularly handy in combination with iteration functions.

Different list-iteration functions combine iteration results in different ways. The `map` function uses the per-element results to create a new list:

```
> (map sqrt (list 1 4 9 16))
(1 2 3 4)
> (map (lambda (i)
         (string-append i "!"))
       (list "peanuts" "popcorn" "crackerjack"))
("peanuts!" "popcorn!" "crackerjack!")
```

The `andmap` and `ormap` functions combine the results by `and`ing or `or`ing:

```
> (andmap string? (list "a" "b" "c"))
#t
> (andmap string? (list "a" "b" 6))
#f
> (ormap number? (list "a" "b" 6))
#t
```

The `filter` function keeps elements for which the body result is true, and discards elements for which it is `#f`:

```
> (filter string? (list "a" "b" 6))
("a" "b")
> (filter positive? (list 1 -2 6 7 0))
(1 6 7)
```

The `map`, `andmap`, `ormap`, and `filter` functions can all handle multiple lists, instead of just a single list. The lists must all have the same length, and the given function must accept one argument for each list:

```
> (map (lambda (s n) (substring s 0 n))
       (list "peanuts" "popcorn" "crackerjack")
       (list 6 3 7))
("peanut" "pop" "cracker")
```

The `foldl` function generalizes some iteration functions. It uses the per-element function to both process an element and combine it with the "current" value, so the per-element function takes an extra first argument. Also, a starting "current" value must be provided before the lists:

```
> (foldl (lambda (v elem)
           (+ v (* elem elem)))
         0
         '(1 2 3))
12
```

Despite its generality, `foldl` is not as popular as the other functions. One reason is that `map`, `ormap`, `andmap`, and `filter` cover the most common kinds of list loops.

Scheme provides a general *list comprehension* form `for/list`, which builds a list by iterating through *sequences*. List comprehensions and related iteration forms are described in see §11 “Iterations and Comprehensions”.

2.3.2 List Iteration from Scratch

Although `map` and other iteration functions predefined, they are not primitive in any interesting sense. You can write equivalent iterations using a handful of list primitives.

Since a Scheme list is a linked list, the two core operations on a non-empty list are

- `first`: get the first thing in the list; and
- `rest`: get the rest of the list.

Examples:

```
> (first (list 1 2 3))
1
> (rest (list 1 2 3))
(2 3)
```

To create a new node for a linked list—that is, to add to the front of the list—use the `cons` function, which is short for “construct.” To get an empty list to start with, use the `empty` constant:

```
> empty
()
> (cons "head" empty)
("head")
> (cons "dead" (cons "head" empty))
("dead" "head")
```

To process a list, you need to be able to distinguish empty lists from non-empty lists, because `first` and `rest` work only on non-empty lists. The `empty?` function detects empty lists, and `cons?` detects non-empty lists:

```
> (empty? empty)
#t
> (empty? (cons "head" empty))
#f
> (cons? empty)
#f
> (cons? (cons "head" empty))
#t
```

With these pieces, you can write your own versions of the `length` function, `map` function, and more.

Examples:

```
(define (my-length lst)
  (cond
   [(empty? lst) 0]
   [else (+ 1 (my-length (rest lst)))]))

> (my-length empty)
0
> (my-length (list "a" "b" "c"))
3

(define (my-map f lst)
  (cond
   [(empty? lst) empty]
   [else (cons (f (first lst))
               (my-map f (rest lst)))]))

> (my-map string-upcase (list "ready" "set" "go"))
("READY" "SET" "GO")
```

If the derivation of the above definitions is mysterious to you, consider reading *How to Design Programs*. If you are merely suspicious of the use of recursive calls instead of a looping construct, then read on.

2.3.3 Tail Recursion

Both the `my-length` and `my-map` functions run in $O(n)$ time for a list of length n. This is easy to see by imagining how `(my-length (list "a" "b" "c"))` must evaluate:

```
(my-length (list "a" "b" "c"))
= (+ 1 (my-length (list "b" "c")))
= (+ 1 (+ 1 (my-length (list "c"))))
= (+ 1 (+ 1 (+ 1 (my-length (list)))))
= (+ 1 (+ 1 (+ 1 0)))
= (+ 1 (+ 1 1))
= (+ 1 2)
= 3
```

For a list with n elements, evaluation will stack up n `(+ 1 ...)` additions, and then finally add them up when the list is exhausted.

You can avoid piling up additions by adding along the way. To accumulate a length this way,

we need a function that takes both a list and the length of the list seem so far; the code below uses a local function `iter` that accumulates the length in an argument `len`:

```
#:eval list-eval
(define (my-length lst)
  ; local function iter:
  (define (iter lst len)
    (cond
     [(empty? lst) len]
     [else (iter (rest lst) (+ len 1))]))
  ; body of my-length calls iter:
  (iter lst 0))
```

Now evaluation looks like this:

```
(my-length (list "a" "b" "c"))
= (iter (list "a" "b" "c") 0)
= (iter (list "b" "c") 1)
= (iter (list "c") 2)
= (iter (list) 3)
3
```

The revised `my-length` runs in constant space, just as the evaluation steps above suggest. That is, when the result of a function call, like `(iter (list "b" "c") 1)`, is exactly the result of some other function call, like `(iter (list "c") 2)`, then the first one doesn't have to wait around for the second one, because that takes up space for no good reason.

This evaluation behavior is sometimes called *tail-call optimization*, but it's not merely an "optimization" in Scheme; it's a guarantee about the way the code will run. More precisely, an expression in *tail position* with respect to another expression does not take extra computation space over the other expression.

In the case of `my-map`, $O(n)$ space complexity is reasonable, since it has to generate a result of size $O(n)$. Nevertheless, you can reduce the constant factor by accumulating the result list. The only catch is that the accumulated list will be backwards, so you'll have to reverse it at the very end:

```
#:eval list-eval
(define (my-map f lst)
  (define (iter lst backward-result)
    (cond
     [(empty? lst) (reverse backward-result)]
     [else (iter (rest lst)
                 (cons (f (first lst))
                       backward-result))]))
  (iter lst empty))
```

It turns out that if you write

```
(define (my-map f lst)
  (for/list ([i lst])
    (f i)))
```

then the `for/list` form in the function both is expanded to essentially the same code as the `iter` local definition and use. The difference is merely syntactic convenience.

2.3.4 Recursion versus Iteration

The `my-length` and `my-map` examples demonstrate that iteration is just a special case of recursion. In many languages, it's important to try to fit as many computations as possible into iteration form. Otherwise, performance will be bad, and moderately large inputs can lead to stack overflow. Similarly, in Scheme, it is often important to make sure that tail recursion is used to avoid $O(n)$ space consumption when the computation is easily performed in constant space.

At the same time, recursion does not lead to particularly bad performance in Scheme, and there is no such thing as stack overflow; you can run out of memory if a computation involves too much context, but exhausting memory typically requires orders of magnitude deeper recursion than would trigger a stack overflow in other languages. These considerations, combined with the fact that tail-recursive programs automatically run the same as a loop, lead Scheme programmers to embrace recursive forms rather than avoid them.

Suppose, for example, that you want to remove consecutive duplicates from a list. While such a function can be written as a loop that remembers the previous element for each iteration, a Scheme programmer would more likely just write the following:

```
(define (remove-dups l)
  (cond
   [(empty? l) empty]
   [(empty? (rest l)) l]
   [else
    (let ([i (first l)])
      (if (equal? i (first (rest l)))
          (remove-dups (rest l))
          (cons i (remove-dups (rest l)))))]))

> (remove-dups (list "a" "b" "b" "b" "c" "c"))
("a" "b" "c")
```

In general, this function consumes $O(n)$ space for an input list of length n, but that's fine, since it produces an $O(n)$ result. If the input list happens to be mostly consecutive duplicates, then the resulting list can be much smaller than $O(n)$—and `remove-dups` will also use much

less than $O(n)$ space! The reason is that when the function discards duplicates, it returns the result of a `remove-dups` call directly, so the tail-call "optimization" kicks in:

```
(remove-dups (list "a" "b" "b" "b" "b" "b"))
= (cons "a" (remove-dups (list "b" "b" "b" "b" "b")))
= (cons "a" (remove-dups (list "b" "b" "b" "b")))
= (cons "a" (remove-dups (list "b" "b" "b")))
= (cons "a" (remove-dups (list "b" "b")))
= (cons "a" (remove-dups (list "b")))
= (cons "a" (list "b"))
= (list "a" "b")
```

2.4 Pairs, Lists, and Scheme Syntax

The `cons` function actually accepts any two values, not just a list for the second argument. When the second argument is not `empty` and not itself produced by `cons`, the result prints in a special way. The two values joined with `cons` are printed between parentheses, but with a dot (i.e., a period surrounded by whitespace) in between:

```
> (cons 1 2)
(1 . 2)
> (cons "banana" "split")
("banana" . "split")
```

Thus, a value produced by `cons` is not always a list. In general, the result of `cons` is a *pair*. The more traditional name for the `cons?` function is `pair?`, and we'll use the traditional name from now on.

The name `rest` also makes less sense for non-list pairs; the more traditional names for `first` and `rest` are `car` and `cdr`, respectively. (Granted, the traditional names are also nonsense. Just remember that "a" comes before "d," and `cdr` is pronounced "could-er.")

Examples:

```
> (car (cons 1 2))
1
> (cdr (cons 1 2))
2
> (pair? empty)
#f
> (pair? (cons 1 2))
#t
> (pair? (list 1 2 3))
#t
```

Scheme's pair datatype and its relation to lists is essentially a historical curiosity, along with

the dot notation for printing and the funny names `car` and `cdr`. Pairs are deeply wired into to the culture, specification, and implementation of Scheme, however, so they survive in the language.

You are perhaps most likely to encounter a non-list pair when making a mistake, such as accidentally reversing the arguments to `cons`:

```
> (cons (list 2 3) 1)
((2 3) . 1)
> (cons 1 (list 2 3))
(1 2 3)
```

Non-list pairs are used intentionally, sometimes. For example, the `make-immutable-hash` function takes a list of pairs, where the `car` of each pair is a key and the `cdr` is an arbitrary value.

The only thing more confusing to new Schemers than non-list pairs is the printing convention for pairs where the second element *is* a pair, but *is not* a list:

```
> (cons 0 (cons 1 2))
(0 1 . 2)
```

In general, the rule for printing a pair is as follows: use the dot notation always, but if the dot is immediately followed by an open parenthesis, then remove the dot, the open parenthesis, and the matching close parenthesis. Thus, `(0 . (1 . 2))` becomes `(0 1 . 2)`, and `(1 . (2 . (3 . ())))` becomes `(1 2 3)`.

2.4.1 Quoting Pairs and Symbols with `quote`

After you see

```
> (list (list 1) (list 2) (list 3))
((1) (2) (3))
```

enough times, you'll wish (or you're already wishing) that there was a way to write just `((1) (2) (3))` and have it mean the list of lists that prints as `((1) (2) (3))`. The `quote` form does exactly that:

```
> (quote ((1) (2) (3)))
((1) (2) (3))
> (quote ("red" "green" "blue"))
("red" "green" "blue")
> (quote ())
()
```

The `quote` form works with the dot notation, too, whether the quoted form is normalized by

the dot-parenthesis elimination rule or not:

```
> (quote (1 . 2))
(1 . 2)
> (quote (0 . (1 . 2)))
(0 1 . 2)
```

Naturally, lists of any kind can be nested:

```
> (list (list 1 2 3) 5 (list "a" "b" "c"))
((1 2 3) 5 ("a" "b" "c"))
> (quote ((1 2 3) 5 ("a" "b" "c")))
((1 2 3) 5 ("a" "b" "c"))
```

If you wrap an identifier with `quote`, then you get output that looks like an identifier:

```
> (quote jane-doe)
jane-doe
```

A value that prints like an identifier is a *symbol*. In the same way that parenthesized output should not be confused with expressions, a printed symbol should not be confused with an identifier. In particular, the symbol `(quote map)` has nothing to do with the `map` identifier or the predefined function that is bound to `map`, except that the symbol and the identifier happen to be made up of the same letters.

Indeed, the intrinsic value of a symbol is nothing more than its character content. In this sense, symbols and strings are almost the same thing, and the main difference is how they print. The functions `symbol->string` and `string->symbol` convert between them.

Examples:

```
> map
#<procedure:map>
> (quote map)
map
> (symbol? (quote map))
#t
> (symbol? map)
#f
> (procedure? map)
#t
> (string->symbol "map")
map
> (symbol->string (quote map))
"map"
```

When `quote` is used on a parenthesized sequence of identifiers, it creates a list of symbols:

```
> (quote (road map))
```

```
(road map)
> (car (quote (road map)))
road
> (symbol? (car (quote (road map))))
#t
```

2.4.2 Abbreviating `quote` with `'`

If `(quote (1 2 3))` still seems like too much typing, you can abbreviate by just putting `'` in front of `(1 2 3)`:

```
> '(1 2 3)
(1 2 3)
> 'road
road
> '((1 2 3) road ("a" "b" "c"))
((1 2 3) road ("a" "b" "c"))
```

In the documentation, `'` is printed in green along with the form after it, since the combination is an expression that is a constant. In DrScheme, only the `'` is colored green. DrScheme is more precisely correct, because the meaning of `quote` can vary depending on the context of an expression. In the documentation, however, we routinely assume that standard bindings are in scope, and so we paint quoted forms in green for extra clarity.

A `'` expands to a `quote` form in quite a literal way. You can see this if you put a `'` in front of a form that has a `'`:

```
> (car '(quote road))
quote
> (car ''road)
quote
```

Beware, however, that the REPL's printer recognizes the symbol `quote` when printing output, and then it uses `'` in the output:

```
> 'road
road
> ''road
'road
> '(quote road)
'road
```

2.4.3 Lists and Scheme Syntax

Now that you know the truth about pairs and lists, and now that you've seen `quote`, you're ready to understand the main way in which we have been simplifying Scheme's true syntax.

The syntax of Scheme is not defined directly in terms of character streams. Instead, the syntax is determined by two layers:

- a *read* layer, which turns a sequence of characters into lists, symbols, and other constants; and
- an *expand* layer, which processes the lists, symbols, and other constants to parse them as an expression.

The rules for printing and reading go together. For example, a list is printed with parentheses, and reading a pair of parentheses produces a list. Similarly, a non-list pair is printed with the dot notation, and a dot on input effectively runs the dot-notation rules in reverse to obtain a pair.

One consequence of the read layer for expressions is that you can use the dot notation in expressions that are not quoted forms:

```
> (+ 1 . (2))
3
```

This works because `(+ 1 . (2))` is just another way of writing `(+ 1 2)`. It is practically never a good idea to write application expressions using this dot notation; it's just a consequence of the way Scheme's syntax is defined.

Normally, `.` is allowed by the reader only with a parenthesized sequence, and only before the last element of the sequence. However, a pair of `.`s can also appear around a single element in a parenthesized sequence, as long as the element is not first or last. Such a pair triggers a reader conversion that moves the element between `.`s to the front of the list. The conversion enables a kind of general infix notation:

```
> (1 . < . 2)
#t
> '(1 . < . 2)
(< 1 2)
```

This two-dot convention is non-traditional, and it has essentially nothing to do with the dot notation for non-list pairs. PLT Scheme programmers use the infix convention sparingly—mostly for asymmetric binary operators such as `<` and `is-a?`.

3 Built-In Datatypes

The previous chapter introduced some of Scheme's built-in datatypes: numbers, booleans, strings, lists, and procedures. This section provides a more complete coverage of the built-in datatypes for simple forms of data.

3.1 Booleans

Scheme has two distinguished constants to represent boolean values: `#t` for true and `#f` for false. Uppercase `#T` and `#F` are parsed as the same values, but the lowercase forms are preferred.

The `boolean?` procedure recognizes the two boolean constants. In the result of a test expression for `if`, `cond`, `and`, `or`, etc., however, any value other than `#f` counts as true.

Examples:

```
> (= 2 (+ 1 1))
#t
> (boolean? #t)
#t
> (boolean? #f)
#t
> (boolean? "no")
#f
> (if "no" 1 0)
1
```

3.2 Numbers

A Scheme *number* is either exact or inexact:

- An *exact* number is either
  - an arbitrarily large or small integer, such as `5`, `99999999999999999`, or `-17`;
  - a rational that is exactly the ratio of two arbitrarily small or large integers, such as `1/2`, `9999999999999999/2`, or `-3/4`; or
  - a complex number with exact real and imaginary parts (where the imaginary part is not zero), such as `1+2i` or `1/2+3/4i`.
- An *inexact* number is either

- an IEEE floating-point representation of a number, such as `2.0` or `3.14e+87`, where the IEEE infinities and not-a-number are written `+inf.0`, `-inf.0`, and `+nan.0` (or `-nan.0`); or
- a complex number with real and imaginary parts that are IEEE floating-point representations, such as `2.0+3.0i` or `-inf.0+nan.0i`; as a special case, an inexact complex number can have an exact zero real part with an inexact imaginary part.

Inexact numbers print with a decimal point or exponent specifier, and exact numbers print as integers and fractions. The same conventions apply for reading number constants, but `#e` or `#i` can prefix a number to force its parsing as an exact or inexact number. The prefixes `#b`, `#o`, and `#x` specify binary, octal, and hexadecimal interpretation of digits.

§12.6.3 "Reading Numbers" in §"**Reference**: PLT Scheme" documents the fine points of the syntax of numbers.

Examples:

```
> 0.5
0.5
> #e0.5
1/2
> #x03BB
955
```

Computations that involve an inexact number produce inexact results, so that inexactness acts as a kind of taint on numbers. Beware, however, that Scheme offers no "inexact booleans", so computations that branch on the comparison of inexact numbers can nevertheless produce exact results. The procedures `exact->inexact` and `inexact->exact` convert between the two types of numbers.

Examples:

```
> (/ 1 2)
1/2
> (/ 1 2.0)
0.5
> (if (= 3.0 2.999) 1 2)
2
> (inexact->exact 0.1)
3602879701896397/36028797018963968
```

Inexact results are also produced by procedures such as `sqrt`, `log`, and `sin` when an exact result would require representing real numbers that are not rational. Scheme can represent only rational numbers and complex numbers with rational parts.

Examples:

```
> (sin 0)   ; rational...
0
> (sin 1/2) ; not rational...
0.479425538604203
```

In terms of performance, computations with small integers are typically the fastest, where "small" means that the number fits into one bit less than the machine's word-sized representation for signed numbers. Computation with very large exact integers or with non-integer exact numbers can be much more expensive than computation with inexact numbers.

```
(define (sigma f a b)
  (if (= a b)
      0
      (+ (f a) (sigma f (+ a 1) b))))

> (time (round (sigma (lambda (x) (/ 1 x)) 1 2000)))
cpu time: 127 real time: 128 gc time: 0
8
> (time (round (sigma (lambda (x) (/ 1.0 x)) 1 2000)))
cpu time: 0 real time: 0 gc time: 0
8.0
```

The number categories *integer*, *rational*, *real* (always rational), and *complex* are defined in the usual way, and are recognized by the procedures `integer?`, `rational?`, `real?`, and `complex?`, in addition to the generic `number?`. A few mathematical procedures accept only real numbers, but most implement standard extensions to complex numbers.

Examples:

```
> (integer? 5)
#t
> (complex? 5)
#t
> (integer? 5.0)
#t
> (integer? 1+2i)
#f
> (complex? 1+2i)
#t
> (complex? 1.0+2.0i)
#t
> (abs -5)
5
> (abs -5+2i)
abs: expects argument of type <real number>; given -5+2i
> (sin -5+2i)
3.6076607742131563+1.0288031496599335i
```

The `=` procedure compares numbers for numerical equality. If it is given both inexact and exact numbers to compare, it essentially converts the inexact numbers to exact before comparing. The `eqv?` (and therefore `equal?`) procedure, in contrast, compares numbers considering both exactness and numerical equality.

Examples:

```
> (= 1 1.0)
#t
> (eqv? 1 1.0)
#f
```

Beware of comparisons involving inexact numbers, which by their nature can have surprising behavior. Even apparently simple inexact numbers may not mean what you think they mean; for example, while a base-2 IEEE floating-point number can represent `1/2` exactly, it can only approximate `1/10`:

Examples:

```
> (= 1/2 0.5)
#t
> (= 1/10 0.1)
#f
> (inexact->exact 0.1)
3602879701896397/36028797018963968
```

§3.2 "Numbers" in §"**Reference**: PLT Scheme" provides more on numbers and number procedures.

3.3 Characters

A Scheme *character* corresponds to a Unicode *scalar value*. Roughly, a scalar value is an unsigned integer whose representation fits into 21 bits, and that maps to some notion of a natural-language character or piece of a character. Technically, a scalar value is a simpler notion than the concept called a "character" in the Unicode standard, but it's an approximation that works well for many purposes. For example, any accented Roman letter can be represented as a scalar value, as can any Chinese character.

Although each Scheme character corresponds to an integer, the character datatype is separate from numbers. The `char->integer` and `integer->char` procedures convert between scalar-value numbers and the corresponding character.

A printable character normally prints as `#\` followed by the represented character. An unprintable character normally prints as `#\u` followed by the scalar value as hexadecimal number. A few characters are printed specially; for example, the space and linefeed characters print as `#\space` and `#\newline`, respectively.

§12.6.13 "Reading Characters" in §"**Reference**: PLT Scheme" documents the fine points of the syntax of characters.

Examples:

```
> (integer->char 65)
#\A
> (char->integer #\A)
65
> #\λ
#\λ
> #\u03BB
```

```
#\λ
> (integer->char 17)
#\u0011
> (char->integer #\space)
32
```

The `display` procedure directly writes a character to the current output port (see §8 "Input and Output"), in contrast to the character-constant syntax used to print a character result.

Examples:

```
> #\A
#\A
> (display #\A)
A
```

Scheme provides several classification and conversion procedures on characters. Beware, however, that conversions on some Unicode characters work as a human would expect only when they are in a string (e.g., upcasing "ß" or downcasing "Σ").

Examples:

```
> (char-alphabetic? #\A)
#t
> (char-numeric? #\0)
#t
> (char-whitespace? #\newline)
#t
> (char-downcase #\A)
#\a
> (char-upcase #\ß)
#\ß
```

The `char=?` procedure compares two or more characters, and `char-ci=?` compares characters ignoring case. The `eqv?` and `equal?` procedures behave the same as `char=?` on characters; use `char=?` when you want to more specifically declare that the values being compared are characters.

Examples:

```
> (char=? #\a #\A)
#f
> (char-ci=? #\a #\A)
#t
> (eqv? #\a #\A)
#f
```

§3.5 "Characters" in §"**Reference**: PLT Scheme" provides more on characters and character procedures.

3.4 Strings (Unicode)

A *string* is a fixed-length array of characters. It prints using doublequotes, where double-quote and backslash characters within the string are escaped with backslashes. Other common string escapes are supported, including `\n` for a linefeed, `\r` for a carriage return, octal escapes using `\` followed by up to three octal digits, and hexadecimal escapes with `\u` (up to four digits). Unprintable characters in a string are normally shown with `\u` when the string is printed.

§12.6.6 "Reading Strings" in §"**Reference**: PLT Scheme" documents the fine points of the syntax of strings.

The `display` procedure directly writes the characters of a string to the current output port (see §8 "Input and Output"), in contrast to the string-constant syntax used to print a string result.

Examples:

```
> "Apple"
"Apple"
> "\u03BB"
"λ"
> (display "Apple")
Apple
> (display "a \"quoted\" thing")
a "quoted" thing
> (display "two\nlines")
two
lines
> (display "\u03BB")
λ
```

A string can be mutable or immutable; strings written directly as expressions are immutable, but most other strings are mutable. The `make-string` procedure creates a mutable string given a length and optional fill character. The `string-ref` procedure accesses a character from a string (with 0-based indexing); the `string-set!` procedure changes a character in a mutable string.

Examples:

```
> (string-ref "Apple" 0)
#\A
> (define s (make-string 5 #\.))
> s
"....."
> (string-set! s 2 #\λ)
> s
"..λ.."
```

String ordering and case operations are generally *locale-independent*; that is, they work the same for all users. A few *locale-dependent* operations are provided that allow the way

that strings are case-folded and sorted to depend on the end-user's locale. If you're sorting strings, for example, use `string<?` or `string-ci<?` if the sort result should be consistent across machines and users, but use `string-locale<?` or `string-locale-ci<?` if the sort is purely to order strings for an end user.

Examples:

```
> (string<? "apple" "Banana")
#f
> (string-ci<? "apple" "Banana")
#t
> (string-upcase "Straße")
"STRASSE"
> (parameterize ([current-locale "C"])
    (string-locale-upcase "Straße"))
"STRAßE"
```

For working with plain ASCII, working with raw bytes, or encoding/decoding Unicode strings as bytes, use byte strings.

§3.3 "Strings" in §"**Reference**: PLT Scheme" provides more on strings and string procedures.

3.5 Bytes and Byte Strings

A *byte* is an exact integer between `0` and `255`, inclusive. The `byte?` predicate recognizes numbers that represent bytes.

Examples:

```
> (byte? 0)
#t
> (byte? 256)
#f
```

A *byte string* is similar to a string—see §3.4 "Strings (Unicode)"—but its content is a sequence of bytes instead of characters. Byte strings can be used in applications that process pure ASCII instead of Unicode text. The printed form of a byte string supports such uses in particular, because a byte string prints like the ASCII decoding of the byte string, but prefixed with a `#`. Unprintable ASCII characters or non-ASCII bytes in the byte string are written with octal notation.

§12.6.6 "Reading Strings" in §"**Reference**: PLT Scheme" documents the fine points of the syntax of byte strings.

Examples:

```
> #"Apple"
#"Apple"
> (bytes-ref #"Apple" 0)
65
> (make-bytes 3 65)
#"AAA"
> (define b (make-bytes 2 0))
```

```
> b
#"\0\0"
> (bytes-set! b 0 1)
> (bytes-set! b 1 255)
> b
#"\1\377"
```

The `display` form of a byte string writes its raw bytes to the current output port (see §8 “Input and Output”). Technically, `display` of a normal (i.e,. character) string prints the UTF-8 encoding of the string to the current output port, since output is ultimately defined in terms of bytes; `display` of a byte string, however, writes the raw bytes with no encoding. Along the same lines, when this documentation shows output, it technically shows the UTF-8-decoded form of the output.

Examples:

```
> (display #"Apple")
Apple
> (display "\316\273")  ; same as ""

> (display #"\316\273") ; UTF-8 encoding of λ
λ
```

For explicitly converting between strings and byte strings, Scheme supports three kinds of encodings directly: UTF-8, Latin-1, and the current locale’s encoding. General facilities for byte-to-byte conversions (especially to and from UTF-8) fill the gap to support arbitrary string encodings.

Examples:

```
> (bytes->string/utf-8 #"\316\273")
"λ"
> (bytes->string/latin-1 #"\316\273")
""
> (parameterize ([current-locale "C"])  ; C locale supports ASCII,
    (bytes->string/locale #"\316\273")) ; only, so...
bytes->string/locale: byte string is not a valid encoding
for the current locale: #"\316\273"
> (let ([cvt (bytes-open-converter "cp1253" ; Greek code page
                                   "UTF-8")]
        [dest (make-bytes 2)])
    (bytes-convert cvt #"\353" 0 1 dest)
    (bytes-close-converter cvt)
    (bytes->string/utf-8 dest))
"λ"
```

§3.4 “Byte Strings” in §“**Reference**: PLT Scheme” provides more on byte strings and byte-string procedures.

3.6 Symbols

A *symbol* is an atomic value that prints like an identifier. An expression that starts with `'` and continues with an identifier produces a symbol value.

Examples:
```
> 'a
a
> (symbol? 'a)
#t
```

For any sequence of characters, exactly one corresponding symbol is *interned*; calling the `string->symbol` procedure, or `read`ing a syntactic identifier, produces an interned symbol. Since interned symbols can be cheaply compared with `eq?` (and thus `eqv?` or `equal?`), they serves as a convenient values to use for tags and enumerations.

Symbols are case-sensitive. By using a `#ci` prefix or in other ways, the reader can be made to case-fold character sequences to arrive at a symbol, but the reader preserves case by default.

Examples:
```
> (eq? 'a 'a)
#t
> (eq? 'a (string->symbol "a"))
#t
> (eq? 'a 'b)
#f
> (eq? 'a 'A)
#f
> #ci'A
a
```

Any string (i.e., any character sequence) can be supplied to `string->symbol` to obtain the corresponding symbol. For reader input, any character can appear directly in an identifier, except for whitespace and the following special characters:

```
( ) [ ] { } " , ' ` ; # | \
```

Actually, `#` is disallowed only at the beginning of a symbol, and then only if not followed by `%`; otherwise, `#` is allowed, too. Also, `.` by itself is not a symbol.

Whitespace or special characters can be included in an identifier by quoting them with `|` or `\`. These quoting mechanisms are used in the printed form of identifiers that contain special characters or that might otherwise look like numbers.

Examples:
```
> (string->symbol "one, two")
|one, two|
```

```
> (string->symbol "6")
|6|
```

The `display` form of a symbol is the same as the corresponding string.

§12.6.2 "Reading Symbols" in §"**Reference**: PLT Scheme" documents the fine points of the syntax of symbols.

Examples:

```
> (display 'Apple)
Apple
> (display '|6|)
6
```

The `gensym` and `string->uninterned-symbol` procedures generate fresh *uninterned* symbols that are not equal (according to `eq?`) to any previously interned or uninterned symbol. Uninterned symbols are useful as fresh tags that cannot be confused with any other value.

Examples:

```
> (define s (gensym))
> s
g42
> (eq? s 'g42)
#f
> (eq? 'a (string->uninterned-symbol "a"))
#f
```

§3.6 "Symbols" in §"**Reference**: PLT Scheme" provides more on symbols.

3.7 Keywords

A *keyword* value is similar to a symbol (see §3.6 "Symbols"), but its printed form is prefixed with `#:`.

§12.6.14 "Reading Keywords" in §"**Reference**: PLT Scheme" documents the fine points of the syntax of keywords.

Examples:

```
> (string->keyword "apple")
#:apple
> '#:apple
#:apple
> (eq? '#:apple (string->keyword "apple"))
#t
```

More precisely, a keyword is analogous to an identifier; in the same way that an identifier can be quoted to produce a symbol, a keyword can be quoted to produce a value. The same term "keyword" is used in both cases, but we sometimes use *keyword value* to refer more specifically to the result of a quote-keyword expression or of `string->keyword`. An unquoted keyword is not an expression, just as an unquoted identifier does not produce a symbol:

Examples:

```
> not-a-symbol-expression
reference to undefined identifier: not-a-symbol-expression
> #:not-a-keyword-expression
eval:4:0: #%datum: keyword used as an expression in:
#:not-a-keyword-expression
```

Despite their similarities, keywords are used in a different way than identifiers or symbols. Keywords are intended for use (unquoted) as special markers in argument lists and in certain syntactic forms. For run-time flags and enumerations, use symbols instead of keywords. The example below illustrates the distinct roles of keywords and symbols.

Examples:

```
> (define dir (find-system-path 'temp-dir)) ; not '#:temp-dir
> (with-output-to-file (build-path dir "stuff.txt")
    (lambda () (printf "example\n"))
    ; optional #:mode argument can be 'text or 'binary
    #:mode 'text
    ; optional #:exists argument can be 'replace, 'truncate, ...
    #:exists 'replace)
```

3.8 Pairs and Lists

A *pair* joins two arbitrary values. The `cons` procedure constructs pairs, and the `car` and `cdr` procedures extract the first and second elements of the pair, respectively. The `pair?` predicate recognizes pairs.

Some pairs print by wrapping parentheses around the printed forms of the two pair elements, putting a `.` between them.

Examples:

```
> (cons 1 2)
(1 . 2)
> (cons (cons 1 2) 3)
((1 . 2) . 3)
> (car (cons 1 2))
1
> (cdr (cons 1 2))
2
> (pair? (cons 1 2))
#t
```

A *list* is a combination of pairs that creates a linked list. More precisely, a list is either the empty list `null`, or it is a pair whose first element is a list element and whose second element is a list. The `list?` predicate recognizes lists. The `null?` predicate recognizes the empty list.

A list prints as a pair of parentheses wrapped around the list elements.

Examples:

```
> null
()
> (cons 0 (cons 1 (cons 2 null)))
(0 1 2)
> (list? null)
#t
> (list? (cons 1 (cons 2 null)))
#t
> (list? (cons 1 2))
#f
```

An expression with `'` followed by the printed form of a pair or list produces a pair or list constant.

Examples:

```
> '()
()
> '(1 . 2)
(1 . 2)
> '(1 2 3)
(1 2 3)
```

Pairs are immutable (contrary to Lisp tradition), and `pair?` and `list?` recognize immutable pairs and lists, only. The `mcons` procedure creates a mutable pair, which works with `set-mcar!` and `set-mcdr!`, as well as `mcar` and `mcdr`.

Examples:

```
> (define p (mcons 1 2))
> p
{1 . 2}
> (pair? p)
#f
> (mpair? p)
#t
> (set-mcar! p 0)
> p
{0 . 2}
```

Among the most important predefined procedures on lists are those that iterate through the list's elements:

```
> (map (lambda (i) (/ 1 i))
       '(1 2 3))
(1 1/2 1/3)
```

```
> (andmap (lambda (i) (i . < . 3))
          '(1 2 3))
#f
> (ormap (lambda (i) (i . < . 3))
         '(1 2 3))
#t
> (filter (lambda (i) (i . < . 3))
          '(1 2 3))
(1 2)
> (foldl (lambda (v i) (+ v i))
         10
         '(1 2 3))
16
> (for-each (lambda (i) (display i))
            '(1 2 3))
123
> (member "Keys"
          '("Florida" "Keys" "U.S.A."))
("Keys" "U.S.A.")
> (assoc 'where
         '((when "3:30") (where "Florida") (who "Mickey")))
(where "Florida")
```

3.9 Vectors

§3.9 "Pairs and Lists" in §"**Reference**: PLT Scheme" provides more on pairs and lists.

A *vector* is a fixed-length array of arbitrary values. Unlike a list, a vector supports constant-time access and update of its elements.

A vector prints similar to a list—as a parenthesized sequence of its elements—but a vector is prefixed with `#`. For a vector as an expression, an optional length can be supplied. Also, a vector as an expression implicitly quotes the forms for its content, which means that identifiers and parenthesized forms in a vector constant represent symbols and lists.

§12.6.9 "Reading Vectors" in §"**Reference**: PLT Scheme" documents the fine points of the syntax of vectors.

Examples:

```
> #("a" "b" "c")
#("a" "b" "c")
> #(name (that tune))
#(name (that tune))
> (vector-ref #("a" "b" "c") 1)
"b"
> (vector-ref #(name (that tune)) 1)
(that tune)
```

Like strings, a vector is either mutable or immutable, and vectors written directly as expressions are immutable.

Vector can be converted to lists and vice-versa via `list->vector` and `vector->list`; such conversions are particularly useful in combination with predefined procedures on lists. When allocating extra lists seems too expensive, use consider using looping forms like `fold-for`, which recognize vectors as well as lists.

Examples:

```
> (list->vector (map string-titlecase
                     (vector->list #("three" "blind" "mice"))))
#("Three" "Blind" "Mice")
```

§3.11 "Vectors" in §"**Reference**: PLT Scheme" provides more on vectors and vector procedures.

3.10 Hash Tables

A *hash table* implements a mapping from keys to values, where both keys and values can be arbitrary Scheme values, and access and update to the table are normally constant-time operations. Keys are compared using `equal?` or `eq?`, depending on whether the hash table is created with `make-hash` or `make-hasheq`.

Examples:

```
> (define ht (make-hash))
> (hash-set! ht "apple" '(red round))
> (hash-set! ht "banana" '(yellow long))
> (hash-ref ht "apple")
(red round)
> (hash-ref ht "coconut")
hash-ref: no value found for key: "coconut"
> (hash-ref ht "coconut" "not there")
"not there"
```

A literal hash table can be written as an expression by using `#hash` (for an `equal?`-based table) or `#hasheq` (for an `eq?`-based table). A parenthesized sequence must immediately follow `#hash` or `#hasheq`, where each element is a sequence is a dotted key–value pair. Literal hash tables are immutable, but they can be extended functionally (producing a new hash table without changing the old one) using `hash-set`.

Examples:

```
> (define ht #hash(("apple" . red) ("banana" . yellow)))
> (hash-ref ht "apple")
red
> (define ht2 (hash-set ht "coconut" 'brown))
> (hash-ref ht "coconut")
hash-ref: no value found for key: "coconut"
> (hash-ref ht2 "coconut")
brown
> ht2
#hash(("apple" . red) ("banana" . yellow) ("coconut" . brown))
```

A hash table can optionally retain its keys *weakly*, so each mapping is retained only so long as the key is retained elsewhere.

§12.6.11 "Reading Hash Tables" in §"**Reference**: PLT Scheme" documents the fine points of the syntax of hash table literals.

Examples:

```
> (define ht (make-weak-hasheq))
> (hash-set! ht (gensym) "can you see me?")
> (collect-garbage)
> (hash-count ht)
0
```

Beware that even a weak hash table retains its values strongly, as long as the corresponding key is accessible. This creates a catch-22 dependency when a value refers back to its key, so that the mapping is retained permanently. To break the cycle, map the key to an ephemeron that pairs the value with its key (in addition to the implicit pairing of the hash table).

Examples:

```
> (define ht (make-weak-hasheq))
> (let ([g (gensym)])
    (hash-set! ht g (list g)))
> (collect-garbage)
> (hash-count ht)
1

> (define ht (make-weak-hasheq))
> (let ([g (gensym)])
    (hash-set! ht g (make-ephemeron g (list g))))
> (collect-garbage)
> (hash-count ht)
0
```

§3.13 "Hash Tables" in §"**Reference**: PLT Scheme" provides more on hash tables and hash-table procedures.

3.11 Boxes

A *box* is like a single-element vector. It prints as `#&` followed by the printed form of the boxed value. A `#&` form can also be used as an expression, but since the resulting box is constant, it has practically no use.

Examples:

```
> (define b (box "apple"))
> b
#&"apple"
> (unbox b)
"apple"
> (set-box! b '(banana boat))
```

```
> b
#&(banana boat)
```

§3.12 “Boxes” in §“**Reference**: PLT Scheme” provides more on boxes and box procedures.

3.12 Void and Undefined

Some procedures or expression forms have no need for a result value. For example, the `display` procedure is called only for the side-effect of writing output. In such cases the result value is normally a special constant that prints as `#<void>`. When the result of an expression is simply `#<void>`, the REPL does not print anything.

The `void` procedure takes any number of arguments and returns `#<void>`. (That is, the identifier `void` is bound to a procedure that returns `#<void>`, instead of being bound directly to `#<void>`.)

Examples:

```
> (void)
> (void 1 2 3)
> (list (void))
(#<void>)
```

A constant that prints as `#<undefined>` is used as the result of a reference to a local binding when the binding is not yet initialized. Such early references are not possible for bindings that correspond to procedure arguments, `let` bindings, or `let*` bindings; early reference requires a recursive binding context, such as `letrec` or local `define`s in a procedure body. Also, early references to top-level and module-level bindings raise an exception, instead of producing `#<undefined>`. For these reasons, `#<undefined>` rarely appears.

```
(define (strange)
  (define x x)
  x)

> (strange)
#<undefined>
```

4 Expressions and Definitions

The §2 “Scheme Essentials” chapter introduced some of Scheme’s syntactic forms: definitions, procedure applications, conditionals, and so on. This section provides more details on those forms, plus a few additional basic forms.

4.1 Notation

This chapter (and the rest of the documentation) uses a slightly different notation than the character-based grammars of the §2 “Scheme Essentials” chapter. The grammar for a use of a syntactic form `something` is shown like this:

```
(something [id ...+] an-expr ...)
```

The italicized meta-variables in this specification, such as *id* and *an-expr*, use the syntax of Scheme identifiers, so *an-expr* is one meta-variable. A naming convention implicitly defines the meaning of many meta-variables:

- A meta-variable that ends in *id* stands for an identifier, such as `x` or `my-favorite-martian`.
- A meta-identifier that ends in *keyword* stands for a keyword, such as `#:tag`.
- A meta-identifier that ends with *expr* stands for any sub-form, and it will be parsed as an expression.
- A meta-identifier that ends with *body* stands for any sub-form; it will be parsed as either a local definition or an expression. A *body* can parse as a definition only if it is not preceded by any expression, and the last *body* must be an expression; see also §4.5.4 “Internal Definitions”.

Square brackets in the grammar indicate a parenthesized sequence of forms, where square brackets are normally used (by convention). That is, square brackets *do not* mean optional parts of the syntactic form.

A `...` indicates zero or more repetitions of the preceding form, and `...+` indicates one or more repetitions of the preceding datum. Otherwise, non-italicized identifiers stand for themselves.

Based on the above grammar, then, here are a few conforming uses of `something`:

```
(something [x])
```

```
(something [x] (+ 1 2))
(something [x my-favorite-martian x] (+ 1 2) #f)
```

Some syntactic-form specifications refer to meta-variables that are not implicitly defined and not previously defined. Such meta-variables are defined after the main form, using a BNF-like format for alternatives:

```
(something-else [thing ...+] an-expr ...)

thing = thing-id
      | thing-keyword
```

The above example says that, within a `something-else` form, a *thing* is either an identifier or a keyword.

4.2 Identifiers and Binding

The context of an expression determines the meaning of identifiers that appear in the expression. In particular, starting a module with the language `scheme`, as in

```
#lang scheme
```

means that, within the module, the identifiers described in this guide start with the meaning described here: `cons` refers to the function that creates a pair, `car` refers to the function that extracts the first element of a pair, and so on.

§3.6 "Symbols" introduces the syntax of identifiers.

Forms like `define`, `lambda`, and `let` associate a meaning with one or more identifiers; that is, they *bind* identifiers. The part of the program for which the binding applies is the *scope* of the binding. The set of bindings in effect for a given expression is the expression's *environment*.

For example, in

```
#lang scheme

(define f
  (lambda (x)
    (let ([y 5])
      (+ x y))))

(f 10)
```

the `define` is a binding of `f`, the `lambda` has a binding for `x`, and the `let` has a binding for

`y`. The scope of the binding for `f` is the entire module; the scope of the `x` binding is `(let ([y 5]) (+ x y))`; and the scope of the `y` binding is just `(+ x y)`. The environment of `(+ x y)` includes bindings for `y`, `x`, and `f`, as well as everything in `scheme`.

A module-level `define` can bind only identifiers that are not already bound within the module. For example, `(define cons 1)` is a syntax error in a `scheme` module, since `cons` is provided by `scheme`. A local `define` or other binding forms, however, can give a new local binding for an identifier that already has a binding; such a binding *shadows* the existing binding.

Examples:

```
(define f
  (lambda (append)
    (define cons (append "ugly" "confusing"))
    (let ([append 'this-was])
      (list append cons))))

> (f list)
(this-was ("ugly" "confusing"))
```

Even identifiers like `define` and `lambda` get their meanings from bindings, though they have *transformer* bindings (which means that they indicate syntactic forms) instead of value bindings. Since `define` has a transformer binding, the identifier `define` cannot be used by itself to get a value. However, the normal binding for `define` can be shadowed.

Examples:

```
> define
eval:1:0: define: bad syntax in: define
> (let ([define 5]) define)
5
```

Shadowing standard bindings in this way is rarely a good idea, but the possibility is an inherent part of Scheme's flexibility.

4.3 Function Calls (Procedure Applications)

An expression of the form

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

is a function call—also known as a *procedure application*—when *`proc-expr`* is not an identifier that is bound as a syntax transformer (such as `if` or `define`).

4.3.1 Evaluation Order and Arity

A function call is evaluated by first evaluating the *proc-expr* and all *arg-expr*s in order (left to right). Then, if *proc-expr* produces a function that accepts as many arguments as supplied *arg-expr*s, the function is called. Otherwise, an exception is raised.

Examples:

```
> (cons 1 null)
(1)
> (+ 1 2 3)
6
> (cons 1 2 3)
cons: expects 2 arguments, given 3: 1 2 3
> (1 2 3)
procedure application: expected procedure, given: 1;
arguments were: 2 3
```

Some functions, such as `cons`, accept a fixed number of arguments. Some functions, such as `+` or `list`, accept any number of arguments. Some functions accept a range of argument counts; for example `substring` accepts either two or three arguments. A function's *arity* is the number of arguments that it accepts.

4.3.2 Keyword Arguments

Some functions accept *keyword arguments* in addition to by-position arguments. For that case, an *arg* can be an *arg-keyword arg-expr* sequence instead of just a *arg-expr*:

§3.7 "Keywords" introduces keywords.

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

arg = arg-expr
    | arg-keyword arg-expr
```

For example,

```
(go "super.ss" #:mode 'fast)
```

calls the function bound to `go` with `"super.ss"` as a by-position argument, and with `'fast` as an argument associated with the `#:mode` keyword. A keyword is implicitly paired with the expression that follows it.

Since a keyword by itself is not an expression, then

```
(go "super.ss" #:mode #:fast)
```

is a syntax error. The `#:mode` keyword must be followed by an expression to produce an argument value, and `#:fast` is not an expression.

The order of keyword *arg*s determines the order in which *arg-expr*s are evaluated, but a function accepts keyword arguments independent of their position in the argument list. The above call to `go` can be equivalently written

```
(go #:mode 'fast "super.ss")
```

§2.5 "Procedure Applications and **#%app**" in §"**Reference**: PLT Scheme" provides more on procedure applications.

4.3.3 The `apply` Function

The syntax for function calls supports any number of arguments, but a specific call always specifies a fixed number of arguments. As a result, a function that takes a list of arguments cannot directly apply a function like `+` to all of the items in the list:

```
(define (avg lst) ; doesn't work...
  (/ (+ lst) (length lst)))

> (avg '(1 2 3))
+: expects argument of type <number>; given (1 2 3)

(define (avg lst) ; doesn't always work...
 (/ (+ (list-ref lst 0) (list-ref lst 1) (list-ref lst 2))
    (length lst)))

> (avg '(1 2 3))
2
> (avg '(1 2))
list-ref: index 2 too large for list: (1 2)
```

The `apply` function offers a way around this restriction. It takes a function and a *list* arguments, and it applies the function to the arguments:

```
(define (avg lst)
  (/ (apply + lst) (length lst)))

> (avg '(1 2 3))
2
> (avg '(1 2))
3/2
> (avg '(1 2 3 4))
5/2
```

As a convenience, the `apply` function accepts additional arguments between the function and the list. The additional arguments are effectively `cons`ed onto the argument list:

```
(define (anti-sum lst)
  (apply - 0 lst))

> (anti-sum '(1 2 3))
-6
```

The `apply` function supports only by-position arguments. To apply a function with keyword arguments, use the `keyword-apply` function, which accepts a function to apply and three lists. The first two lists are in parallel, where the first list contains keywords (sorted by `keyword<`), and the second list contains a corresponding argument for each keyword. The third list contains by-position function arguments, as for `apply`.

```
(keyword-apply go
               '(#:mode)
               '(fast)
               '("super.ss"))
```

4.4 Functions (Procedures): `lambda`

A `lambda` expression creates a function. In the simplest case, a `lambda` expression has the form

```
(lambda (arg-id ...)
  body ...+)
```

A `lambda` form with *n* *arg-id*s accepts *n* arguments:

```
> ((lambda (x) x)
   1)
1
> ((lambda (x y) (+ x y))
   1 2)
3
> ((lambda (x y) (+ x y))
   1)
#<procedure>: expects 2 arguments, given 1: 1
```

4.4.1 Declaring a Rest Argument

A `lambda` expression can also have the form

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

That is, a `lambda` expression can have a single *rest-id* that is not surrounded by parentheses. The resulting function accepts any number of arguments, and the arguments are put into a list bound to *rest-id*.

Examples:

```
> ((lambda x x)
   1 2 3)
(1 2 3)
> ((lambda x x))
()
> ((lambda x (car x))
   1 2 3)
1
```

Functions with a *rest-id* often use `apply` to call another function that accepts any number of arguments.

§4.3.3 “The `apply` Function” describes `apply`.

Examples:

```
(define max-mag
  (lambda nums
    (apply max (map magnitude nums))))

> (max 1 -2 0)
1
> (max-mag 1 -2 0)
2
```

The `lambda` form also supports required arguments combined with a *rest-id*:

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

The result of this form is a function that requires at least as many arguments as *arg-id*s, and also accepts any number of additional arguments.

Examples:

```
(define max-mag
  (lambda (num . nums)
    (apply max (map magnitude (cons num nums)))))
```

```
> (max-mag 1 -2 0)
2
> (max-mag)
procedure max-mag: expects at least 1 argument, given 0
```

A *rest-id* variable is sometimes called a *rest argument*, because it accepts the "rest" of the function arguments.

4.4.2 Declaring Optional Arguments

Instead of just an identifier, an argument (other than a rest argument) in a `lambda` form can be specified with an identifier and a default value:

```
(lambda gen-formals
  body ...+)

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

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

A argument of the form `[arg-id default-expr]` is optional. When the argument is not supplied in an application, *default-expr* produces the default value. The *default-expr* can refer to any preceding *arg-id*, and every following *arg-id* must have a default as well.

Examples:

```
(define greet
  (lambda (given [surname "Smith"])
    (string-append "Hello, " given " " surname)))

> (greet "John")
"Hello, John Smith"
> (greet "John" "Doe")
"Hello, John Doe"

(define greet
  (lambda (given [surname (if (equal? given "John")
                              "Doe"
                              "Smith")])
    (string-append "Hello, " given " " surname)))
```

```
> (greet "John")
"Hello, John Doe"
> (greet "Adam")
"Hello, Adam Smith"
```

4.4.3 Declaring Keyword Arguments

A `lambda` form can declare an argument to be passed by keyword, instead of position. Keyword arguments can be mixed with by-position arguments, and default-value expressions can be supplied for either kind of argument:

§4.3.2 "Keyword Arguments" introduces function calls with keywords.

```
(lambda gen-formals
  body ...+)

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

An argument specified as *arg-keyword arg-id* is supplied by an application using the same *arg-keyword*. The position of the keyword–identifier pair in the argument list does not matter for matching with arguments in an application, because it will be matched to an argument value by keyword instead of by position.

```
(define greet
  (lambda (given #:last surname)
    (string-append "Hello, " given " " surname)))

> (greet "John" #:last "Smith")
"Hello, John Smith"
> (greet #:last "Doe" "John")
"Hello, John Doe"
```

An *arg-keyword* `[`*arg-id default-expr*`]` argument specifies a keyword-based argument with a default value.

Examples:

```
(define greet
  (lambda (#:hi [hi "Hello"] given #:last [surname "Smith"])
    (string-append hi ", " given " " surname)))

> (greet "John")
"Hello, John Smith"
```

```
> (greet "Karl" #:last "Marx")
"Hello, Karl Marx"
> (greet "John" #:hi "Howdy")
"Howdy, John Smith"
> (greet "Karl" #:last "Marx" #:hi "Guten Tag")
"Guten Tag, Karl Marx"
```

The `lambda` form does not directly support the creation of a function that accepts "rest" keywords. To construct a function that accepts all keyword arguments, use `make-keyword-procedure`. The function supplied to `make-keyword-procedure` receives keyword arguments through parallel lists in the first two (by-position) arguments, and then all by-position arguments from an application as the remaining by-position arguments.

§4.3.3 "The `apply` Function" introduces `keyword-apply`.

Examples:

```
(define (trace-wrap f)
  (make-keyword-procedure
   (lambda (kws kw-args . rest)
     (printf "Called with ~s ~s ~s\n" kws kw-args rest)
     (keyword-apply f kws kw-args rest))))
```

```
> ((trace-wrap greet) "John" #:hi "Howdy")
Called with (#:hi) ("Howdy") ("John")
"Howdy, John Smith"
```

4.4.4 Arity-Sensitive Functions: `case-lambda`

§2.6 "Procedure Expressions: `lambda` and `case-lambda`" in §"**Reference**: PLT Scheme" provides more on function expressions.

The `case-lambda` form creates a function that can have completely different behaviors depending on the number of arguments that are supplied. A case-lambda expression has the form

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

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

where each `[formals body ...+]` is analogous to `(lambda formals body ...+)`. Applying a function produced by `case-lambda` is like applying a `lambda` for the first case that matches the number of given arguments.

Examples:

```
(define greet
  (case-lambda
    [(name) (string-append "Hello, " name)]
    [(given surname) (string-append "Hello, " given " " surname)]))

> (greet "John")
"Hello, John"
> (greet "John" "Smith")
"Hello, John Smith"
> (greet)
procedure greet: no clause matching 0 arguments
```

A `case-lambda` function cannot directly support optional or keyword arguments.

4.5 Definitions: `define`

A basic definition has the form

```
(define id expr)
```

in which case *id* is bound to the result of *expr*.

Examples:

```
(define salutation (list-ref '("Hi" "Hello") (random 2)))

> salutation
"Hi"
```

4.5.1 Function Shorthand

The `define` form also supports a shorthand for function definitions:

```
(define (id arg ...) body ...+)
```

which is a shorthand for

```
(define id (lambda (arg ...) body ...+))
```

Examples:

```
(define (greet name)
  (string-append salutation ", " name))

> (greet "John")
"Hi, John"

(define (greet first [surname "Smith"] #:hi [hi salutation])
  (string-append hi ", " first " " surname))

> (greet "John")
"Hi, John Smith"
> (greet "John" #:hi "Hey")
"Hey, John Smith"
> (greet "John" "Doe")
"Hi, John Doe"
```

The function shorthand via `define` also supports a "rest" argument (i.e., a final argument to collect extra arguments in a list):

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

which is a shorthand

```
(define id (lambda (arg ... . rest-id) body ...+))
```

Examples:

```
(define (avg . l)
  (/ (apply + l) (length l)))

> (avg 1 2 3)
2
```

4.5.2 Curried Function Shorthand

Consider the following `make-add-suffix` function that takes a string and returns another function that takes a string:

```
(define make-add-suffix
  (lambda (s2)
    (lambda (s) (string-append s s2))))
```

Although it's not common, result of `make-add-suffix` could be called directly, like this:

```
> ((make-add-suffix "!") "hello")
"hello!"
```

In a sense, `make-add-suffix` is a function takes two arguments, but it takes them one at a time. A function that takes some of its arguments and returns a function to consume more is sometimes called a *curried function*.

Using the function-shorthand form of `define`, `make-add-suffix` can be written equivalently as

```
(define (make-add-suffix s2)
  (lambda (s) (string-append s s2)))
```

This shorthand reflects the shape of the function call `(make-add-suffix "!")`. The `define` form further supports a shorthand for defining curried functions that reflects nested function calls:

```
(define ((make-add-suffix s2) s)
  (string-append s s2))

> ((make-add-suffix "!") "hello")
"hello!"

(define louder (make-add-suffix "!"))
(define less-sure (make-add-suffix "?"))

> (less-sure "really")
"really?"
> (louder "really")
"really!"
```

The full syntax of the function shorthand for `define` is as follows:

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

head = id
     | (head args)

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

The expansion of this shorthand has one nested `lambda` form for each *head* in the definition, where the innermost *head* corresponds to the outermost `lambda`.

4.5.3 Multiple Values and `define-values`

A Scheme expression normally produces a single result, but some expressions can produce multiple results. For example, `quotient` and `remainder` each produce a single value, but `quotient/remainder` produces the same two values at once:

```
> (quotient 13 3)
4
> (remainder 13 3)
1
> (quotient/remainder 13 3)
4
1
```

As shown above, the REPL prints each result value on its own line.

Multiple-valued functions can be implemented in terms of the `values` function, which takes any number of values and returns them as the results:

```
> (values 1 2 3)
1
2
3

(define (split-name name)
  (let ([parts (regexp-split " " name)])
    (if (= (length parts) 2)
        (values (list-ref parts 0) (list-ref parts 1))
        (error "not a <first> <last> name"))))

> (split-name "Adam Smith")
"Adam"
"Smith"
```

The `define-values` form binds multiple identifiers at once to multiple results produced from a single expression:

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

The number of results produced by the *expr* must match the number of *id*s.

Examples:

```
(define-values (given surname) (split-name "Adam Smith"))
```

```
> given
"Adam"
> surname
"Smith"
```

A `define` form (that is not a function shorthand) is equivalent to a `define-values` form with a single *id*.

§2.12 "Definitions: `define`, `define-syntax`, ..." in §"**Reference**: PLT Scheme" provides more on definitions.

4.5.4 Internal Definitions

When the grammar for a syntactic form specifies *body*, then the corresponding form can be either a definition or an expression. A definition as a *body* is an *internal definition*.

All internal definitions in a *body* sequence must appear before any expression, and the last *body* must be an expression.

For example, the syntax of `lambda` is

```
(lambda gen-formals
  body ...+)
```

so the following are valid instances of the grammar:

```
(lambda (f)                 ; no definitions
  (printf "running\n")
  (f 0))

(lambda (f)                 ; one definition
  (define (log-it what)
    (printf "~a\n"))
  (log-it "running")
  (f 0)
  (log-it "done"))

(lambda (f n)               ; two definitions
  (define (call n)
    (if (zero? n)
        (log-it "done")
        (begin
          (log-it "running")
          (f 0)
          (call (- n 1)))))
  (define (log-it what)
```

```
    (printf "~a\n"))
  (call f n))
```

Internal definitions in a particular *body* sequence are mutually recursive; that is, any definition can refer to any other definition—as long as the reference isn't actually evaluated before its definition takes place. If a definition is referenced too early, the result is a special value `#<undefined>`.

Examples:

```
(define (weird)
  (define x x)
  x)

> (weird)
#<undefined>
```

A sequence of internal definitions using just `define` is easily translated to an equivalent `letrec` form (as introduced in the next section). However, other definition forms can appear as a *body*, including `define-values`, `define-struct` (see §5 "Programmer-Defined Datatypes") or `define-syntax` (see §17 "Macros").

§1.2.3.7 "Internal Definitions" in §"**Reference**: PLT Scheme" documents the fine points of internal definitions.

4.6 Local Binding

Although internal `define`s can be used for local binding, Scheme provides three forms that give the programmer more control over bindings: `let`, `let*`, and `letrec`.

4.6.1 Parallel Binding: `let`

A `let` form binds a set of identifiers, each to the result of some expression, for use in the `let` body:

§2.7 "Local Binding: `let`, `let*`, `letrec`, ..." in §"**Reference**: PLT Scheme" also documents `let`.

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

The *id*s are bound "in parallel." That is, no *id* is bound in the right-hand side *expr* for any *id*, but all are available in the *body*. The *id*s must be different from each other.

Examples:

```
> (let ([me "Bob"])
    me)
"Bob"
> (let ([me "Bob"]
```

```
        [myself "Robert"]
        [I "Bobby"])
    (list me myself I))
("Bob" "Robert" "Bobby")
> (let ([me "Bob"]
        [me "Robert"])
    me)
eval:7:0: let: duplicate identifier at: me in: (let ((me
"Bob") (me "Robert")) me)
```

The fact that an *id*'s *expr* does not see its own binding is often useful for wrappers that must refer back to the old value:

```
> (let ([+ (lambda (x y)
             (if (string? x)
                 (string-append x y)
                 (+ x y)))]) ; use original +
    (list (+ 1 2)
          (+ "see" "saw")))
(3 "seesaw")
```

Occasionally, the parallel nature of `let` bindings is convenient for swapping or rearranging a set of bindings:

```
> (let ([me "Tarzan"]
        [you "Jane"])
    (let ([me you]
          [you me])
      (list me you)))
("Jane" "Tarzan")
```

The characterization of `let` bindings as "parallel" is not meant to imply concurrent evaluation. The *expr*s are evaluated in order, even though the bindings are delayed until all *expr*s are evaluated.

4.6.2 Sequential Binding: `let*`

The syntax of `let*` is the same as `let`:

§2.7 "Local Binding: `let`, `let*`, `letrec`, ..." in §"**Reference**: PLT Scheme" also documents `let*`.

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

The difference is that each *id* is available for use in later *expr*s, as well as in the *body*. Furthermore, the *id*s need not be distinct, and the most recent binding is the visible one.

Examples:

```
> (let* ([x (list "Borroughs")]
         [y (cons "Rice" x)]
         [z (cons "Edgar" y)])
    (list x y z))
(("Borroughs") ("Rice" "Borroughs") ("Edgar" "Rice" "Borroughs"))
> (let* ([name (list "Borroughs")]
         [name (cons "Rice" name)]
         [name (cons "Edgar" name)])
    name)
("Edgar" "Rice" "Borroughs")
```

In other words, a `let*` form is equivalent to nested `let` forms, each with a single binding:

```
> (let ([name (list "Borroughs")])
    (let ([name (cons "Rice" name)])
      (let ([name (cons "Edgar" name)])
        name)))
("Edgar" "Rice" "Borroughs")
```

4.6.3 Recursive Binding: `letrec`

The syntax of `letrec` is also the same as `let`:

§2.7 "Local Binding: `let`, `let*`, `letrec`, ..." in §"**Reference**: PLT Scheme" also documents `letrec`.

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

While `let` makes its bindings available only in the *body*s, and `let*` makes its bindings available to any later binding *expr*, `letrec` makes its bindings available to all other *expr*s—even earlier ones. In other words, `letrec` bindings are recursive.

The *expr*s in a `letrec` form are most often `lambda` forms for recursive and mutually recursive functions:

```
> (letrec ([swing
            (lambda (t)
              (if (eq? (car t) 'tarzan)
                  (cons 'vine
                        (cons 'tarzan (cddr t)))
                  (cons (car t)
                        (swing (cdr t)))))])
    (swing '(vine tarzan vine vine)))
(vine vine tarzan vine)
```

```
> (letrec ([tarzan-in-tree?
            (lambda (name path)
              (or (equal? name "tarzan")
                  (and (directory-exists? path)
                       (tarzan-in-directory? path))))]
           [tarzan-in-directory?
            (lambda (dir)
              (ormap (lambda (elem)
                       (tarzan-in-tree? (path-element->string elem)
                                        (build-path dir elem)))
                     (directory-list dir)))])
    (tarzan-in-tree? "tmp" (find-system-path 'temp-dir)))
#f
```

While the *expr*s of a `letrec` form are typically `lambda` expressions, they can be any expression. The expressions are evaluated in order, and after each value is obtained, it is immediately associated with its corresponding *id*. If an *id* is referenced before its value is ready, the result is `#<undefined>`, as just as for internal definitions.

```
> (letrec ([quicksand quicksand])
    quicksand)
#<undefined>
```

4.6.4 Named `let`

A named `let` is an iteration and recursion form. It uses the same syntactic keyword `let` as for local binding, but an identifier after the `let` (instead of an immediate open parenthesis) triggers a different parsing.

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

A named `let` form is equivalent to

```
(letrec ([proc-id (lambda (arg-id ...)
                    body ...+)])
  (proc-id init-expr ...))
```

That is, a named `let` binds a function identifier that is visible only in the function's body, and it implicitly calls the function with the values of some initial expressions.

Examples:

```
(define (duplicate pos lst)
  (let dup ([i 0]
            [lst lst])
    (cond
     [(= i pos) (cons (car lst) lst)]
     [else (cons (car lst) (dup (+ i 1) (cdr lst)))])))

> (duplicate 1 (list "apple" "cheese burger!" "banana"))
("apple" "cheese burger!" "cheese burger!" "banana")
```

4.6.5 Multiple Values: `let-values`, `let*-values`, `letrec-values`

In the same way that `define-values` binds multiple results in a definition (see §4.5.3 "Multiple Values and `define-values`"), `let-values`, `let*-values`, and `letrec-values` bind multiple results locally.

§2.7 "Local Binding: `let`, `let*`, `letrec`, ..." in §"**Reference**: PLT Scheme" also documents multiple-value binding forms.

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

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

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

Each *expr* must produce as many values as corresponding *id*s. The binding rules are the same for the forms without `-values` forms: the *id*s of `let-values` are bound only in the *body*s, the *id*s of `let*-valuess` are bound in *expr*s of later clauses, and the *id*s of `letrec-values` are bound for all *expr*s.

Examples:

```
> (let-values ([(q r) (quotient/remainder 14 3)])
    (list q r))
(4 2)
```

4.7 Conditionals

Most functions used for branching, such as `<` and `string?`, produce either `#t` or `#f`. Scheme's branching forms, however, treat any value other than `#f` as true. We we say a *true value* to mean any value other than `#f`.

This convention for "true value" meshes well with protocols where `#f` can serve as failure or to indicate that an optional value is not supplied. (Beware of overusing this trick, and remember that an exception is usually a better mechanism to report failure.)

For example, the `member` function serves double duty; it can be used to find the tail of a list that starts with a particular item, or it can be used to simply check whether an item is present in a list:

```
> (member "Groucho" '("Harpo" "Zeppo"))
#f
> (member "Groucho" '("Harpo" "Groucho" "Zeppo"))
("Groucho" "Zeppo")
> (if (member "Groucho" '("Harpo" "Zeppo"))
      'yep
      'nope)
nope
> (if (member "Groucho" '("Harpo" "Groucho" "Zeppo"))
      'yep
      'nope)
yep
```

4.7.1 Simple Branching: `if`

In an `if` form,

§2.10 "Conditionals: `if`, `cond`, `and`, and `or`" in §"**Reference**: PLT Scheme" also documents `if`.

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

the *`test-expr`* is always evaluated. If it produces any value other than `#f`, then *`then-expr`* is evaluated. Otherwise, *`else-expr`* is evaluated.

An `if` form must have both an *`then-expr`* and an *`else-expr`*; the latter is not optional. To perform (or skip) side-effects based on a *`test-expr`*, use `when` or `unless`, which we describe later in §4.8 "Sequencing".

4.7.2 Combining Tests: `and` and `or`

Scheme's `and` and `or` are syntactic forms, rather than functions. Unlike a function, the `and` and `or` forms can skip evaluation of later expressions if an earlier one determines the answer.

§2.10 "Conditionals: `if`, `cond`, `and`, and `or`" in §"**Reference**: PLT Scheme" also documents `and` and `or`.

```
(and expr ...)
```

An `or` form produces `#f` if any of its `expr`s produces `#f`. Otherwise, it produces the value of its last *expr*. As a special case, `(and)` produces `#t`.

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

The `and` form produces `#f` if any of its *expr*s produces `#f`. Otherwise, it produces the first non-`#f` value from its `expr`s. As a special case, `(or)` produces `#f`.

Examples:

```
> (define (got-milk? lst)
    (and (not (null? lst))
         (or (eq? 'milk (car lst))
             (got-milk? (cdr lst))))) ; recurs only if needed
> (got-milk? '(apple banana))
#f
> (got-milk? '(apple milk banana))
#t
```

If evaluation reaches the last *expr* of an `and` or `or` form, then the *expr*'s value directly determines the `and` or `or` result. Therefore, the last `expr` is in tail position, which means that the above `got-milk?` function runs in constant space.

§2.3.3 "Tail Recursion" introduces tail calls and tail positions.

4.7.3 Chaining Tests: `cond`

The `cond` form chains a series of tests to select a result expression. To a first approximation, the syntax of `cond` is as follows:

§2.10 "Conditionals: `if`, `cond`, `and`, and `or`" in §"**Reference**: PLT Scheme" also documents `cond`.

```
(cond [test-expr expr ...+]
      ...)
```

Each *test-expr* is evaluated in order. If it produces `#f`, the corresponding *expr*s are ignored, and evaluation proceeds to the next *test-expr*. As soon as a *test-expr* produces a true value, its *text-expr*s are evaluated to produce the result for the `cond` form, and no further *test-expr*s are evaluated.

The last *test-expr* in a `cond` can be replaced by `else`. In terms of evaluation, `else` serves as a synonym for `#t`, but it clarifies that the last clause is meant to catch all remaining cases. If `else` is not used, then it is possible that no *test-expr*s produce a true value; in that case, the result of the `cond` expression is `#<void>`.

Examples:

```
> (cond
    [(= 2 3) (error "wrong!")]
    [(= 2 2) 'ok])
ok
> (cond
    [(= 2 3) (error "wrong!")])
> (cond
    [(= 2 3) (error "wrong!")]
    [else 'ok])
ok

(define (got-milk? lst)
  (cond
    [(null? lst) #f]
    [(eq? 'milk (car lst)) #t]
    [else (got-milk? (cdr lst))]))

> (got-milk? '(apple banana))
#f
> (got-milk? '(apple milk banana))
#t
```

The full syntax of `cond` includes two more kinds of clauses:

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

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

The `=>` variant captures the true result of its *test-expr* and passes it to the result of the *proc-expr*, which must be a function of one argument.

Examples:

```
> (define (after-groucho lst)
    (cond
      [(member "Groucho" lst) => cdr]
      [else (error "not there")]))
> (after-groucho '("Harpo" "Groucho" "Zeppo"))
("Zeppo")
> (after-groucho '("Harpo" "Zeppo"))
not there
```

A clause that includes only a *test-expr* is rarely used. It captures the true result of the *test-expr*, and simply returns the result for the whole `cond` expression.

4.8 Sequencing

Scheme programmers prefer to write programs with as few side-effects as possible, since purely functional code is more easily tested and composed into larger programs. Interaction with the external environment, however, requires sequencing, such as when writing to a display, opening a graphical window, or manipulating a file on disk.

4.8.1 Effects Before: `begin`

A `begin` expression sequences expressions:

§2.13 "Sequencing: `begin`, `begin0`, and `begin-for-syntax`" in §"**Reference**: PLT Scheme" also documents `begin`.

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

The *expr*s are evaluated in order, and the result of all but the last *expr* is ignored. The result from the last *expr* is the result of the `begin` form, and it is in tail position with respect to the `begin` form.

Examples:

```
(define (print-triangle height)
  (if (zero? height)
      (void)
      (begin
       (display (make-string height #\*))
       (newline)
       (print-triangle (sub1 height)))))

> (print-triangle 4)
****
```

```
***
**
*
```

Many forms, such as `lambda` or `cond` support a sequence of expressions even without a `begin`. Such positions are sometimes said to have an *implicit begin*.

Examples:
```
(define (print-triangle height)
  (cond
    [(not (positive? height))
     (display (make-string height #\*))
     (newline)
     (print-triangle (sub1 height))]))

> (print-triangle 4)
```

The `begin` form is special at the top level, at module level, or as a `body` after only internal definitions. In those positions, instead of forming an expression, the content of `begin` is spliced into the surrounding context.

Examples:
```
> (let ([curly 0])
   (begin
    (define moe (+ 1 curly))
    (define larry (+ 1 moe)))
   (list larry curly moe))
(2 0 1)
```

This splicing behavior is mainly useful for macros, as we discuss later in §17 “Macros”.

4.8.2 Effects After: `begin0`

A `begin0` expression has the same syntax as a `begin` expression:

§2.13 “Sequencing: `begin`, `begin0`, and `begin-for-syntax`” in §“**Reference**: PLT Scheme” also documents `begin0`.

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

The difference is that `begin0` returns the result of the first `expr`, instead of the result of the last `expr`. The `begin0` form is useful for implementing side-effects that happen after a computation, especially in the case where the computation produces an unknown number of results.

Examples:

```
(define (log-times thunk)
  (printf "Start: ~s\n" (current-inexact-milliseconds))
  (begin0
    (thunk)
    (printf "End..: ~s\n" (current-inexact-milliseconds))))
```

```
> (log-times (lambda () (sleep 0.1) 0))
Start: 1209543041873.994
End..: 1209543041974.822
0
> (log-times (lambda () (values 1 2)))
Start: 1209543041975.047
End..: 1209543041975.063
1
2
```

4.8.3 Effects If...: `when` and `unless`

§2.14 "Guarded Evaluation: `when` and `unless`" in §"**Reference**: PLT Scheme" also documents `when` and `unless`.

The `when` form combines an `if`-style conditional with sequencing for the "then" clause and no "else" clause:

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

If *test-expr* produces a true value, then all of the *then-expr*s are evaluated. Otherwise, no *then-expr*s are evaluated. The result is `#<void>` in any case.

The `unless` form is similar:

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

The difference is that the *test-expr* result is inverted: the *then-expr*s are evaluated only if the *test-expr* result is `#f`.

Examples:

```
(define (enumerate lst)
 (if (null? (cdr lst))
     (printf "~a.\n" (car lst))
     (begin
       (printf "~a, " (car lst))
```

```
        (when (null? (cdr (cdr lst)))
          (printf "and "))
        (enumerate (cdr lst)))))

> (enumerate '("Larry" "Curly" "Moe"))
Larry, Curly, and Moe.

(define (print-triangle height)
  (unless (zero? height)
    (display (make-string height #\*))
    (newline)
    (print-triangle (sub1 height))))

> (print-triangle 4)
****
***
**
*
```

4.9 Assignment: `set!`

§2.15 "Assignment: `set!` and `set!-values`" in §"**Reference**: PLT Scheme" also documents `set!`.

Assign to a variable using `set!`:

```
(set! id expr)
```

A `set!` expression evaluates *expr* and changes *id* (which must be bound in the enclosing environment) to the resulting value. The result of the `set!` expression itself is `#<void>`.

Examples:

```
(define greeted null)

(define (greet name)
  (set! greeted (cons name greeted))
  (string-append "Hello, " name))

> (greet "Athos")
"Hello, Athos"
> (greet "Porthos")
"Hello, Porthos"
> (greet "Aramis")
"Hello, Aramis"
> greeted
("Aramis" "Porthos" "Athos")
```

```
(define (make-running-total)
  (let ([n 0])
    (lambda ()
      (set! n (+ n 1))
      n)))
(define win (make-running-total))
(define lose (make-running-total))
```

```
> (win)
1
> (win)
2
> (lose)
1
> (win)
3
```

4.9.1 Guidelines for Using Assignment

Although using `set!` is sometimes appropriate, Scheme style generally discourages the use of `set!`. The following guidelines may help explain when using `set!` is appropriate.

- As in any modern language, assigning to shared identifier is no substitute for passing an argument to a procedure or getting its result.

Really awful example:

```
(define name "unknown")
(define result "unknown")
(define (greet)
  (set! result (string-append "Hello, " name)))   > (set! name "John")
> (greet)
> result
"Hello, John"
```

Ok example:

```
(define (greet name)
  (string-append "Hello, " name))   > (greet "John")
"Hello, John"
> (greet "Anna")
"Hello, Anna"
```

- A sequence of assignments to a local variable is far inferior to nested bindings.

Bad example:

```
> (let ([tree 0])
```

```
    (set! tree (list tree 1 tree))
    (set! tree (list tree 2 tree))
    (set! tree (list tree 3 tree))
    tree)
(((0 1 0) 2 (0 1 0)) 3 ((0 1 0) 2 (0 1 0)))
```

Ok example:

```
> (let* ([tree 0]
         [tree (list tree 1 tree)]
         [tree (list tree 2 tree)]
         [tree (list tree 3 tree)])
    tree)
(((0 1 0) 2 (0 1 0)) 3 ((0 1 0) 2 (0 1 0)))
```

- Using assignment to accumulate results from an iteration is bad style. Accumulating through a loop argument is better.

Somewhat bad example:

```
(define (sum lst)
  (let ([s 0])
    (for-each (lambda (i) (set! s (+ i s)))
              lst)
    s))   > (sum '(1 2 3))
6
```

Ok example:

```
(define (sum lst)
  (let loop ([lst lst][s 0])
    (if (null? lst)
        s
        (loop (cdr lst) (+ s (car lst))))))   > (sum '(1 2 3))
6
```

Better (use an existing function) example:

```
(define (sum lst)
  (apply + lst))   > (sum '(1 2 3))
6
```

Good (a general approach) example:

```
(define (sum lst)
  (for/fold ([s 0])
            ([i (in-list lst)])
    (+ s i)))   > (sum '(1 2 3))
6
```

- For cases where stateful objects are necessary or appropriate, then implementing the object's state with `set!` is fine.

Ok example:

```
(define next-number!
  (let ([n 0])
    (lambda ()
      (set! n (add1 n))
      n)))   > (next-number!)
1
> (next-number!)
2
> (next-number!)
3
```

All else being equal, a program that uses no assignments or mutation is always preferable to one that uses assignments or mutation. While side effects are to be avoided, however, they should be used if the resulting code is significantly more readable or if it implements a significantly better algorithm.

The use of mutable values, such as vectors and hash tables, raises fewer suspicions about the style of a program than using `set!` directly. Nevertheless, simply replacing `set!`s in a program with a `vector-set!`s obviously does not improve the style of the program.

4.9.2 Multiple Values: `set!-values`

§2.15 "Assignment: `set!` and `set!-values`" in §"**Reference**: PLT Scheme" also documents `set!-values`.

The `set!-values` form assigns to multiple variables at once, given an expression that produces an appropriate number of values:

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

This form is equivalent to using `let-values` to receive multiple results from *expr*, and then assigning the results individually to the *id*s using `set!`.

Examples:

```
(define game
  (let ([w 0]
        [l 0])
    (lambda (win?)
      (if win?
          (set! w (+ w 1))
          (set! l (+ l 1)))
      (begin0
        (values w l)
        ; swap sides...
```

```
        (set!-values (w l) (values l w))))))

> (game #t)
1
0
> (game #t)
1
1
> (game #f)
1
2
```

4.10 Quoting: `quote` and `'`

§2.1 "Literals: `quote` and `#%datum`" in §"**Reference**: PLT Scheme" also documents `quote`.

The `quote` form produces a constant:

```
(quote datum)
```

The syntax of a `datum` is technically specified as anything that the `read` function parses as a single element. The value of the `quote` form is the same value that `read` would produce given *`datum`*.

To a good approximation, the resulting value is such that *`datum`* is the value's printed representation. Thus, it can be a symbol, a boolean, a number, a (character or byte) string, a character, a keyword, an empty list, a pair (or list) containing more such values, a vector containing more such values, a hash table containing more such values, or a box containing another such value.

Examples:

```
> (quote apple)
apple
> (quote #t)
#t
> (quote 42)
42
> (quote "hello")
"hello"
> (quote ())
()
> (quote ((1 2 3) #("z" x) . the-end))
((1 2 3) #("z" x) . the-end)
> (quote (1 2 . (3)))
(1 2 3)
```

As the last example above shows, the *datum* does not have to be the normalized printed form of a value. A *datum* cannot be a printed representation that starts with `#<`, however, so it cannot be `#<void>`, `#<undefined>`, or a procedure.

The `quote` form is rarely used for a *datum* that is a boolean, number, or string by itself, since the printed forms of those values can already be used as constants. The `quote` form is more typically used for symbols and lists, which have other meanings (identifiers, function calls, etc.) when not quoted.

An expression

```
'datum
```

is a shorthand for

```
(quote datum)
```

and this shorthand is almost always used instead of `quote`. The shorthand applies even within the *datum*, so it can produce a list containing `quote`.

§12.6.7 "Reading Quotes" in §"**Reference**: PLT Scheme" provides more on the `'` shorthand.

Examples:

```
> 'apple
apple
> '"hello"
"hello"
> '(1 2 3)
(1 2 3)
> (display '(you can 'me))
(you can (quote me))
```

4.11 Quasiquoting: `quasiquote` and `` ` ``

The `quasiquote` form is similar to `quote`:

```
(quasiquote datum)
```

However, for each `(unquote expr)` that appears within the *datum*, the *expr* is evaluated to produce a value that takes the place of the `unquote` sub-form.

Examples:

```
> (quasiquote (1 2 (unquote (+ 1 2)) (unquote (- 5 1))))
(1 2 3 4)
```

The `unquote-splicing` form is similar to `unquote`, but its *expr* must produce a list, and the `unquote-splicing` form must appear in a context that produces either a list or a vector. As the name suggests, the resulting list is spliced into the context of its use.

Examples:
```
> (quasiquote (1 2 (unquote-splicing (list (+ 1 2) (- 5 1))) 5))
(1 2 3 4 5)
```

If a `quasiquote` form appears within an enclosing `quasiquote` form, then the inner `quasiquote` effectively cancels one layer of `unquote` and `unquote-splicing` forms, so that a second `unquote` or `unquote-splicing` is needed.

Examples:
```
> (quasiquote (1 2 (quasiquote (unquote (+ 1 2)
                                 (unquote (unquote (- 5 1)))))))
(1 2 (quasiquote (unquote (+ 1 2)) (unquote 4)))
```

The evaluation above will not actually print as shown. Instead, the shorthand form of `quasiquote` and `unquote` will be used: `` ` `` (i.e., a backquote) and `,` (i.e., a comma). The same shorthands can be used in expressions:

Examples:
```
> `(1 2 `(,(+ 1 2) ,,(- 5 1)))
(1 2 `(,(+ 1 2) ,4))
```

The shorthand for of `unquote-splicing` is `,@`:

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

4.12 Simple Dispatch: `case`

The `case` form dispatches to a clause by matching the result of an expression to the values for the clause:

```
(case expr
  [(datum ...+) expr ...+]
  ...)
```

Each *datum* will be compared to the result of the first *expr* using eqv?. Since eqv? doesn't work on many kinds of values, notably symbols and lists, each *datum* is typically a number, symbol, or boolean.

Multiple *datum*s can be supplied for each clause, and the corresponding *expr* is evaluated of any of the *datum*s match.

Examples:
```
> (let ([v (random 6)])
    (printf "~a\n" v)
    (case v
      [(0) 'zero]
      [(1) 'one]
      [(2) 'two]
      [(3 4 5) 'many]))
0
zero
```

The last clause of a case form can use else, just like cond:

Examples:
```
> (case (random 6)
    [(0) 'zero]
    [(1) 'one]
    [(2) 'two]
    [else 'many])
one
```

For more general pattern matching, use match, which is introduced in §12 "Pattern Matching".

5 Programmer-Defined Datatypes

§4 "Structures" in §"**Reference**: PLT Scheme" also documents structure types.

New datatypes are normally created with the `define-struct` form, which is the topic of this chapter. The class-based object system, which we defer to §13 "Classes and Objects", offers an alternate mechanism for creating new datatypes, but even classes and objects are implemented in terms of structure types.

5.1 Simple Structure Types: `define-struct`

§4.1 "Defining Structure Types: `define-struct`" in §"**Reference**: PLT Scheme" also documents `define-struct`.

To a first approximation, the syntax of `define-struct` is

```
(define-struct struct-id (field-id ...))
```

A `define-struct` declaration binds *`struct-id`*, but only to static information about the structure type that cannot be used directly:

```
(define-struct posn (x y))

> posn
eval:2:0: posn: identifier for static struct-type
information cannot be used as an expression in: posn
```

We show two uses of the *`struct-id`* binding below in §5.2 "Copying and Update" and §5.3 "Structure Subtypes".

Meanwhile, in addition to defining *`struct-id`*, `define-struct` also defines a number of identifiers that are built from *`struct-id`* and the *`field-id`*s:

- `make-`*`struct-id`* : a *constructor* function that takes as many arguments as the number of *`field-id`*s, and returns an instance of the structure type.

Examples:

```
> (make-posn 1 2)
#<posn>
```

- *`struct-id`*`?` : a *predicate* function that takes a single argument and returns `#t` if it is an instance of the structure type, `#f` otherwise.

Examples:

```
> (posn? 3)
#f
> (posn? (make-posn 1 2))
```

```
#t
```

- *struct-id-field-id* : for each *field-id*, an *accessor* that extracts the value of the corresponding field from an instance of the structure type.

Examples:
```
> (posn-x (make-posn 1 2))
1
> (posn-y (make-posn 1 2))
2
```

- struct:*struct-id* : a *structure type descriptor*, which is a value that represents the structure type as a first-class value (with #:super, as discussed later in §5.7 “More Structure Type Options”).

A define-struct form places no constraints on the kinds of values that can appear for fields in an instance of the structure type. For example, (make-posn "apple" #f) produces an instance of posn, even though "apple" and #f are not valid coordinates for the obvious uses of posn instances. Enforcing constraints on field values, such as requiring them to be numbers, is normally the job of a contract, as discussed later in §7 “Contracts”.

5.2 Copying and Update

The struct-copy form clones a structure and optionally updates specified fields in the clone. This process is sometimes called a *functional update*, because the result is a structure with updated field values. but the original structure is not modified.

```
(struct-copy struct-id struct-expr [field-id expr] ...)
```

The *struct-id* that appears after struct-copy must be a structure type name bound by define-struct (i.e., the name that cannot be used directly as an expression). The *struct-expr* must produce an instance of the structure type. The result is a new instance of the structure tpe that is like the old one, except that the field indicated by each *field-id* gets the value of the corresponding *expr*.

Examples:
```
> (define p1 (make-posn 1 2))
> (define p2 (struct-copy posn p1 [x 3]))
> (list (posn-x p2) (posn-y p2))
(3 2)
> (list (posn-x p1) (posn-x p2))
(1 3)
```

5.3 Structure Subtypes

An extended form of `define-struct` can be used to define a *structure subtype*, which is a structure type that extends an existing structure type:

```
(define-struct (struct-id super-id) (field-id ...))
```

The *super-id* must be a structure type name bound by `define-struct` (i.e., the name that cannot be used directly as an expression).

Examples:

```
(define-struct posn (x y))
(define-struct (3d-posn posn) (z))
```

A structure subtype inherits the fields of its supertype, and the subtype constructor accepts the values for the subtype fields after values for the supertype fields. An instance of a structure subtype can be used with the predicate and accessors of the supertype.

Examples:

```
> (define p (make-3d-posn 1 2 3))
> p
#<3d-posn>
> (posn? p)
#t
> (posn-x p)
1
> (3d-posn-z p)
3
```

5.4 Opaque versus Transparent Stucture Types

With a structure type definition like

```
(define-struct posn (x y))
```

an instance of the structure type prints in a way that does not show any information about the fields values. That is, structure types by default are *opaque*. If the accessors and mutators of a structure type are kept private to a module, then no other module can rely on the representation of the type's instances.

To make a structure type *transparent*, use the `#:transparent` keyword after the field-name sequence:

```
(define-struct posn (x y)
                #:transparent)

> (make-posn 1 2)
#(struct:posn 1 2)
```

An instance of a transparent structure type prints like a vector, and it shows the content of the structure's fields. A transparent structure type also allows reflective operations, such as `struct?` and `struct-info`, to be used on its instances (see §16 "Reflection and Dynamic Evaluation").

Structure types are opaque by default, because opaque structure instances provide more encapsulation guarantees. That is, a library can use an opaque structure to encapsulate data, and clients of the library cannot manipulate the data in the structure except as allowed by the library.

5.5 Structure Type Generativity

Each time that a `define-struct` form is evaluated, it generates a structure type that is distinct from all existing structure types, even if some other structure type has the same name and fields.

This generativity is useful for enforcing abstractions and implementing programs such as interpreters, but beware of placing a `define-struct` form in positions that are evaluated multiple times.

Examples:

```
(define (add-bigger-fish lst)
  (define-struct fish (size) #:transparent) ; new every time
  (cond
   [(null? lst) (list (make-fish 1))]
   [else (cons (make-fish (* 2 (fish-size (car lst))))
               lst)]))

> (add-bigger-fish null)
(#(struct:fish 1))
> (add-bigger-fish (add-bigger-fish null))
fish-size: expects args of type <struct:fish>; given
instance of a different <struct:fish>

(define-struct fish (size) #:transparent)
(define (add-bigger-fish lst)
  (cond
   [(null? lst) (list (make-fish 1))]
   [else (cons (make-fish (* 2 (fish-size (car lst))))
```

```
                lst)]))

> (add-bigger-fish (add-bigger-fish null))
(#(struct:fish 2) #(struct:fish 1))
```

5.6 Prefab Stucture Types

Although a transparent structure type prints in a way that shows its content, the printed form of the structure cannot be used in an expression to get the structure back, unlike the printed form of a number, string, symbol, or list.

A *prefab* ("previously fabricated") structure type is a built-in type that is known to the Scheme printer and expression reader. Infinitely many such types exist, and they are indexed by name, field count, supertype, and other such details. The printed form of a prefab structure is similar to a vector, but it starts `#s` instead of just `#`, and the first element in the printed form is the prefab structure type's name.

The following examples show instances of the `sprout` prefab structure type that has one field. The first instance has a field value `'bean`, and the second has field value `'alfalfa`:

```
> '#s(sprout bean)
#s(sprout bean)
> '#s(sprout alfalfa)
#s(sprout alfalfa)
```

Like numbers and strings, prefab structures are "self-quoting," so the quotes above are optional:

```
> #s(sprout bean)
#s(sprout bean)
```

When you use the `#:prefab` keyword with `define-struct`, instead of generating a new structure type, you obtain bindings that work with the existing prefab structure type:

```
> (define lunch '#s(sprout bean))
> (define-struct sprout (kind) #:prefab)
> (sprout? lunch)
#t
> (sprout-kind lunch)
bean
> (make-sprout 'garlic)
#s(sprout garlic)
```

The field name `kind` above does not matter for finding the prefab structure type; only the name `sprout` and the number of fields matters. At the same time, the prefab structure type `sprout` with three fields is a different structure type than the one with a single field:

```
> (sprout? #s(sprout bean #f 17))
#f
> (define-struct sprout (kind yummy? count) #:prefab) ; redefine
> (sprout? #s(sprout bean #f 17))
#t
> (sprout? lunch)
#f
```

A prefab structure type can have another prefab structure type as its supertype, it can have mutable fields, and it can have auto fields. Variations in any of these dimensions correspond to different prefab structure types, and the printed form of the structure type's name encodes all of the relevant details.

```
> (define-struct building (rooms [location #:mutable]) #:prefab)
> (define-struct (house building) ([occupied #:auto]) #:prefab
    #:auto-value 'no)
> (make-house 5 'factory)
#s((house (1 no) building 2 #(1)) 5 factory no)
```

Every prefab structure type is transparent—but even less abstract than a transparent type, because instances can be created without any access to a particular structure-type declaration or existing examples. Overall, the different options for structure types offer a spectrum of possibilities from more abstract to more convenient:

- Opaque (the default) : Instances cannot be inspected or forged without access to the structure-type declaration. As discussed in the next section, constructor guards and properties can be attached to the structure type to further protect or to specialize the behavior of its instances.
- Transparent : Anyone can inspect or create an instance without access to the structure-type declaration, which means that the value printer can show the content of an instance. All instance creation passes through a constructor guard, however, so that the content of an instance can be controlled, and the behavior of instances can be specialized through properties. Since the structure type is generated by its definition, instances cannot be manufactured simply through the name of the structure type, and therefore cannot be generated automatically by the expression reader.
- Prefab : Anyone can inspect or create an instance at any time, without prior access to a structure-type declaration or an example instance. Consequently, the expression reader can manufacture instances directly. The instance cannot have a constructor guard or properties.

Since the expression reader can generate prefab instances, they are useful when convenient serialization is more important than abstraction. Opaque and transparent structures also can be serialized, however, if they are defined with `define-serializable-struct` as described in §8.4 "Datatypes and Serialization".

5.7 More Structure Type Options

The full syntax of `define-struct` supports many options, both at the structure-type level and at the level of individual fields:

```
(define-struct id-maybe-super (field ...)
               struct-option ...)

id-maybe-super = struct-id
               | (struct-id super-id)

         field = field-id
               | [field-id field-option ...]
```

A *struct-option* always starts with a keyword:

`#:mutable`

Causes all fields of the structure to be mutable, and introduces for each *field-id* a *mutator* `set-struct-id-field-id!` that sets the value of the corresponding field in an instance of the structure type.

Examples:

```
(define-struct dot (x y) #:mutable)
```

(define d (make-dot 1 2)) (dot-x d) (set-dot-x! d 10) (dot-x d)]

The `#:mutable` option can also be used as a *field-option*, in which case it makes an individual field mutable.

Examples:

```
(define-struct person (name [age #:mutable]))
(define friend (make-person "Barney" 5))
> (set-person-age! friend 6)
> (set-person-name! friend "Mary")
```

reference to undefined identifier: set-person-name!

`#:transparent`

Controls reflective access to structure instances, as discussed in a previous section, §5.4 “Opaque versus Transparent Stucture Types”.

`#:inspector` *inspector-expr*

Generalizes `#:transparent` to support more controlled access to reflective operations.

`#:prefab`

Accesses a built-in structure type, as discussed in a previous section, §5.6 “Prefab Stucture Types”.

`#:auto-value` *auto-expr*

Specifies a value to be used for all automatic fields in the structure type, where an automatic field is indicated by the `#:auto` field option. The constructor procedure does not accept arguments for automatic fields, and they are implicitly mutable.

Examples:

```
(define-struct posn (x y [z #:auto])
                #:transparent
                #:auto-value 0)
> (make-posn 1 2)
#(struct:posn 1 2 0)
```

`#:guard` *guard-expr*

Specifies a *constructor guard* procedure to be called whenever an instance of the structure type is created. The guard takes as many arguments as non-automatic fields in the structure type, and it should return the same number of values. The guard can raise an exception if one of the given arguments is unacceptable, or it can convert an argument.

Examples:

```
(define-struct thing (name)
                #:transparent
                #:guard (lambda (name type-name)
                          (cond
                            [(string? name) name]
                            [(number? name)
                             (number->string name)]
                            [else (error "bad name" name)])))
> (make-thing "apple")
#(struct:thing "apple")
> (make-thing 1/2)
#(struct:thing "1/2")
> (make-thing #f)
bad name #f
```

The guard is called even when subtype instances are created. In that case, only the fields accepted by the constructor are provided to the guard (but the subtype's guard gets both the original fields and fields added by the subtype).

Examples:

```
(define-struct (person thing) (age)
               #:transparent
               #:guard (lambda (name age type-name)
                         (if (negative? age)
                             (error "bad age" age)
                             (values name age))))
> (make-person "John" 10)
#(struct:person "John" 10)
> (make-person "Mary" -1)
bad age -1
> (make-person #f 10)
bad name #f
```

`#:property` *prop-expr val-expr*

Associates a *property* and value with the structure type. For example, the `prop:procedure` property allows a structure instance to be used as a function; the property value determines how a call is implemented when using the structure as a function.

Examples:

```
(define-struct greeter (name)
               #:property prop:procedure
                          (lambda (self other)
                            (string-append
                             "Hi " other
                             ", I'm " (greeter-name self))))
(define joe-greet (make-greeter "Joe"))
> (greeter-name joe-greet)
"Joe"
> (joe-greet "Mary")
"Hi Mary, I'm Joe"
> (joe-greet "John")
"Hi John, I'm Joe"
```

`#:super` *super-expr*

An alternative to supplying a `super-id` next to `struct-id`. Instead of the name of a structure type (which is not an expression), *super-expr* should produce a structure type descriptor value. An advantage of `#:super` is that structure type descriptors are values, so they can be passed to procedures.

Examples:

```
(define (make-raven-constructor super-type)
  (define-struct raven ()
                 #:super super-type
                 #:transparent
                 #:property prop:procedure (lambda (self)
                                             'nevermore))
  make-raven)
> (let ([r ((make-raven-constructor struct:posn) 1 2)])
    (list r (r)))
(#(struct:raven 1 2) nevermore)
> (let ([r ((make-raven-constructor struct:thing) "apple")])
    (list r (r)))
(#(struct:raven "apple") nevermore)
```

§4 “Structures” in §“**Reference**: PLT Scheme” provides more on structure types.

6 Modules

Scheme definitions and expressions are normally written inside of a module. Although a REPL evaluates definitions and expressions outside of a module for exploration and debugging purposes, and although `load` can evaluate definitions and expressions from a file as if they appeared in a REPL interaction, code that is meant to last for more than a few seconds belongs in a module.

6.1 Module Basics

The space of module names is distinct from the space of normal Scheme definitions. Indeed, since modules typically reside in files, the space of module names is explicitly tied to the filesystem at run time. For example, if the file `"/home/molly/cake.ss"` contains

```
#lang scheme

(provide print-cake)

; draws a cake with n candles
(define (print-cake n)
  (printf "   ~a  \n" (make-string n #\.))
  (printf " .-~a-.\n" (make-string n #\|))
  (printf " | ~a |\n" (make-string n #\space))
  (printf "---~a---\n" (make-string n #\-)))
```

then it can be used as the source of a module whose full name is based on the path `"/home/molly/cake.ss"`. The `provide` line exports the definition `print-cake` so that it can be used outside the module.

Instead of using its full path, a module is more likely to be referenced by a relative path. For example, a file `"/home/molly/random-cake.ss"` could use the `"cake.ss"` module like this:

```
#lang scheme

(require "cake.ss")

(print-cake (random 30))
```

The relative reference `"cake.ss"` in the import `(require "cake.ss")` works because the `"cake.ss"` module source is in the same directory as the `"random-cake.ss"` file. (Unix-style relative paths are used for relative module references on all platforms, much like relative URLs.)

Library modules that are distributed with PLT Scheme are usually referenced through an unquoted, suffixless path. The path is relative to the library installation directory, which contains directories for individual library *collections*. The module below refers to the `"date.ss"` library that is part of the `"scheme"` collection.

```
#lang scheme

(require scheme/date)

(printf "Today is ~s\n"
        (date->string (seconds->date (current-seconds))))
```

In addition to the main collection directory, which contains all collections that are part of the installation, collections can also be installed in a user-specific location. Finally, additional collection directories can be specified n configuration files or through the `PLTCOLLECTS` search path. Try running the following program to find out where your collections are:

```
#lang scheme

(require setup/dirs)

(find-collects-dir) ; main collection directory
(find-user-collects-dir) ; user-specific collection directory
(get-collects-search-dirs) ; complete search path
```

We discuss more forms of module reference later in §6.3 “Module Paths”.

6.2 Module Syntax

The `#lang` at the start of a module file begins a shorthand for a `module` form, much like `'` is a shorthand for a `quote` form. Unlike `'`, the `#lang` shorthand does not work well in a REPL, in part because it must be terminated by an end-of-file, but also because the longhand expansion of `#lang` depends on the name of the enclosing file.

6.2.1 The `module` Form

The longhand form of a module declaration, which works in a REPL as well as a file, is

```
(module name-id initial-module-path
  decl ...)
```

where the *name-id* is a name for the module, *initial-module-path* is an initial import, and each *decl* is an import, export, definition, or expression. In the case of a file, *name-id* must match the name of the containing file, minus its directory path or file extension.

The *initial-module-path* is needed because even the `require` form must be imported for further use in the module body. In other words, the *initial-module-path* import bootstraps the syntax available in the body. The most commonly used *initial-module-path* is `scheme`, which supplies most of the bindings described in this guide, including `require`, `define`, and `provide`. Another commonly used *initial-module-path* is `scheme/base`, which provides less functionality, but still much of the most commonly needed functions and syntax.

For example, the `"cake.ss"` example of the previous section could be written as

```
(module cake scheme
  (provide print-cake)

  (define (print-cake n)
    (printf "   ~a  \n" (make-string n #\.))
    (printf " .-~a-.\n" (make-string n #\|))
    (printf " | ~a |\n" (make-string n #\space))
    (printf "---~a---\n" (make-string n #\-))))
```

Furthermore, this `module` form can be evaluated in a REPL to declare a `cake` module that is not associated with any file. To refer to such an unassociated module, quote the module name:

Examples:

```
> (require 'cake)
> (print-cake 3)
   ...
 .-|||-.
 |     |
 ---------
```

Declaring a module does not immediately evaluate the body definitions and expressions of the module. The module must be explicitly `required` at the top level to trigger evaluation. After evaluation is triggered once, later `requires` do not re-evaluate the module body.

Examples:

```
> (module hi scheme
    (printf "Hello\n"))
> (require 'hi)
Hello
> (require 'hi)
```

6.2.2 The `#lang` Shorthand

The body of a `#lang` shorthand has no specific syntax, because the syntax is determined by the language name that follows `#lang`.

In the case of `#lang scheme`, the syntax is

```
#lang scheme
decl ...
```

which reads the same as

```
(module name scheme
  decl ...)
```

where *name* is derived from the name of the file that contains the `#lang` form.

The `#lang scheme/base` form has the same syntax as `#lang scheme`, except that the long-hand expansion uses `scheme/base` instead of `scheme`. The `#lang honu` form, in contrast, has a completely different syntax that doesn't even look like Scheme, and which we do not attempt to describe in this guide.

Unless otherwise specified, a module that is documented as a "language" using the `#lang` notation will expand to `module` in the same way as `#lang scheme`. The documented language name can be used directly with `module` or `require`, too.

6.3 Module Paths

A *module path* is a reference to a module, as used with `require` or as the *initial-module-path* in a `module` form. It can be any of several forms:

`(quote id)`

A module path that is a quoted identifier refers to a non-file `module` declaration using the identifier. This form of module reference makes the most sense in a REPL.

Examples:

```
> (module m scheme
    (provide color)
    (define color "blue"))
> (module n scheme
    (require 'm)
    (printf "my favorite color is ~a\n" color))
> (require 'n)
my favorite color is blue
```

---

id

A module path that is an unquoted identifier refers to an installed library. The *id* is constrained to contain only ASCII letters, ASCII numbers, `+`, `-`, `_`, and `/`, where `/` separates path elements within the identifier. The elements refer to collections and sub-collections, instead of directories and sub-directories.

An example of this form is `scheme/date`. It refers to the module whose source is the `"date.ss"` file in the `"scheme"` collection, which is installed as part of PLT Scheme. The `".ss"` suffix is added automatically.

Another example of this form is `scheme`, which is commonly used at the initial import. The path `scheme` is shorthand for `scheme/main`; when an *id* has no `/`, then `/main` is automatically added to the end. Thus, `scheme` or `scheme/main` refers to the module whose source is the `"main.ss"` file in the `"scheme"` collection.

Examples:

```
> (module m scheme
    (require scheme/date)

    (printf "Today is ~s\n"
            (date->string (seconds->date (current-seconds)))))
> (require 'm)
Today is "Wednesday, April 30th, 2008"
```

---

rel-string

A string module path is a relative path using Unix-style conventions: `/` is the path separator, `..` refers to the parent directory, and `.` refers to the same directory. The *rel-string* must not start or end with a path separator.

The path is relative to the enclosing file, if any, or it is relative to the current directory. (More precisely, the path is relative to the value of `(current-load-relative-directory)`, which is set while loading a file.)

§6.1 “Module Basics” shows examples using relative paths.

`(lib` *rel-string*`)`

Like an unquoted-identifier path, but expressed as a string instead of an identifier. Also, the *rel-string* can end with a file suffix, in case the relevant suffix is not `".ss"`.

Example of this form include `(lib "scheme/date.ss")` and `(lib "scheme/date")`, which are equivalent to `scheme/date`. Other examples include `(lib "scheme")`, `(lib "scheme/main")`, and `(lib "scheme/main.ss")`, which are all equivalent to `scheme`.

Examples:

```
> (module m (lib "scheme")
    (require (lib "scheme/date.ss"))

    (printf "Today is ~s\n"
            (date->string (seconds->date (current-seconds)))))
> (require 'm)
Today is "Wednesday, April 30th, 2008"
```

`(planet` *id*`)`

Accesses a third-party library that is distributed through the PLaneT server. The library is downloaded the first time that it is needed, and then the local copy is used afterward.

The *id* encodes several pieces of information separated by a `/`: the package owner, then package name with optional version information, and an optional path to a specific library with the package. Like *id* as shorthand for a `lib` path, a `".ss"` suffix is added automatically, and `/main` is used as the path if no sub-path element is supplied.

Examples:

```
> (module m (lib "scheme")
    ; Use "schematics"'s "random.plt" 1.0, file "random.ss":
    (require (planet schematics/random:1/random))
    (display (random-gaussian)))
> (require 'm)
0.9050686838895684
```

`(planet` *package-string*`)`

Like the symbol form of a `planet`, but using a string instead of an identifier. Also, the *package-string* can end with a file suffix, in case the relevant suffix is not `".ss"`.

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

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

A more general form to access a library from the PLaneT server. In this general form, a PLaneT reference starts like a `lib` reference with a relative path, but the path is followed by information about the producer, package, and version of the library. The specified package is downloaded and installed on demand.

The *vers*es specify a constraint on the acceptable version of the package, where a version number is a sequence of non-negative integers, and the constraints determine the allowable values for each element in the sequence. If no constraint is provided for a particular element, then any version is allowed; in particular, omitting all *vers*es means that any version is acceptable. Specifying at least one *vers* is strongly recommended.

For a version constraint, a plain *nat* is the same as `(+ nat)`, which matches *nat* or higher for the corresponding element of the version number. A `(`*start-nat end-nat*`)` matches any number in the range *start-nat* to *end-nat*, inclusive. A `(= nat)` matches only exactly *nat*. A `(- nat)` matches *nat* or lower.

Examples:

```
> (module m (lib "scheme")
    (require (planet "random.ss" ("schematics" "random.plt" 1 0)))
    (display (random-gaussian)))
> (require 'm)
0.9050686838895684
```

```
(file string)
```

Refers to a file, where *string* is a relative or absolute path using the current platform's conventions. This form is not portable, and it should *not* be used when a plain, portable `rel-string` suffices.

6.4 Imports: `require`

The `require` form imports from another module. A `require` form can appear within a module, in which case it introduces bindings from the specified module into importing module. A `require` form can also appear at the top level, in which case it both imports bindings

and *instantiates* the specified module; that is, it evaluates the body definitions and expressions of the specified module, if they have not been evaluated already.

A single `require` can specify multiple imports at once:

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

Specifying multiple *require-spec*s in a single `require` is essentially the same as using multiple `require`s, each with a single *require-spec*. The difference is minor, and confined to the top-level: a single `require` can import a given identifier at most once, whereas a separate `require` can replace the bindings of a previous `require` (both only at the top level, outside of a module).

The allowed shape of a *require-spec* is defined recursively:

---

module-path

In its simplest form, a *require-spec* is a *module-path* (as defined in the previous section, §6.3 "Module Paths"). In this case, the bindings introduced by `require` are determined by `provide` declarations within each module referenced by each *module-path*.

Examples:

```
> (module m scheme
    (provide color)
    (define color "blue"))
> (module n scheme
    (provide size)
    (define size 17))
> (require 'm 'n)
> (list color size)
("blue" 17)
```

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

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

An `only-in` form limits the set of bindings that would be introduced by a base *require-spec*. Also, `only-in` optionally renames each binding that is preserved: in a `[`*orig-id bind-id*`]` form, the *orig-id* refers to a binding implied by *require-spec*, and *bind-id* is the name that will be bound in the importing context instead of *bind-id*.

Examples:

```
> (module m (lib "scheme")
    (provide tastes-great?
             less-filling?)
    (define tastes-great? #t)
    (define less-filling? #t))
> (require (only-in 'm tastes-great?))
> tastes-great?
#t
> less-filling?
reference to undefined identifier: less-filling?
> (require (only-in 'm [less-filling? lite?]))
> lite?
#t
```

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

This form is the complement of `only`: it excludes specific bindings from the set specified by *require-spec*.

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

This form supports renaming like `only-in`, but leaving alone identifiers from *require-spec* that are not mentioned as an *orig-id*.

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

This is a shorthand for renaming, where *prefix-id* is added to the front of each identifier specified by *require-spec*.

The `only-in`, `except-in`, `rename-in`, and `prefix-in` forms can be nested to implement more complex manipulations of imported bindings. For example,

```
(require (prefix-in m: (except-in 'm ghost)))
```

imports all bindings that `m` exports, except for the `ghost` binding, and with local names that are prefixed with `m:`.

Equivalently, the `prefix-in` could be applied before `except-in`, as long as the omission with `except-in` is specified using the `m:` prefix:

```
(require (except-in (prefix m: 'm) m:ghost))
```

6.5 Exports: `provide`

By default, all of a module's definitions are private to the module. The `provide` form specifies definitions to be made available where the module is `required`.

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

A `provide` form can only appear at module level (i.e., in the immediate body of a `module`). Specifying multiple *`provide-spec`*s in a single `provide` is exactly the same as using multiple `provide`s each with a single *`provide-spec`*.

Each identifier can be exported at most once from a module across all `provide`s within the module. More precisely, the external name for each export must be distinct; the same internal binding can be exported multiple times with different external names.

The allowed shape of a *`provide-spec`* is defined recursively:

`identifier`

In its simplest form, a *`provide-spec`* indicates a binding within its module to be exported. The binding can be from either a local definition, or from an import.

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

A `rename-out` form is similar to just specifying an identifier, but the exported binding *`orig-id`* is given a different name, *`export-id`*, to importing modules.

`(struct-out` *`struct-id`*`)`

A `struct-out` form exports the bindings created by `(define-struct` *`struct-id ....`*`)`.

See §5 "Programmer-Defined Datatypes" for information on `define-struct`.

`(all-defined-out)`

The `all-defined-out` shorthand exports all bindings that are defined within the exporting module (as opposed to imported).

Use of the `all-defined-out` shorthand is generally discouraged, because it makes less clear the actual exports for a module, and because PLT Scheme programmers get into the habit of thinking that definitions can be added freely to a module without affecting its public interface (which is not the case when `all-defined-out` is used).

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

The `all-from-out` shorthand exports all bindings in the module that were imported using a *require-spec* that is based on *module-path*.

Although different *module-path*s could refer to the same file-based module, re-exporting with `all-from-out` is based specifically on the *module-path* reference, and not the module that is actually referenced.

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

Like *provide-spec*, but omitting the export of each *id*, where *id* is the external name of the binding to omit.

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

Like *provide-spec*, but adding *prefix-id* to the beginning of the external name for each exported binding.

6.6 Assignment and Redefinition

The use of `set!` on variables defined within a module is limited to the body of the defining module. That is, a module is allowed to change the value of its own definitions, and such changes are visible to importing modules. However, an importing context is not allowed to change the value of an imported binding.

Examples:

```
> (module m scheme
    (provide counter increment!)
    (define counter 0)
    (define (increment!)
      (set! counter (add1 counter))))
> (require 'm)
> counter
```

```
0
> (increment!)
> counter
1
> (set! counter -1)
set!: cannot mutate module-required variable in: counter
```

As the above example illustrates, a module can always grant others the ability to change its exports by providing a mutator function, such as `increment!`.

The prohibition on assignment of imported variables helps support modular reasoning about programs. For example, in the module,

```
(module m scheme
  (provide rx:fish fishy-string?)
  (define rx:fish #rx"fish")
  (define (fishy-string? s)
    (regexp-match? s rx:fish)))
```

the function `fishy-string?` will always match strings that contain "fish", no matter how other modules use the `rx:fish` binding. For essentially the same reason that it helps programmers, the prohibition on assignment to imports also allows many programs to be executed more efficiently.

Along the same lines, re-declaration of a module is not generally allowed. Indeed, for file-based modules, simply changing the file does not lead to a re-declaration, because file-based modules are loaded on demand, and the previously loaded declarations satisfy future requests. It is possible to use Scheme's reflection support to re-declare a module, however, and non-file modules can be re-declared in the REPL; in such cases, the redeclaration may fail if it involves the re-definition of a previously immutable binding.

```
> (module m scheme
    (define pie 3.141597))
> (require 'm)
> (module m scheme
    (define pie 3))
define-values: cannot re-define a constant: pie in module:
'm
```

For exploration and debugging purposes, the Scheme reflective layer provides a `compile-enforce-module-constants` parameter to disable the enforcement of constants.

```
> (compile-enforce-module-constants #f)
> (module m2 scheme
    (provide pie)
    (define pie 3.141597))
> (require 'm2)
```

```
> (module m2 scheme
    (provide pie)
    (define pie 3))
> (compile-enforce-module-constants #t)
> pie
3
```

7 Contracts

This chapter provides a gentle introduction to PLT Scheme's contract system. For the complete details see the §7 "Contracts" section in the reference manual.

7.1 Contracts and Boundaries

Like a contract between two business partners, a software contract is an agreement between two parties. The agreement specifies obligations and guarantees for each "product" (or value) that is handed from one party to the other.

A contract thus establishes a boundary between the two parties. Whenever a value crosses this boundary, the contract monitoring system performs contract checks, making sure the partners abide by the established contract.

In this spirit, PLT Scheme supports contracts only at module boundaries. Specifically, programmers may attach contracts to `provide` clauses and thus impose constraints and promises on the use of exported values. For example, the export specification

```
#lang scheme

(provide/contract
  [amount positive?])
(define amount ...)
```

promises to all clients of the above module that amount will always be a positive number. The contract system monitors `a`'s obligation carefully. Every time a client refers to `amount`, the monitor checks that the value of `amount` is indeed a positive number.

The contracts library is built into the Scheme language, but if you wish to use `scheme/base`, you can explicitly require the contracts library like this:

```
#lang scheme/base
(require scheme/contract) ; now we can write contracts

(provide/contract
  [amount positive?])
(define amount ...)
```

7.1.1 A first contract violation

Suppose the creator of `a` had written

```
#lang scheme
```

```
(provide/contract
  [amount positive?])

(define amount 0)
```

When module a is required, the monitoring system signals a violation of the contract and blame a for breaking its promises.

7.1.2 A subtle contract violation

Suppose the creator of a had written

```
#lang scheme

(provide/contract
  [amount positive?])

(define amount 'amount)
```

In that case, the monitoring system applies positive? to a symbol, but positive? reports an error, because its domain is only numbers. To make the contract capture our intentions for all Scheme values, we can ensure that the value is both a number and is positive, combining the two contracts with and/c:

```
(provide/contract
  [amount (and/c number? positive?)])
```

7.1.3 Imposing obligations on a module's clients

On occasion, a module may want to enter a contract with another module only if the other module abides by certain rules. In other words, the module isn't just promising some services, it also demands the client to deliver something. This kind of thing happens when a module exports a function, an object, a class or other values that enable values to flow in both directions.

7.2 Simple Contracts on Functions

When a module exports a function, it establishes two channels of communication between itself and the client module that imports the function. If the client module calls the function, it sends a value into the "server" module. Conversely, if such a function call ends and the function returns a value, the "server" module sends a value back to the "client" module.

It is important to keep this picture in mind when you read the explanations of the various ways of imposing contracts on functions.

7.2.1 Restricting the arguments of a function

Functions usually don't work on all possible Scheme values but only on a select subset such as numbers, booleans, etc. Here is a module that may represent a bank account:

```
#lang scheme

(provide/contract
  [create  (-> string? number? any)]
  [deposit (-> number? any)])

(define amount 0)
(define (create name initial-deposit) ...)
(define (deposit a) (set! amount (+ amount a)))
```

It exports two functions:

- `create`: The function's contract says that it consumes two arguments, a string and a number, and it promises nothing about the return value.
- `deposit`: The function's contract demands from the client modules that they apply it to numbers. It promises nothing about the return value.

If a "client" module were to apply `deposit` to `'silly`, it would violate the contract. The contract monitoring system would catch this violation and blame "client" for breaking the contract with the above module.

Note: Instead of `any` you could also use the more specific contract `void?`, which says that the function will always return the `(void)` value. This contract, however, would require the contract monitoring system to check the return value every time the function is called, even though the "client" module can't do much with this value anyway. In contrast, `any` tells the monitoring system *not* to check the return value. Additionally, it tells a potential client that the "server" module *makes no promises at all* about the function's return value.

7.2.2 Arrows

It is natural to use an arrow to say that an exported value is a function. In decent high schools, you learn that a function has a domain and a range, and that you write this fact down like this:

```
f : A -> B
```

Here the A and B are sets; A is the domain and B is the range.

Functions in a programming language have domains and ranges, too. In statically typed languages, you write down the names of types for each argument and for the result. When all you have, however, is a Scheme name, such as create or deposit, you want to tell the reader what the name represents (a function) and, if it is a function (or some other complex value) what the pieces are supposed to be. This is why we use a -> to say "hey, expect this to be a function."

So -> says "this is a contract for a function." What follows in a function contracts are contracts (sub-contracts if you wish) that tell the reader what kind of arguments to expect and what kind of a result the function produces. For example,

```
(provide/contract
 [create (-> string? number? boolean? account?)])
```

says that create is a function of three arguments: a string, a number, and a boolean. Its result is an account.

In short, the arrow -> is a *contract combinator*. Its purpose is to combine other contracts into a contract that says "this is a function *and* its arguments and its result are like that."

7.2.3 Infix contract notation

If you are used to mathematics, you like the arrow in between the domain and the range of a function, not at the beginning. If you have read *How to Design Programs*, you have seen this many times. Indeed, you may have seen contracts such as these in other people's code:

```
(provide/contract
  [create (string? number? boolean? . -> . account?)])
```

If a PLT Scheme S-expression contains two dots with a symbol in the middle, the reader re-arranges the S-expression and place the symbol at the front. Thus,

```
(string? number? boolean? . -> . account?)
```

is really just a short-hand for

```
(-> string? number? boolean? account?)
```

Of course, placing the arrow to the left of the range follows not only mathematical tradition but also that of typed functional languages.

7.2.4 Rolling your own contracts for function arguments

The `deposit` function adds the given number to the value of `amount`. While the function's contract prevents clients from applying it to non-numbers, the contract still allows them to apply the function to complex numbers, negative numbers, or inexact numbers, all of which do not represent amounts of money.

To this end, the contract system allows programmers to define their own contracts:

```
#lang scheme

(define (amount? a)
  (and (number? a) (integer? a) (exact? a) (>= a 0)))

(provide/contract
  ; an amount is a natural number of cents
  ; is the given number an amount?
  [deposit (-> amount? any)]
  [amount? (-> any/c boolean?)])

(define this 0)
(define (deposit a) (set! this (+ this a)))
```

The module introduces a predicate, `amount?`. The `provide` clause refers to this predicate, as a contract, for its specification of the contract of `deposit`.

Of course it makes no sense to restrict a channel of communication to values that the client doesn't understand. Therefore the module also exports the `amount?` predicate itself, with a contract saying that it accepts an arbitrary value and returns a boolean.

In this case, we could also have used `natural-number/c`, which is a contract defined in `scheme/contract` that is equivalent to `amount` (modulo the name):

```
#lang scheme

(provide/contract
  ; an amount is a natural number of cents
  [deposit (-> natural-number/c any)])

(define this 0)
(define (deposit a) (set! this (+ this a)))
```

Lesson: learn about the built-in contracts in `scheme/contract`.

7.2.5 The `and/c`, `or/c`, and `listof` contract combinators

Both `and/c` and `or/c` ombine contracts and they do what you expect them to do.

For example, if we didn't have `natural-number/c`, the `amount?` contract is a bit opaque. Instead, we would define it as follows:

```
#lang scheme

(define amount
  (and/c number? integer? exact? (or/c positive? zero?)))

(provide/contract
  ; an amount is a natural number of cents
  ; is the given number an amount?
  [deposit (-> amount any)])

(define this 0)
(define (deposit a) (set! this (+ this a)))
```

That is, amount is a contract that enforces the following conditions: the value satisfies `number?` and `integer?` and `exact?` and is either `positive?` or `zero?`.

Oh, we almost forgot. What do you think `(listof char?)` means? Hint: it is a contract!

7.2.6 Restricting the range of a function

Consider a utility module for creating strings from banking records:

```
#lang scheme

(define (has-decimal? str)
  (define L (string-length str))
  (and (>= L 3)
       (char=?
        #\.
        (string-ref result (- L 3)))))

(provide/contract
  ; convert a random number to a string
  [format-number (-> number? string?)]

  ; convert an amount into a dollar based string
  [format-nat (-> natural-number/c
                  (and/c string? has-decimal?))])
```

The contract of the exported function `format-number` specifies that the function consumes a number and produces a string.

The contract of the exported function `format-nat` is more interesting than the one of `format-number`. It consumes only natural numbers. Its range contract promises a string that has a `.` in the third position from the right.

Exercise 2 Strengthen the promise of the range contract for `format-nat` so that it admits only strings with digits and a single dot.

Solution to exercise 2

```
#lang scheme

(define (digit-char? x)
  (member x '(#\1 #\2 #\3 #\4 #\5 #\6 #\7 #\8 #\9 #\0)))

(define (has-decimal? str)
  (define L (string-length str))
  (and (>= L 3)
       (char=?
        #\.
        (string-ref result (- L 3)))))

(define (is-decimal-string? str)
  (define L (string-length str))
  (and (has-decimal? str)
       (andmap digit-char?
               (string->list (substring result 0 (- L 3))))
       (andmap digit-char?
               (string->list (substring result (- L 2) L)))))

(provide/contract
  ...
  ; convert a random number to a string
  [format-number (-> number? string?)]

  ; convert an amount (natural number) of cents
  ; into a dollar based string
  [format-nat (-> natural-number/c
                  (lambda (result)
                    (and (string? result)
                         (is-decimal-string? result))))])
```

7.2.7 The difference between `any` and `any/c`

The contract `any/c` accepts any value, and `any` is a keyword that can appear in the range of the function contracts (`->`, `->*`, and `->d`), so it is natural to wonder what the difference between these two contracts is:

```
(-> integer? any)
(-> integer? any/c)
```

Both allow any result, right? There are two differences:

- In the first case, the function may return anything at all, including multiple values. In the second case, the function may return any value, but not more than one. For example, this function:

  ```
  (define (f x) (values (+ x 1) (- x 1)))
  ```

  meets the first contract, but not the second one.

- Relatedly, this means that a call to a function that has the second contract is not a tail call. So, for example, the following program is an infinite loop that takes only a constant amount of space, but if you replace `any` with `any/c`, it uses up all of the memory available.

```
(module server scheme
  (provide/contract
   [f (-> (-> procedure? any) boolean?)])
  (define (f g) (g g)))

(module client scheme
  (require 'server)
  (f f))

(require 'client)
```

7.3 Contracts on Functions in General

7.3.1 Contract error messages that contain "..."

You wrote your module. You added contracts. You put them into the interface so that client programmers have all the information from interfaces. It's a piece of art:

```
#lang scheme

(provide/contract
```

```
 [deposit (-> (lambda (x)
                (and (number? x) (integer? x) (>= x 0)))
              any)])

(define this 0)
(define (deposit a) ...)
```

Several clients used your module. Others used their modules in turn. And all of a sudden one of them sees this error message:

> *bank-client broke the contract (-> ??? any) it had with myaccount on deposit; expected <???>, given: -10*

Clearly, `bank-client` is a module that uses `myaccount` but what is the *???* doing there? Wouldn't it be nice if we had a name for this class of data much like we have string, number, and so on?

For this situation, PLT Scheme provides *flat named contracts*. The use of "contract" in this term shows that contracts are first-class values. The "flat" means that the collection of data is a subset of the built-in atomic classes of data; they are described by a predicate that consumes all Scheme values and produces a boolean. The "named" part says what we want to do, which is to name the contract so that error messages become intelligible:

```
#lang scheme

(define (amount? x) (and (number? x) (integer? x) (>= x 0)))
(define amount (flat-named-contract 'amount amount?))

(provide/contract
 [deposit (amount . -> . any)])

(define this 0)
(define (deposit a) ...)
```

With this little change, the error message becomes all of the sudden quite readable:

> *bank-client broke the contract (-> amount any) it had with myaccount on deposit; expected <amount>, given: -10*

7.3.2 Optional arguments

Take a look at this excerpt from a string-processing module, inspired by the Scheme cookbook:

```
#lang scheme

(provide/contract
 ; pad the given str left and right with
 ; the (optional) char so that it is centered
 [string-pad-center (->* (string? natural-number/c)
                         (char?)
                         string?)])

(define (string-pad-center str width [pad #\space])
  (define field-width (min width (string-length str)))
  (define rmargin (ceiling (/ (- width field-width) 2)))
  (define lmargin (floor (/ (- width field-width) 2)))
  (string-append (build-string lmargin (λ (x) pad))
                 str
                 (build-string rmargin (λ (x) pad))))
```

The module exports `string-pad-center`, a function that creates a string of a given `width` with the given string in the center. The default fill character is `#\space`; if the client module wishes to use a different character, it may call `string-pad-center` with a third argument, a `char`, overwriting the default.

The function definition uses optional arguments, which is appropriate for this kind of functionality. The interesting point here is the formulation of the contract for the `string-pad-center`.

The contract combinator `->*`, demands several groups of contracts:

- The first one is a parenthesized group of contracts for all required arguments. In this example, we see two: `string?` and `natural-number/c`.
- The second one is a parenthesized group of contracts for all optional arguments: `char?`.
- The last one is a single contract: the result of the function.

Note if a default value does not satisfy a contract, you won't get a contract error for this interface. In contrast to type systems, we do trust you; if you can't trust yourself, you need to communicate across boundaries for everything you write.

7.3.3 Rest arguments

We all know that `+` in Beginner Scheme is a function that consumes at least two numbers but, in principle, arbitrarily many more. Defining the function is easy:

```
(define (plus fst snd . rst)
  (foldr + (+ fst snd) rst))
```

Describing this function via a contract is difficult because of the rest argument (`rst`).

Here is the contract:

```
(provide/contract
 [plus (->* (number? number?) () #:rest (listof number?) number?)])
```

The `->*` contract combinator empowers you to specify functions that consume a variable number of arguments or functions like `plus`, which consume "at least this number" of arguments but an arbitrary number of additional arguments.

The contracts for the required arguments are enclosed in the first pair of parentheses:

```
(number? number?)
```

For `plus` they demand two numbers. The empty pair of parenthesis indicates that there are no optional arguments (not counting the rest arguments) and the contract for the rest argument follows `#:rest`

```
(listof number?)
```

Since the remainder of the actual arguments are collected in a list for a rest parameter such as `rst`, the contract demands a list of values; in this specific examples, these values must be number.

7.3.4 Keyword arguments

Sometimes, a function accepts many arguments and remembering their order can be a nightmare. To help with such functions, PLT Scheme has keyword arguments.

For example, consider this function that creates a simple GUI and asks the user a yes-or-no question:

```
#lang scheme/gui

(define (ask-yes-or-no-question #:question question
                                #:default answer
                                #:title title
                                #:width w
                                #:height h)
  (define d (new dialog% [label title] [width w] [height h]))
  (define msg (new message% [label question] [parent d]))
  (define (yes) (set! answer #t) (send d show #f))
```

```
  (define (no) (set! answer #f) (send d show #f))
  (define yes-b (new button%
                     [label "Yes"] [parent d]
                     [callback (λ (x y) (yes))]
                     [style (if answer '(border) '())]))
  (define no-b (new button%
                    [label "No"] [parent d]
                    [callback (λ (x y) (no))]
                    [style (if answer '() '(border))]))
  (send d show #t)
  answer)

(provide/contract
 [ask-yes-or-no-question
  (-> #:question string?
      #:default boolean?
      #:title string?
      #:width exact-integer?
      #:height exact-integer?
      boolean?)])
```

Note that if you really want to ask a yes-or-no question via a GUI, you should use `message-box/custom` (and generally speaking, avoiding the responses "yes" and "no" in your dialog is a good idea, too ...).

The contract for `ask-yes-or-no-question` uses our old friend the `->` contract combinator. Just like `lambda` (or `define`-based functions) use keywords for specifying keyword arguments, it uses keywords for specifying contracts on keyword arguments. In this case, it says that `ask-yes-or-no-question` must receive five keyword arguments, one for each of the keywords `#:question`, `#:default`, `#:title`, `#:width`, and `#:height`. Also, just like in a function definition, the keywords in the `->` may appear in any order.

7.3.5 Optional keyword arguments

Of course, many of the parameters in `ask-yes-or-no-question` (from the previous question) have reasonable defaults, and should be made optional:

```
(define (ask-yes-or-no-question #:question question
                                #:default answer
                                #:title [title "Yes or No?"]
                                #:width [w 400]
                                #:height [h 200])
  ...)
```

To specify this function's contract, we need to use `->*`. It too supports keywords just as you might expect, in both the optional and mandatory argument sections. In this case, we have mandatory keywords `#:question` and `#:default`, and optional keywords `#:title`, `#:width`, and `#:height`. So, we write the contract like this:

```
(provide/contract
 [ask-yes-or-no-question
  (->* (#:question string?
        #:default boolean?)

       (#:title string?
        #:width exact-integer?
        #:height exact-integer?)

       boolean?)])
```

putting the mandatory keywords in the first section and the optional ones in the second section.

7.3.6 When a function's result depends on its arguments

Here is an excerpt from an imaginary (pardon the pun) numerics module:

```
#lang scheme
(provide/contract
 [sqrt.v1 (->d ([argument (>=/c 1)])
               ()
               [result (<=/c argument)])])
...
```

The contract for the exported function `sqrt.v1` uses the `->d` rather than `->` function contract. The "d" stands for *dependent* contract, meaning the contract for the function range depends on the value of the argument.

In this particular case, the argument of `sqrt.v1` is greater or equal to 1. Hence a very basic correctness check is that the result is smaller than the argument. (Naturally, if this function is critical, one could strengthen this check with additional clauses.)

In general, a dependent function contract looks just like the more general `->*` contract, but with names added that can be used elsewhere in the contract.

Yes, there are many other contract combinators such as `<=/c` and `>=/c`, and it pays off to look them up in the contract section of the reference manual. They simplify contracts tremendously and make them more accessible to potential clients.

7.3.7 When contract arguments depend on each other

Eventually bank customers want their money back. Hence, a module that implements a bank account must include a method for withdrawing money. Of course, ordinary accounts don't

let customers withdraw an arbitrary amount of money but only as much as they have in the account.

Suppose the account module provides the following two functions:

```
balance : (-> account amount)
withdraw : (-> account amount account)
```

Then, informally, the proper precondition for `withdraw` is that "the balance of the given account is greater than or equal to the given (withdrawal) amount." The postcondition is similar to the one for `deposit`: "the balance of the resulting account is larger than (or equal to) than the one of the given account." You could of course also formulate a full-fledged correctness condition, namely, that the balance of the resulting account is equal to the balance of the given one, plus the given amount.

The following module implements accounts imperatively and specifies the conditions we just discussed:

```
#lang scheme

; section 1: the contract definitions
(define-struct account (balance))
(define amount natural-number/c)

(define msg> "account a with balance larger than ~a expected")
(define msg< "account a with balance less than ~a expected")

(define (mk-account-contract acc amt op msg)
  (define balance0 (balance acc))
  (define (ctr a)
    (and (account? a) (op balance0 (balance a))))
  (flat-named-contract (format msg balance0) ctr))

; section 2: the exports
(provide/contract
 [create   (amount . -> . account?)]
 [balance  (account? . -> . amount)]
 [withdraw (->d ([acc account?]
                 [amt (and/c amount (<=/c (balance acc)))])
                ()
                [result (mk-account-contract acc amt > msg>)])]
 [deposit  (->d ([acc account?]
                 [amt amount])
                ()
                [result (mk-account-contract acc amt < msg<)])])

; section 3: the function definitions
```

```
(define balance account-balance)

(define (create amt) (make-account amt))

(define (withdraw acc amt)
  (set-account-balance! acc (- (balance acc) amt))
  acc)

(define (deposit acc amt)
  (set-account-balance! acc (+ (balance acc) amt))
  acc)
```

The second section is the export interface:

- `create` consumes an initial deposit and produces an account. This kind of contract is just like a type in a statically typed language, except that statically typed languages usually don't support the type "natural numbers" (as a full-fledged subtype of numbers).
- `balance` consumes an account and computes its current balance.
- `withdraw` consumes an account, named `acc`, and an amount, `amt`. In addition to being an `amount`, the latter must also be less than `(balance acc)`, i.e., the balance of the given account. That is, the contract for `amt` depends on the value of `acc`, which is what the `->d` contract combinator expresses.

  The result contract is formed on the fly: `(mk-account-contract acc amt > msg>)`. It is an application of a contract-producing function that consumes an account, an amount, a comparison operator, and an error message (a format string). The result is a contract.
- `deposit`'s contract has been reformulated using the `->d` combinator.

The code in the first section defines all those pieces that are needed for the formulation of the export contracts: `account?`, `amount`, error messages (format strings), and `mk-account-contract`. The latter is a function that extracts the current balance from the given account and then returns a named contract, whose error message (contract name) is a string that refers to this balance. The resulting contract checks whether an account has a balance that is larger or smaller, depending on the given comparison operator, than the original balance.

7.3.8 Ensuring that a function properly modifies state

The `->d` contract combinator can also ensure that a function only modifies state according to certain constraints. For example, consider this contract (it is a slightly simplified from the function `preferences:add-panel` in the framework):

```
(->d ([parent (is-a?/c area-container-window<%>)])
     ()
     [_
      (let ([old-children (send parent get-children)])
        (λ (child)
          (andmap eq?
                  (append old-children (list child))
                  (send parent get-children))))])
```

It says that the function accepts a single argument, named `parent`, and that `parent` must be an object matching the interface `area-container-window<%>`.

The range contract ensures that the function only modifies the children of `parent` by adding a new child to the front of the list. It accomplishes this by using the `_` instead of a normal identifier, which tells the contract library that the range contract does not depend on the values of any of the results, and thus the contract library evaluates the expression following the `_` when the function is called, instead of when it returns. Therefore the call to the `get-children` method happens before the function under the contract is called. When the function under contract returns, its result is passed in as `child`, and the contract ensures that the children after the function return are the same as the children before the function called, but with one more child, at the front of the list.

To see the difference in a toy example that focuses on this point, consider this program

```
#lang scheme
(define x '())
(define (get-x) x)
(define (f) (set! x (cons 'f x)))
(provide/contract
 [f (->d () () [_ (begin (set! x (cons 'ctc x)) any/c)])]
 [get-x (-> (listof symbol?))])
```

If you were to require this module, call `f`, then the result of `get-x` would be `'(f ctc)`. In contrast, if the contract for `f` were

```
(->d () () [res (begin (set! x (cons 'ctc x)) any/c)])
```

(only changing the underscore to `res`), then the result of `get-x` would be `'(ctc f)`.

7.3.9 Contracts for `case-lambda`

Dybvig, in Chapter 5 of the Chez Scheme User's Guide, explains the meaning and pragmatics of `case-lambda` with the following example (among others):

```
(define substring1
  (case-lambda
```

```
    [(s) (substring1 s 0 (string-length s))]
    [(s start) (substring1 s start (string-length s))]
    [(s start end) (substring s start end)]))
```

This version of `substring` has one of the following signature:

- just a string, in which case it copies the string;
- a string and an index into the string, in which case it extracts the suffix of the string starting at the index; or
- a string a start index and an end index, in which case it extracts the fragment of the string between the two indices.

The contract for such a function is formed with the `case->` combinator, which combines as many functional contracts as needed:

```
(provide/contract
  [substring1
   (case->
    (string? . -> . string?)
    (string? natural-number/c . -> . string?)
    (string? natural-number/c natural-number/c . -> . string?))])
```

As you can see, the contract for `substring1` combines three function contracts, just as many clauses as the explanation of its functionality required.

7.3.10 Multiple result values

The function `split` consumes a list of `char`s and delivers the string that occurs before the first occurrence of `#\newline` (if any) and the rest of the list:

```
(define (split l)
  (define (split l w)
    (cond
      [(null? l) (values (list->string (reverse w)) '())]
      [(char=? #\newline (car l))
       (values (list->string (reverse w)) (cdr l))]
      [else (split (cdr l) (cons (car l) w))]))
  (split l '()))
```

It is a typical multiple-value function, returning two values by traversing a single list.

The contract for such a function can use the ordinary function arrow `->`, since it treats `values` specially, when it appears as the last result:

```
(provide/contract
 [split (-> (listof char?)
            (values string? (listof char?)))])
```

The contract for such a function can also be written using `->*`, just like `plus`:

```
(provide/contract
 [split (->* ((listof char?))
             ()
             (values string? (listof char?)))])
```

As before the contract for the argument is wrapped in an extra pair of parentheses (and must always be wrapped like that) and the empty pair of parentheses indicates that there are no optoinal arguments. The contracts for the results are inside `values`: a string and a list of characters.

Now suppose we also want to ensure that the first result of `split` is a prefix of the given word in list format. In that case, we need to use the `->d` contract combinator:

```
(define (substring-of? s)
  (flat-named-contract
    (format "substring of ~s" s)
    (lambda (s2)
      (and (string? s2)
           (<= (string-length s2) s)
           (equal? (substring s 0 (string-length s2)) s2)))))

(provide/contract
 [split (->d ([fl (listof char?)])
             ()
             (values [s (substring-of (list->string fl))]
                     [c (listof char?)]))])
```

Like `->*`, the `->d` combinator uses a function over the argument to create the range contracts. Yes, it doesn't just return one contract but as many as the function produces values: one contract per value. In this case, the second contract is the same as before, ensuring that the second result is a list of `char`s. In contrast, the first contract strengthens the old one so that the result is a prefix of the given word.

This contract is expensive to check of course. Here is a slightly cheaper version:

```
(provide/contract
 [split (->d ([fl (listof char?)])
             ()
             (values [s (string-len/c (length fl))]
                     [c (listof char?)]))])
```

Click on string-len/c to see what it does.

7.3.11 Procedures of some fixed, but statically unknown arity

Imagine yourself writing a contract for a function that accepts some other function and a list of numbers that eventually applies the former to the latter. Unless the arity of the given function matches the length of the given list, your procedure is in trouble.

Consider this n-step function:

```
; (number ... -> (union #f number?)) (listof number) -> void
(define (n-step proc inits)
  (let ([inc (apply proc inits)])
    (when inc
      (n-step proc (map (λ (x) (+ x inc)) inits)))))
```

The argument of n-step is proc, a function proc whose results are either numbers or false, and a list. It then applies proc to the list inits. As long as proc returns a number, n-step treats that number as an increment for each of the numbers in inits and recurs. When proc returns false, the loop stops.

Here are two uses:

```
; nat -> nat
(define (f x)
  (printf "~s \n" x)
  (if (= x 0) #f -1))
(n-step f '(2))

; nat nat -> nat
(define (g x y)
  (define z (+ x y))
  (printf "~s\n" (list x y z))
  (if (= z 0) #f -1))

(n-step g '(1 1))
```

A contract for n-step must specify two aspects of proc's behavior: its arity must include the number of elements in inits, and it must return either a number or #f. The latter is easy, the former is difficult. At first glance, this appears to suggest a contract that assigns a *variable-arity* to proc:

```
(->* ()
     (listof any/c)
     (or/c number? false/c))
```

This contract, however, says that the function must accept *any* number of arguments, not a *specific* but *undetermined* number. Thus, applying `n-step` to `(lambda (x) x)` and `(list 1)` breaks the contract because the given function accepts only one argument.

The correct contract uses the `unconstrained-domain->` combinator, which specifies only the range of a function, not its domain. It is then possible to combine this contract with an arity test to specify the correct `n-step`'s contract:

```
(provide/contract
 [n-step
  (->d ([proc
         (and/c (unconstrained-domain->
                 (or/c false/c number?))
                (λ (f) (procedure-arity-includes?
                        f
                        (length inits))))]
        [inits (listof number?)])
       ()
       any)])
```

7.4 Contracts on Structures

Modules deal with structures in two ways. First they export `struct` definitions, i.e., the ability to create structs of a certain kind, to access their fields, to modify them, and to distinguish structs of this kind against every other kind of value in the world. Second, on occasion a module exports a specific struct and wishes to promise that its fields contain values of a certain kind. This section explains how to protect structs with contracts for both uses.

7.4.1 Promising something about a specific struct

Yes. If your module defines a variable to be a structure, then on export you can specify the structures shape:

```
#lang scheme
(require lang/posn)

(define origin (make-posn 0 0))

(provide/contract
 [origin (struct/c posn zero? zero?)])
```

In this example, the module imports a library for representing positions, which exports a `posn` structure. One of the `posn`s it creates and exports stands for the origin, i.e., `(0,0)`, of

the grid.

7.4.2 Promising something about a specific vector

Yes, again. See the help desk for information on `vector/c` and similar contract combinators for (flat) compound data.

7.4.3 Ensuring that all structs are well-formed

The book *How to Design Programs* teaches that `posn`s should contain only numbers in their two fields. With contracts we would enforce this informal data definition as follows:

```
#lang scheme
(define-struct posn (x y))

(provide/contract
 [struct posn ((x number?) (y number?))]
 [p-okay posn?]
 [p-sick posn?])

(define p-okay (make-posn 10 20))
(define p-sick (make-posn 'a 'b))
```

This module exports the entire structure definition: `make-posn`, `posn?`, `posn-x`, `posn-y`, `set-posn-x!`, and `set-posn-y!`. Each function enforces or promises that the two fields of a `posn` structure are numbers—when the values flow across the module boundary.

Thus, if a client calls `make-posn` on `10` and `'a`, the contract system signals a contract violation.

The creation of `p-sick` inside of the `posn` module, however, does not violate the contracts. The function `make-posn` is used internally so `'a` and `'b` don't cross the module boundary. Similarly, when `p-sick` crosses the boundary of `posn`, the contract promises a `posn?` and nothing else. In particular, this check does *not* require that the fields of `p-sick` are numbers.

The association of contract checking with module boundaries implies that `p-okay` and `p-sick` look alike from a client's perspective until the client extracts the pieces:

```
#lang scheme
(require lang/posn)

... (posn-x p-sick) ...
```

Using `posn-x` is the only way the client can find out what a `posn` contains in the `x` field. The

application of `posn-x` sends `p-sick` back into the `posn` module and the result value – `'a` here – back to the client, again across the module boundary. At this very point, the contract system discovers that a promise is broken. Specifically, `posn-x` doesn't return a number but a symbol and is therefore blamed.

This specific example shows that the explanation for a contract violation doesn't always pinpoint the source of the error. The good news is that the error is located in the `posn` module. The bad news is that the explanation is misleading. Although it is true that `posn-x` produced a symbol instead of a number, it is the fault of the programmer who created a `posn` from symbols, i.e., the programmer who added

```
(define p-sick (make-posn 'a 'b))
```

to the module. So, when you are looking for bugs based on contract violations, keep this example in mind.

Exercise 1 Use your knowledge from the §7.4.1 "Promising something about a specific struct" section on exporting specific structs and change the contract for `p-sick` so that the error is caught when `sick` is exported.

Solution to exercise 1

A single change suffices:

```
(provide/contract
 ...
 [p-sick (struct/c posn number? number?)])
```

Instead of exporting `p-sick` as a plain `posn?`, we use a `struct/c` contract to enforce constraints on its components.

7.4.4 Checking properties of data structures

Contracts written using `struct/c` immediately check the fields of the data structure, but sometimes this can have disastrous effects on the performance of a program that does not, itself, inspect the entire data structure.

As an example, consider the the binary search tree search algorithm. A binary search tree is like a binary tree, except that the numbers are organized in the tree to make searching the tree fast. In particular, for each interior node in the tree, all of the numbers in the left subtree are smaller than the number in the node, and all of the numbers in the right subtree are larger than the number in the node.

We can implement a search function `in?` that takes advantage of the structure of the binary search tree.

```
#lang scheme
```

```
(define-struct node (val left right))

; determines if 'n' is in the binary search tree 'b',
; exploiting the binary search tree invariant
(define (in? n b)
  (cond
    [(null? b) #f]
    [else (cond
            [(= n (node-val b))
             #t]
            [(< n (node-val b))
             (in? n (node-left b))]
            [(> n (node-val b))
             (in? n (node-right b))])]))

; a predicate that identifies binary search trees
(define (bst-between? b low high)
  (or (null? b)
      (and (<= low (node-val b) high)
           (bst-between? (node-left b) low (node-val b))
           (bst-between? (node-right b) (node-val b) high))))

(define (bst? b) (bst-between? b -inf.0 +inf.0))

(provide (struct node (val left right)))
(provide/contract
 [bst? (any/c . -> . boolean?)]
 [in? (number? bst? . -> . boolean?)])
```

In a full binary search tree, this means that the `in?` function only has to explore a logarithmic number of nodes.

The contract on `in?` guarantees that its input is a binary search tree. But a little careful thought reveals that this contract defeats the purpose of the binary search tree algorithm. In particular, consider the inner `cond` in the `in?` function. This is where the `in?` function gets its speed: it avoids searching an entire subtree at each recursive call. Now compare that to the `bst-between?` function. In the case that it returns `#t`, it traverses the entire tree, meaning that the speedup of `in?` is lost.

In order to fix that, we can employ a new strategy for checking the binary search tree contract. In particular, if we only checked the contract on the nodes that `in?` looks at, we can still guarantee that the tree is at least partially well-formed, but without changing the complexity.

To do that, we need to use `define-contract-struct` in place of `define-struct`. Like `define-struct`, `define-contract-struct` defines a maker, predicate, and selectors for

a new structure. Unlike `define-struct`, it also defines contract combinators, in this case `node/c` and `node/dc`. Also unlike `define-struct`, it does not allow mutators, making its structs always immutable.

The `node/c` function accepts a contract for each field of the struct and returns a contract on the struct. More interestingly, the syntactic form `node/dc` allows us to write dependent contracts, i.e., contracts where some of the contracts on the fields depend on the values of other fields. We can use this to define the binary search tree contract:

```
#lang scheme

(define-contract-struct node (val left right))

; determines if 'n' is in the binary search tree 'b'
(define (in? n b) ... as before ...)

; bst-between : number number -> contract
; builds a contract for binary search trees
; whose values are betweeen low and high
(define (bst-between/c low high)
  (or/c null?
        (node/dc [val (between/c low high)]
                 [left (val) (bst-between/c low val)]
                 [right (val) (bst-between/c val high)])))

(define bst/c (bst-between/c -inf.0 +inf.0))

(provide make-node node-left node-right node-val node?)
(provide/contract
 [bst/c contract?]
 [in? (number? bst/c . -> . boolean?)])
```

In general, each use of `node/dc` must name the fields and then specify contracts for each field. In the above, the `val` field is a contract that accepts values between `low` and `high`. The `left` and `right` fields are dependent on the value of the `val` field, indicated by their second sub-expressions. Their contracts are built by recursive calls to the `bst-between/c` function. Taken together, this contract ensures the same thing that the `bst-between?` function checked in the original example, but here the checking only happens as `in?` explores the tree.

Although this contract improves the performance of `in?`, restoring it to the logarithmic behavior that the contract-less version had, it is still imposes a fairly large constant overhead. So, the contract library also provides `define-opt/c` that brings down that constant factor by optimizing its body. Its shape is just like the `define` above. It expects its body to be a contract and then optimizes that contract.

```
(define-opt/c (bst-between/c low high)
```

```
(or/c null?
      (node/dc [val (between/c low high)]
               [left (val) (bst-between/c low val)]
               [right (val) (bst-between/c val high)])))
```

7.5 Examples

This section illustrates the current state of PLT Scheme's contract implementation with a series of examples from Mitchell and McKim's text book "Design by Contract, by Example" [Addison and Wesley, 2002].

Mitchell and McKim's principles for design by contract DbC are derived from the 1970s style algebraic specifications. The overall goal of DbC is to specify the constructors of an algebra in terms of its observers. While we reformulate Mitchell and McKim's terminology and we use a mostly applicative, we retain their terminology of "classes" and "objects":

- **Separate queries from commands.**

  A *query* returns a result but does not change the observable properties of an object. A *command* changes the visible properties of an object, but does not return a result. In applicative implementation a command typically returns an new object of the same class.

- **Separate basic queries from derived queries**

  A *derived query* returns a result that is computable in terms of basic queries.

- **For each derived query, write a post-condition contract that specifies the result in terms of the basic queries.**

- **For each command, write a post-condition contract that specifies the changes to the observable properties in terms of the basic queries.**

- **For each query and command, decide on suitable pre-condition contract.**

Each of the following sections corresponds to a chapter in Mitchell and McKim's book (but not all chapters show up here). We recommend that you read the contracts first (near the end of the first modules), then the implementation (in the first modules), and then the test module (at the end of each section).

Mitchell and McKim use Eiffel as the underlying programming language and employ a conventional imperative programming style. Our long-term goal is to transliterate their examples into applicative Scheme, structure-oriented imperative Scheme, and PLT Scheme's class system.

Note: To mimic Mitchell and McKim's informal notion of parametericity (parametric polymorphism), we use first-class contracts. At several places, this use of first-class contracts improves on Mitchell and McKim's design (see comments in interfaces).

7.5.1 A Customer Manager Component for Managing Customer Relationships

This first module contains some struct definitions in a separate module in order to better track bugs.

```
#lang scheme
; data definitions

(define id? symbol?)
(define id-equal? eq?)
(define-struct basic-customer (id name address) #:mutable)

; interface
(provide/contract
 [id?                   (-> any/c boolean?)]
 [id-equal?             (-> id? id? boolean?)]
 [struct basic-customer ((id id?)
                         (name string?)
                         (address string?))])
; end of interface
```

This module contains the program that uses the above.

```
#lang scheme

(require "1.ss") ; the module just above

; implementation
; [listof (list basic-customer? secret-info)]
(define all '())

(define (find c)
  (define (has-c-as-key p)
    (id-equal? (basic-customer-id (car p)) c))
  (define x (filter has-c-as-key all))
  (if (pair? x) (car x) x))

(define (active? c)
  (define f (find c))
  (pair? (find c)))

(define not-active? (compose not active? basic-customer-id))

(define count 0)

(define (add c)
```

```
  (set! all (cons (list c 'secret) all))
  (set! count (+ count 1)))

(define (name id)
  (define bc-with-id (find id))
  (basic-customer-name (car bc-with-id)))

(define (set-name id name)
  (define bc-with-id (find id))
  (set-basic-customer-name! (car bc-with-id) name))

(define c0 0)
; end of implementation

(provide/contract
 ; how many customers are in the db?
 [count    natural-number/c]
 ; is the customer with this id active?
 [active?  (-> id? boolean?)]
 ; what is the name of the customer with this id?
 [name     (-> (and/c id? active?) string?)]
 ; change the name of the customer with this id
 [set-name (->d ([id id?] [nn string?])
                ()
                [result any/c] ; result contract
                #:post-cond
                (string=? (name id) nn))]

 [add      (->d ([bc (and/c basic-customer? not-active?)])
                ()
                ; A pre-post condition contract must use
                ; a side-effect to express this contract
                ; via post-conditions
                #:pre-cond (set! c0 count)
                [result any/c] ; result contract
                #:post-cond (> count c0))])
```

The tests:

```
#lang scheme
(require (planet "test.ss" ("schematics" "schemeunit.plt" 2))
         (planet "text-ui.ss" ("schematics" "schemeunit.plt" 2)))
(require "1.ss" "1b.ss")

(add (make-basic-customer 'mf "matthias" "brookstone"))
(add (make-basic-customer 'rf "robby" "beverly hills park"))
```

```
(add (make-basic-customer 'fl "matthew" "pepper clouds town"))
(add (make-basic-customer 'sk "shriram" "i city"))

(test/text-ui
 (test-suite
  "manager"
  (test-equal? "id lookup" "matthias" (name 'mf))
  (test-equal? "count" 4 count)
  (test-true "active?" (active? 'mf))
  (test-false "active? 2" (active? 'kk))
  (test-true "set name" (void? (set-name 'mf "matt")))))
```

7.5.2 A Parameteric (Simple) Stack

```
#lang scheme

; a contract utility
(define (eq/c x) (lambda (y) (eq? x y)))

(define-struct stack (list p? eq))

(define (initialize p? eq) (make-stack '() p? eq))
(define (push s x)
  (make-stack (cons x (stack-list s)) (stack-p? s) (stack-eq s)))
(define (item-at s i) (list-ref (reverse (stack-list s)) (- i 1)))
(define (count s) (length  (stack-list s)))
(define (is-empty? s) (null? (stack-list s)))
(define not-empty? (compose not is-empty?))
(define (pop s) (make-stack (cdr (stack-list s))
                            (stack-p? s)
                            (stack-eq s)))
(define (top s) (car (stack-list s)))

(provide/contract
 ; predicate
 [stack?     (-> any/c boolean?)]

 ; primitive queries
 ; how many items are on the stack?
 [count      (-> stack? natural-number/c)]

 ; which item is at the given position?
 [item-at
  (->d ([s stack?][i (and/c positive? (<=/c (count s)))]))
```

```
      ()
      [result (stack-p? s)])]

 ; derived queries
 ; is the stack empty?
 [is-empty?
  (->d ([s stack?])
       ()
       [result (eq/c (= (count s) 0))])]

 ; which item is at the top of the stack
 [top
  (->d ([s (and/c stack? not-empty?)])
       ()
       [t (stack-p? s)] ; a stack item, t is its name
       #:post-cond
       ([stack-eq s] t (item-at s (count s))))]

 ; creation
 [initialize
  (->d ([p contract?][s (p p . -> . boolean?)])
       ()
       ; Mitchel and McKim use (= (count s) 0) here to express
       ; the post-condition in terms of a primitive query
       [result (and/c stack? is-empty?)])]

 ; commands
 ; add an item to the top of the stack
 [push
  (->d ([s stack?][x (stack-p? s)])
       ()
       [sn stack?] ; result kind
       #:post-cond
       (and (= (+ (count s) 1) (count sn))
            ([stack-eq s] x (top sn))))]

 ; remove the item at the top of the stack
 [pop
  (->d ([s (and/c stack? not-empty?)])
       ()
       [sn stack?] ; result kind
       #:post-cond
       (= (- (count s) 1) (count sn)))])
```

The tests:

```
#lang scheme
```

```
(require (planet "test.ss" ("schematics" "schemeunit.plt" 2))
         (planet "text-ui.ss" ("schematics" "schemeunit.plt" 2))
         "2.ss")

(define s0 (initialize (flat-contract integer?) =))
(define s2 (push (push s0 2) 1))

(test/text-ui
 (test-suite
  "stack"
  (test-true
   "empty"
   (is-empty? (initialize (flat-contract integer?) =)))
  (test-true "push" (stack? s2))
  (test-true
   "push exn"
   (with-handlers ([exn:fail:contract? (lambda _ #t)])
     (push (initialize (flat-contract integer?)) 'a)
     #f))
  (test-true "pop" (stack? (pop s2)))
  (test-equal? "top" (top s2) 1)
  (test-equal? "toppop" (top (pop s2)) 2)))
```

7.5.3 A Dictionary

```
#lang scheme

; a shorthand for use below
(define-syntax
  (syntax-rules ()
    [( antecedent consequent) (if antecedent consequent #t)]))

; implementation
(define-struct dictionary (l value? eq?))
; the keys should probably be another parameter (exercise)

(define (initialize p eq) (make-dictionary '() p eq))
(define (put d k v)
  (make-dictionary (cons (cons k v) (dictionary-l d))
                   (dictionary-value? d)
                   (dictionary-eq? d)))
(define (rem d k)
  (make-dictionary
   (let loop ([l (dictionary-l d)])
```

```
    (cond
      [(null? l) l]
      [(eq? (caar l) k) (loop (cdr l))]
      [else (cons (car l) (loop (cdr l)))]))
  (dictionary-value? d)
  (dictionary-eq? d)))
(define (count d) (length (dictionary-l d)))
(define (value-for d k) (cdr (assq k (dictionary-l d))))
(define (has? d k) (pair? (assq k (dictionary-l d))))
(define (not-has? d) (lambda (k) (not (has? d k))))
; end of implementation

; interface
(provide/contract
 ; predicates
 [dictionary? (-> any/c boolean?)]
 ; basic queries
 ; how many items are in the dictionary?
 [count       (-> dictionary? natural-number/c)]
 ; does the dictionary define key k?
 [has?        (->d ([d dictionary?][k symbol?])
                   ()
                   [result boolean?]
                   #:post-cond
                   ((zero? (count d)) .  . (not result)))]
 ; what is the value of key k in this dictionary?
 [value-for   (->d ([d dictionary?]
                    [k (and/c symbol? (lambda (k) (has? d k)))])
                   ()
                   [result (dictionary-value? d)])]
 ; initialization
 ; post condition: for all k in symbol, (has? d k) is false.
 [initialize  (->d ([p contract?][eq (p p . -> . boolean?)])
                   ()
                   [result (and/c dictionary? (compose zero? count))])]
 ; commands
 ; Mitchell and McKim say that put shouldn't consume Void (null ptr)
 ; for v. We allow the client to specify a contract for all values
 ; via initialize. We could do the same via a key? parameter
 ; (exercise). add key k with value v to this dictionary
 [put         (->d ([d dictionary?]
                    [k (and symbol? (not-has? d))]
                    [v (dictionary-value? d)])
                   ()
                   [result dictionary?]
                   #:post-cond
```

```
                    (and (has? result k)
                         (= (count d) (- (count result) 1))
                         ([dictionary-eq? d] (value-for result k) v)))]
 ; remove key k from this dictionary
 [rem         (->d ([d dictionary?]
                    [k (and/c symbol? (lambda (k) (has? d k)))])
                   ()
                   [result (and/c dictionary? not-has?)]
                   #:post-cond
                   (= (count d) (+ (count result) 1)))])
; end of interface
```

The tests:

```
#lang scheme
(require (planet "test.ss" ("schematics" "schemeunit.plt" 2))
         (planet "text-ui.ss" ("schematics" "schemeunit.plt" 2))
         "3.ss")

(define d0 (initialize (flat-contract integer?) =))
(define d (put (put (put d0 'a 2) 'b 2) 'c 1))

(test/text-ui
 (test-suite
  "dictionaries"
  (test-equal? "value for" 2 (value-for d 'b))
  (test-false "has?" (has? (rem d 'b) 'b))
  (test-equal? "count" 3 (count d))))
```

7.5.4 A Queue

```
#lang scheme

; Note: this queue doesn't implement the capacity restriction
; of McKim and Mitchell's queue but this is easy to add.

; a contract utility
(define (all-but-last l) (reverse (cdr (reverse l))))
(define (eq/c x) (lambda (y) (eq? x y)))

; implementation
(define-struct queue (list p? eq))

(define (initialize p? eq) (make-queue '() p? eq))
(define items queue-list)
```

```
(define (put q x)
  (make-queue (append (queue-list q) (list x))
              (queue-p? q)
              (queue-eq q)))
(define (count s) (length  (queue-list s)))
(define (is-empty? s) (null? (queue-list s)))
(define not-empty? (compose not is-empty?))
(define (rem s)
  (make-queue (cdr (queue-list s))
              (queue-p? s)
              (queue-eq s)))
(define (head s) (car (queue-list s)))

; interface
(provide/contract
 ; predicate
 [queue?     (-> any/c boolean?)]

 ; primitive queries
 ; Imagine providing this 'query' for the interface of the module
 ; only. Then in Scheme, there is no reason to have count or is-empty?
 ; around (other than providing it to clients). After all items is
 ; exactly as cheap as count.
 [items      (->d ([q queue?]) () [result (listof (queue-p? q))])]

 ; derived queries
 [count      (->d ([q queue?])
                  ; We could express this second part of the post
                  ; condition even if count were a module "attribute"
                  ; in the language of Eiffel; indeed it would use the
                  ; exact same syntax (minus the arrow and domain).
                  ()
                  [result (and/c natural-number/c
                                 (=/c (length (items q))))])]

 [is-empty?  (->d ([q queue?])
                  ()
                  [result (and/c boolean?
                                 (eq/c (null? (items q))))])]

 [head       (->d ([q (and/c queue? (compose not is-empty?))])
                  ()
                  [result (and/c (queue-p? q)
                                 (eq/c (car (items q))))])]
 ; creation
 [initialize (-> contract?
```

```
                (contract? contract? . -> . boolean?)
                (and/c queue? (compose null? items)))]

  ; commands
  [put        (->d ([oldq queue?][i (queue-p? oldq)])
                   ()
                   [result
                    (and/c
                     queue?
                     (lambda (q)
                       (define old-items (items oldq))
                       (equal? (items q) (append old-items (list i))))))])]

  [rem        (->d ([oldq (and/c queue? (compose not is-empty?))])
                   ()
                   [result
                    (and/c queue?
                           (lambda (q)
                             (equal? (cdr (items oldq)) (items q))))])])
; end of interface
```

The tests:

```
#lang scheme
(require (planet "test.ss" ("schematics" "schemeunit.plt" 2))
         (planet "text-ui.ss" ("schematics" "schemeunit.plt" 2))
         "5.ss")

(define s (put (put (initialize (flat-contract integer?) =) 2) 1))

(test/text-ui
 (test-suite
  "queue"
  (test-true
   "empty"
   (is-empty? (initialize (flat-contract integer?) =)))
  (test-true "put" (queue? s))
  (test-equal? "count" 2 (count s))
  (test-true "put exn"
             (with-handlers ([exn:fail:contract? (lambda _ #t)])
               (put (initialize (flat-contract integer?)) 'a)
               #f))
  (test-true "remove" (queue? (rem s)))
  (test-equal? "head" 2 (head s))))
```

7.6 Gotchas

7.6.1 Using `set!` to assign to variables provided via `provide/contract`

The contract library assumes that variables exported via `provide/contract` are not assigned to, but does not enforce it. Accordingly, if you try to `set!` those variables, you may be surprised. Consider the following example:

```
> (module server scheme
    (define (inc-x!) (set! x (+ x 1)))
    (define x 0)
    (provide/contract [inc-x! (-> void?)]
                      [x integer?]))
> (module client scheme
    (require 'server)

    (define (print-latest) (printf "x is ~s\n" x))

    (print-latest)
    (inc-x!)
    (print-latest))
> (require 'client)
x is 0
x is 0
```

Both calls to `print-latest` print `0`, even though the value of `x` has been incremented (and the change is visible inside the module `x`).

To work around this, export accessor functions, rather than exporting the variable directly, like this:

```
#lang scheme

(define (get-x) x)
(define (inc-x!) (set! x (+ x 1)))
(define x 0)
(provide/contract [inc-x! (-> void?)]
                  [get-x (-> integer?)])
```

This is a bug we hope to address in a future release.

8 Input and Output

A Scheme *port* represents an input or output stream, such as a file, a terminal, a TCP connection, or an in-memory string. More specifically, an *input port* represents a stream from which a program can read data, and an *output port* represents a stream for writing data.

8.1 Varieties of Ports

Various functions create various kinds of ports. Here are a few examples:

- **Files:** The `open-output-file` function opens a file for writing, and `open-input-file` opens a file for reading.

Examples:
```
> (define out (open-output-file "data"))
> (display "hello" out)
> (close-output-port out)
> (define in (open-input-file "data"))
> (read-line in)
"hello"
> (close-input-port in)
```

If a file exists already, then `open-output-file` raises an exception by default. Supply an option like `#:exists 'truncate` or `#:exists 'update` to re-write or update the file:

Examples:
```
> (define out (open-output-file "data" #:exists 'truncate))
> (display "howdy" out)
> (close-output-port out)
```

Instead of having to match `open-input-file` and `open-output-file` calls, most Scheme programmers will instead use `call-with-output-file`, which takes a function to call with the output port; when the function returns, the port is closed.

Examples:
```
> (call-with-output-file "data"
                         #:exists 'truncate
                         (lambda (out)
                           (display "hello" out)))
> (call-with-input-file "data"
                        (lambda (in)
                          (read-line in)))
"hello"
```

- **Strings:** The `open-output-string` function creates a port that accumulates data into a string, and `get-output-string` extracts the accumulated string. The `open-input-string` function creates a port to read from a string.

Examples:

```
> (define p (open-output-string))
> (display "hello" p)
> (get-output-string p)
"hello"
> (read-line (open-input-string "goodbye\nfarewell"))
"goodbye"
```

- **TCP Connections:** The `tcp-connect` function creates both an input port and an output port for the client side of a TCP communication. The `tcp-listen` function creates a server, which accepts connections via `tcp-accept`.

Examples:

```
> (define server (tcp-listen 12345))
> (define-values (c-in c-out) (tcp-connect "localhost" 12345))
> (define-values (s-in s-out) (tcp-accept server))
> (display "hello\n" c-out)
> (read-line s-in)
"hello"
> (close-output-port c-out)
> (read-line s-in)
#<eof>
```

- **Process Pipes:** The `subprocess` function runs a new process at the OS level and returns ports that correspond to the subprocess's stdin, stdout, and stderr. (The first three arguments can be certain kinds of existing ports to connect directly to the subprocess, instead of creating new ports.)

Examples:

```
> (define-values (p stdout stdin stderr)
    (subprocess #f #f #f "/usr/bin/wc" "-w"))
> (display "a b c\n" stdin)
> (close-output-port stdin)
> (read-line stdout)
"       3"
> (close-input-port stdout)
> (close-input-port stderr)
```

- **Internal Pipes:** The `make-pipe` function returns two ports that are ends of a pipe. This kind of pipe is internal to Scheme, and not related to OS-level pipes for communicating between different processes.

Examples:

```
> (define-values (in out) (make-pipe))
> (display "garbage" out)
> (close-output-port out)
> (read-line in)
"garbage"
```

8.2 Default Ports

For most simple I/O functions, the target port is an optional argument, and the default is the *current input port* or *current output port*. Furthermore, error messages are written to the *current error port*, which is an output port. The `current-input-port`, `current-output-port`, and `current-error-port` functions return the corresponding current ports.

Examples:

```
> (display "Hi")
Hi
> (display "Hi" (current-output-port)) ; the same
Hi
```

If you start the `mzscheme` program in a terminal, then the current input, output, and error ports are all connected to the terminal. More generally, they are connected to the OS-level stdin, stdout, and stderr. In this guide, the examples show output written to stdout in purple, and output written to stderr in red italics.

Examples:

```
(define (swing-hammer)
  (display "Ouch!" (current-error-port)))

> (swing-hammer)
Ouch!
```

The current-port functions are actually parameters, which means that their values can be set with `parameterize`.

§15.1 "Parameters" (later in this guide) explains more about parameters.

Examples:

```
> (let ([s (open-output-string)])
    (parameterize ([current-error-port s])
      (swing-hammer)
      (swing-hammer)
      (swing-hammer))
    (get-output-string s))
"Ouch!Ouch!Ouch!"
```

8.3 Reading and Writing Scheme Data

As noted throughout §3 “Built-In Datatypes”, Scheme provides two ways to print an instance of a built-in value:

- `write`, which prints a value in the same way that is it printed for a REPL result; and
- `display`, which tends to reduce a value to just its character or byte content—at least for those datatypes that are primarily about characters or bytes, otherwise it falls back to the same output as `write`.

Here are some examples using each:

```
> (write 1/2)
1/2
> (write #\x)
#\x
> (write "hello")
"hello"
> (write #"goodbye")
#"goodbye"
> (write '|dollar sign|)
|dollar sign|
> (write '("alphabet" soup))
("alphabet" soup)
> (write write)
#<procedure:write>
```

```
> (display 1/2)
1/2
> (display #\x)
x
> (display "hello")
hello
> (display #"goodbye")
goodbye
> (display '|dollar sign|)
dollar sign
> (display '("alphabet" soup))
(alphabet soup)
> (display write)
#<procedure:write>
```

The `printf` function supports simple formatting of data and text. In the format string supplied to `printf`, `~a` `display`s the next argument, while `~s` `write`s the next argument.

Examples:

```
(define (deliver who what)
  (printf "Value for ~a: ~s" who what))

> (deliver "John" "string")
Value for John: "string"
```

An advantage of `write`, as opposed to `display`, is that many forms of data can be read back in using `read`.

Examples:

```
> (define-values (in out) (make-pipe))
> (write "hello" out)
> (read in)
```

```
"hello"
> (write '("alphabet" soup) out)
> (read in)
("alphabet" soup)
> (write #hash((a . "apple") (b . "banana")) out)
> (read in)
#hash((a . "apple") (b . "banana"))
```

8.4 Datatypes and Serialization

Prefab structure types (see §5.6 "Prefab Stucture Types") automatically support *serialization*: they can be written to an output stream, and a copy can be read back in from an input stream:

```
> (define-values (in out) (make-pipe))
> (write #s(sprout bean) out)
> (read in)
#s(sprout bean)
```

Other structure types created by `define-struct`, which offer more abstraction than prefab structure types, normally `write` either using `#<....>` notation (for opaque structure types) or using `#(....)` vector notation (for transparent structure types). In neither can can the result be read back in as an instance of the structure type:

```
> (define-struct posn (x y))
> (write (make-posn 1 2))
#<posn>
> (define-values (in out) (make-pipe))
> (write (make-posn 1 2) out)
> (read in)
UNKNOWN::0: read: bad syntax '#<'
```

```
> (define-struct posn (x y) #:transparent)
> (write (make-posn 1 2))
#(struct:posn 1 2)
> (define-values (in out) (make-pipe))
> (write (make-posn 1 2) out)
> (define v (read in))
> v
#(struct:posn 1 2)
> (posn? v)
#f
> (vector? v)
#t
```

The `define-serializable-struct` form defines a structure type that can be `serialize`d

to a value that can be printed using `write` and restored via `read`. The `serialize`d result can be `deserialize`d to get back an instance of the original structure type. The serialization form and functions are provided by the `scheme/serialize` library.

Examples:

```
> (require scheme/serialize)
> (define-serializable-struct posn (x y) #:transparent)
> (deserialize (serialize (make-posn 1 2)))
#(struct:posn 1 2)
> (write (serialize (make-posn 1 2)))
((1) 1 ((#f . deserialize-info:posn-v0)) 0 () () (0 1 2))
> (define-values (in out) (make-pipe))
> (write (serialize (make-posn 1 2)) out)
> (deserialize (read in))
#(struct:posn 1 2)
```

In addition to the names bound by `define-struct`, `define-serializable-struct` binds an identifier with deserialization information, and it automatically `provide`s the deserialization identifier from a module context. This deserialization identifier is accessed reflectively when a value is deserialized.

8.5 Bytes versus Characters

9 Regular Expressions

10 Exceptions and Control

11 Iterations and Comprehensions

The `for` family of syntactic forms support iteration over *sequences*. Lists, vectors, strings, byte strings, input ports, and hash tables can all be used as sequences, and constructors like `in-range` offer even more kinds of sequences.

Variants of `for` accumulate iteration results in different ways, but they all have the same syntactic shape. Simplifying for now, the syntax of `for` is

---

```
(for ([id sequence-expr] ...)
  body ...+)
```

A `for` loop iterates through the sequence produced by the *sequence-expr*. For each element of the sequence, `for` binds the element to *id*, and then it evaluates the *body*s for side effects.

Examples:

```
> (for ([i '(1 2 3)])
    (display i))
123
> (for ([i "abc"])
    (printf "~a..." i))
a...b...c...
```

The `for/list` variant of `for` is more Scheme-like. It accumulates *body* results into a list, instead of evaluating *body* only for side effects. In more technical terms, `for/list` implements a *list comprehension*.

Examples:

```
> (for/list ([i '(1 2 3)])
    (* i i))
(1 4 9)
> (for/list ([i "abc"])
    i)
(#\a #\b #\c)
```

The full syntax of `for` accommodates multiple sequences to iterate in parallel, and the `for*` variant nests the iterations instead of running them in parallel. More variants of `for` and `for*` accumulate *body* results in different ways. In all of these variants, predicates that prune iterations can be included along with bindings.

Before details on the variations of `for`, though, it's best to see the kinds of sequence generators that make interesting examples.

11.1 Sequence Constructors

The `in-range` function generates a sequence of numbers, given an optional starting number (which defaults to `0`), a number before which the sequences ends, and an optional step (which defaults to `1`).

Examples:

```
> (for ([i (in-range 3)])
    (display i))
012
> (for ([i (in-range 1 4)])
    (display i))
123
> (for ([i (in-range 1 4 2)])
    (display i))
13
> (for ([i (in-range 4 1 -1)])
    (display i))
432
> (for ([i (in-range 1 4 1/2)])
    (printf " ~a " i))
 1  3/2  2  5/2  3  7/2
```

The `in-naturals` function is similar, except that the starting number must be an exact non-negative integer (which defaults to `0`), the step is always `1`, and there is no upper limit. A `for` loop using just `in-naturals` will never terminate unless a body expression raises an exception or otherwise escapes.

Examples:

```
> (for ([i (in-naturals)])
    (if (= i 10)
        (error "too much!")
        (display i)))
0123456789
too much!
```

The `stop-before` and `stop-after` functions construct a new sequence given a sequence and a predicate. The new sequence is like the given sequence, but truncated either immediately before or immediately after the first element for which the predicate returns true.

Examples:

```
> (for ([i (stop-before "abc def"
                        char-whitespace?)])
    (display i))
abc
```

Sequence constructors like `in-list`, `in-vector` and `in-string` simply make explicit the use of a list, vector, or string as a sequence. Since they raise an exception when given the wrong kind of value, and since they otherwise avoid a run-time dispatch to determine the sequence type, they enable more efficient code generation; see §11.8 "Iteration Performance" for more information.

Examples:

```
> (for ([i (in-string "abc")])
    (display i))
abc
> (for ([i (in-string '(1 2 3))])
    (display i))
in-string: expected argument of type <string>; given (1 2 3)
```

§3.14 "Sequences" in §"**Reference**: PLT Scheme" provides more on sequences.

11.2 `for` and `for*`

A more complete syntax of `for` is

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

clause = [id sequence-expr]
       | #:when boolean-expr
```

When multiple `[id sequence-expr]` clauses are provided in a `for` form, the corresponding sequences are traversed in parallel:

```
> (for ([i (in-range 1 4)]
        [chapter '("Intro" "Details" "Conclusion")])
    (printf "Chapter ~a. ~a\n" i chapter))
Chapter 1. Intro
Chapter 2. Details
Chapter 3. Conclusion
```

With parallel sequences, the `for` expression stops iterating when any sequence ends. This behavior allows `in-naturals`, which creates an infinite sequence of numbers, to be used for indexing:

```
> (for ([i (in-naturals 1)]
        [chapter '("Intro" "Details" "Conclusion")])
    (printf "Chapter ~a. ~a\n" i chapter))
Chapter 1. Intro
```

```
Chapter 2. Details
Chapter 3. Conclusion
```

The `for*` form, which has the same syntax as `for`, nests multiple sequences instead of running them in parallel:

```
> (for* ([book '("Guide" "Reference")]
         [chapter '("Intro" "Details" "Conclusion")])
    (printf "~a ~a\n" book chapter))
Guide Intro
Guide Details
Guide Conclusion
Reference Intro
Reference Details
Reference Conclusion
```

Thus, `for*` is a shorthand for nested `for`s in the same way that `let*` is a shorthand for nested `let`s.

The `#:when` *`boolean-expr`* form of a *`clause`* is another shorthand. It allows the *`body`*s to evaluate only when the *`boolean-expr`* produces a true value:

```
> (for* ([book '("Guide" "Reference")]
         [chapter '("Intro" "Details" "Conclusion")]
         #:when (not (equal? chapter "Details")))
    (printf "~a ~a\n" book chapter))
Guide Intro
Guide Conclusion
Reference Intro
Reference Conclusion
```

A *`boolean-expr`* with `#:when` can refer to any of the preceding iteration bindings. In a `for` form, this scoping makes sense only if the test is nested in the iteration of the preceding bindings; thus, bindings separated by `#:when` are mutually nested, instead of in parallel, even with `for`.

```
> (for ([book '("Guide" "Reference" "Notes")]
        #:when (not (equal? book "Notes"))
        [i (in-naturals 1)]
        [chapter '("Intro" "Details" "Conclusion" "Index")]
        #:when (not (equal? chapter "Index")))
    (printf "~a Chapter ~a. ~a\n" book i chapter))
Guide Chapter 1. Intro
Guide Chapter 2. Details
Guide Chapter 3. Conclusion
Reference Chapter 1. Intro
Reference Chapter 2. Details
```

11.3 `for/list` and `for*/list`

The `for/list` form, which has the same syntax as `for`, evaluates the *body*s to obtain values that go into a newly constructed list:

```
> (for/list ([i (in-naturals 1)]
             [chapter '("Intro" "Details" "Conclusion")])
    (string-append (number->string i) ". " chapter))
("1. Intro" "2. Details" "3. Conclusion")
```

A `#:when` clause in a `for-list` form prunes the result list along with evaluations of the *body*s:

```
> (for/list ([i (in-naturals 1)]
             [chapter '("Intro" "Details" "Conclusion")]
             #:when (odd? i))
    chapter)
("Intro" "Conclusion")
```

This pruning behavior of `#:when` is more useful with `for/list` than `for`. Whereas a plain `when` form normally suffices with `for`, a `when` expression form in a `for/list` would cause the result list to contain `#<void>`s instead of omitting list elements.

The `for*/list` is like `for*`, nesting multiple iterations:

```
> (for*/list ([book '("Guide" "Ref.")]
              [chapter '("Intro" "Details")])
    (string-append book " " chapter))
("Guide Intro" "Guide Details" "Ref. Intro" "Ref. Details")
```

A `for*/list` form is not quite the same thing as nested `for/list` forms. Nested `for/list`s would produce a list of lists, instead of one flattened list. Much like `#:when`, then, the nesting of `for*/list` is more useful than the nesting of `for*`.

11.4 `for/and` and `for/or`

The `for/and` form combines iteration results with `and`, stopping as soon as it encounters `#f`:

```
> (for/and ([chapter '("Intro" "Details" "Conclusion")])
    (equal? chapter "Intro"))
#f
```

The `for/or` form combines iteration results with `or`, stopping as soon as it encounters a true value:

```
> (for/or ([chapter '("Intro" "Details" "Conclusion")])
    (equal? chapter "Intro"))
#t
```

As usual, the `for*/and` and `for*/or` forms provide the same facility with nested iterations.

11.5 `for/first` and `for/last`

The `for/first` form returns the result of the first time that the *body*s are evaluated, skipping further iterations. This form is most useful with a `#:when` clause.

```
> (for/first ([chapter '("Intro" "Details" "Conclusion" "Index")]
              #:when (not (equal? chapter "Intro")))
    chapter)
"Details"
```

If the *body*s are evaluated zero times, then the result is `#f`.

The `for/last` form runs all iterations, returning the value of the last iteration (or `#f` if no iterations are run):

```
> (for/last ([chapter '("Intro" "Details" "Conclusion" "Index")]
              #:when (not (equal? chapter "Index")))
    chapter)
"Conclusion"
```

As usual, the `for*/first` and `for*/last` forms provide the same facility with nested iterations:

```
> (for*/first ([book '("Guide" "Reference")]
               [chapter '("Intro" "Details" "Conclusion" "Index")]
               #:when (not (equal? chapter "Intro")))
    (list book chapter))
("Guide" "Details")
> (for*/last ([book '("Guide" "Reference")]
              [chapter '("Intro" "Details" "Conclusion" "Index")]
              #:when (not (equal? chapter "Index")))
    (list book chapter))
("Reference" "Conclusion")
```

11.6 `for/fold` and `for*/fold`

The `for/fold` form is a very general way to combine iteration results. Its syntax is slightly different than the syntax of `for`, because accumulation variables must be declared at the beginning:

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

In the simple case, only one `[accum-id init-expr]` is provided, and the result of the `for/fold` is the final value for *accum-id*, which starts out with the value of *init-expr*. In the *clause*s and *body*s, *accum-id* can be referenced to get its current value, and the last *body* provides the value of *accum-id* for the next iteration.

Examples:

```
> (for/fold ([len 0])
            ([chapter '("Intro" "Conclusion")])
    (+ len (string-length chapter)))
15
> (for/fold ([prev #f])
            ([i (in-naturals 1)]
             [chapter '("Intro" "Details" "Details" "Conclusion")]
             #:when (not (equal? chapter prev)))
    (printf "~a. ~a\n" i chapter)
    chapter)
1. Intro
2. Details
4. Conclusion
"Conclusion"
```

When multiple *accum-id*s are specified, then the last *body* must produce multiple values, one for each *accum-id*. The `for/fold` expression itself produces multiple values for the results.

Examples:

```
> (for/fold ([prev #f]
             [counter 1])
            ([chapter '("Intro" "Details" "Details" "Conclusion")]
             #:when (not (equal? chapter prev)))
    (printf "~a. ~a\n" counter chapter)
    (values chapter
            (add1 counter)))
1. Intro
2. Details
3. Conclusion
```

```
"Conclusion"
4
```

11.7 Multiple-Valued Sequences

In the same way that a function or expression can produce multiple values, individual iterations of a sequence can produce multiple elements. For example, a hash table as a sequence generates two values for each iteration: a key and a value.

In the same way that `let-values` binds multiple results to multiple identifiers, `for` can bind multiple sequence elements to multiple iteration identifiers:

While `let` must be changed to `let-values` to bind multiple identifier, `for` simply allows a parenthesized list of identifiers instead of a single identifier in any clause.

```
> (for ([(k v) #hash(("apple" . 1) ("banana" . 3))])
    (printf "~a count: ~a\n" k v))
apple count: 1
banana count: 3
```

This extension to multiple-value bindings works for all `for` variants. For example, `for*/list` nests iterations, builds a list, and also works with multiple-valued sequences:

```
> (for*/list ([(k v) #hash(("apple" . 1) ("banana" . 3))]
              [(i) (in-range v)])
    k)
("apple" "banana" "banana" "banana")
```

11.8 Iteration Performance

Ideally, a `for` iteration should run as fast as a loop that you write by hand as a recursive-function invocation. A hand-written loop, however, is normally specific to a particular kind of data, such as lists. In that case, the hand-written loop uses selectors like `car` and `cdr` directly, instead of handling all forms of sequences and dispatching to an appropriate iterator.

The `for` forms can provide the performance of hand-written loops when enough information is apparent about the sequences to iterate. Specifically, the clause should have one of the following *`fast-clause`* forms:

```
fast-clause = [id fast-seq]
            | [(id) fast-seq]
            | [(id id) fast-indexed-seq]
            | [(id ...) fast-parallel-seq]

fast-seq = (in-range expr expr)
         | (in-range expr expr expr)
         | (in-naturals)
```

```
          | (in-naturals expr)
          | (in-list expr)
          | (in-vector expr)
          | (in-string expr)
          | (in-bytes expr)
          | (stop-before fast-seq predicate-expr)
          | (stop-after fast-seq predicate-expr)

fast-indexed-seq = (in-indexed fast-seq)
                 | (stop-before fast-indexed-seq predicate-expr)
                 | (stop-after fast-indexed-seq predicate-expr)

fast-parallel-seq = (in-parallel fast-seq ...)
                  | (stop-before fast-parallel-seq predicate-expr)
                  | (stop-after fast-parallel-seq predicate-expr)
```

Examples:

```
> (time (for ([i (in-range 100000)])
          (for ([elem (in-list '(a b c d e f g h))]) ; fast
            (void))))
cpu time: 5 real time: 6 gc time: 0
> (time (for ([i (in-range 100000)])
          (for ([elem '(a b c d e f g h)])           ; slower
            (void))))
cpu time: 69 real time: 68 gc time: 0
> (time (let ([seq (in-list '(a b c d e f g h))])
          (for ([i (in-range 100000)])
            (for ([elem seq])                       ; slower
              (void)))))
cpu time: 71 real time: 71 gc time: 0
```

The grammars above are not complete, because the set of syntactic patterns that provide good performance is extensible, just like the set of sequence values. The documentation for a sequence constructor should indicate the performance benefits of using it directly in a `for` *clause*.

§2.16 “Iterations and Comprehensions: `for`, `for/list`, ...” in §“**Reference**: PLT Scheme” provides more on iterations and comprehensions.

12 Pattern Matching

The `match` form supports pattern matching on arbitrary Scheme values, as opposed to functions like `regexp-match` that compare regular expressions to byte and character sequences (see §9 “Regular Expressions”).

13 Classes and Objects

This section is based on a paper [Flatt06].

A `class` expression denotes a first-class value, just like a `lambda` expression:

```
(class superclass-expr decl-or-expr ...)
```

The *superclass-expr* determines the superclass for the new class. Each *decl-or-expr* is either a declaration related to methods, fields, and initialization arguments, or it is an expression that is evaluated each time that the class is instantiated. In other words, instead of a method-like constructor, a class has initialization expressions interleaved with field and method declarations.

By convention, class names end with `%`. The built-in root class is `object%`. The following expression creates a class with public methods `get-size`, `grow`, and `eat`:

```
(class object%
  (init size)                ; initialization argument

  (define current-size size) ; field

  (super-new)                ; superclass initialization

  (define/public (get-size)
    current-size)

  (define/public (grow amt)
    (set! current-size (+ amt current-size)))

  (define/public (eat other-fish)
    (grow (send other-fish get-size))))
```

The `size` initialization argument must be supplied via a named argument when instantiating the class through the `new` form:

```
(new (class object% (init size) ....) [size 10])
```

Of course, we can also name the class and its instance:

```
(define fish% (class object% (init size) ....))
(define charlie (new fish% [size 10]))
```

In the definition of `fish%`, `current-size` is a private field that starts out with the value of the `size` initialization argument. Initialization arguments like `size` are available only during class instantiation, so they cannot be referenced directly from a method. The `current-size`

field, in contrast, is available to methods.

The `(super-new)` expression in `fish%` invokes the initialization of the superclass. In this case, the superclass is `object%`, which takes no initialization arguments and performs no work; `super-new` must be used, anyway, because a class must always invoke its superclass's initialization.

Initialization arguments, field declarations, and expressions such as `(super-new)` can appear in any order within a `class`, and they can be interleaved with method declarations. The relative order of expressions in the class determines the order of evaluation during instantiation. For example, if a field's initial value requires calling a method that works only after superclass initialization, then the field declaration must be placed after the `super-new` call. Ordering field and initialization declarations in this way helps avoid imperative assignment. The relative order of method declarations makes no difference for evaluation, because methods are fully defined before a class is instantiated.

13.1 Methods

Each of the three `define/public` declarations in `fish%` introduces a new method. The declaration uses the same syntax as a Scheme function, but a method is not accessible as an independent function. A call to the `grow` method of a `fish%` object requires the `send` form:

```
> (send charlie grow 6)
> (send charlie get-size)
16
```

Within `fish%`, self methods can be called like functions, because the method names are in scope. For example, the `eat` method within `fish%` directly invokes the `grow` method. Within a class, attempting to use a method name in any way other than a method call results in a syntax error.

In some cases, a class must call methods that are supplied by the superclass but not overridden. In that case, the class can use `send` with `this` to access the method:

```
(define hungry-fish% (class fish% (super-new)
                       (define/public (eat-more fish1 fish2)
                         (send this eat fish1)
                         (send this eat fish2))))
```

Alternately, the class can declare the existence of a method using `inherit`, which brings the method name into scope for a direct call:

```
(define hungry-fish% (class fish% (super-new)
                       (inherit eat)
                       (define/public (eat-more fish1 fish2)
                         (eat fish1) (eat fish2))))
```

With the `inherit` declaration, if `fish%` had not provided an `eat` method, an error would be signaled in the evaluation of the `class` form for `hungry-fish%`. In contrast, with `(send this ....)`, an error would not be signaled until the `eat-more` method is called and the `send` form is evaluated. For this reason, `inherit` is preferred.

Another drawback of `send` is that it is less efficient than `inherit`. Invocation of a method via `send` involves finding a method in the target object's class at run time, making `send` comparable to an interface-based method call in Java. In contrast, `inherit`-based method invocations use an offset within the class's method table that is computed when the class is created.

To achieve performance similar to `inherit`-based method calls when invoking a method from outside the method's class, the programmer must use the `generic` form, which produces a class- and method-specific *generic method* to be invoked with `send-generic`:

```
(define get-fish-size (generic fish% get-size))

> (send-generic charlie get-fish-size)
16
> (send-generic (new hungry-fish% [size 32]) get-fish-size)
32
> (send-generic (new object%) get-fish-size)
generic:get-size for class: fish%: expected argument of
type <instance for class: fish%>; given #(struct:object)
```

Roughly speaking, the form translates the class and the external method name to a location in the class's method table. As illustrated by the last example, sending through a generic method checks that its argument is an instance of the generic's class.

Whether a method is called directly within a `class`, through a generic method, or through `send`, method overriding works in the usual way:

```
(define picky-fish% (class fish% (super-new)
                      (define/override (grow amt)

                        (super grow (* 3/4 amt)))))
(define daisy (new picky-fish% [size 20]))

> (send daisy eat charlie)
> (send daisy get-size)
32
```

The `grow` method in `picky-fish%` is declared with `define/override` instead of `define/public`, because `grow` is meant as an overriding declaration. If `grow` had been declared with `define/public`, an error would have been signaled when evaluating the `class` expression, because `fish%` already supplies `grow`.

Using `define/override` also allows the invocation of the overridden method via a `super` call. For example, the `grow` implementation in `picky-fish%` uses `super` to delegate to the superclass implementation.

13.2 Initialization Arguments

Since `picky-fish%` declares no initialization arguments, any initialization values supplied in `(new picky-fish% ....)` are propagated to the superclass initialization, i.e., to `fish%`. A subclass can supply additional initialization arguments for its superclass in a `super-new` call, and such initialization arguments take precedence over arguments supplied to `new`. For example, the following `size-10-fish%` class always generates fish of size 10:

```
(define size-10-fish% (class fish% (super-new [size 10])))

> (send (new size-10-fish%) get-size)
10
```

In the case of `size-10-fish%`, supplying a `size` initialization argument with `new` would result in an initialization error; because the `size` in `super-new` takes precedence, a `size` supplied to `new` would have no target declaration.

An initialization argument is optional if the `class` form declares a default value. For example, the following `default-10-fish%` class accepts a `size` initialization argument, but its value defaults to 10 if no value is supplied on instantiation:

```
(define default-10-fish% (class fish%
                           (init [size 10])
                           (super-new [size size])))

> (new default-10-fish%)
#(struct:object:default-10-fish% ...)
> (new default-10-fish% [size 20])
#(struct:object:default-10-fish% ...)
```

In this example, the `super-new` call propagates its own `size` value as the `size` initialization argument to the superclass.

13.3 Internal and External Names

The two uses of `size` in `default-10-fish%` expose the double life of class-member identifiers. When `size` is the first identifier of a bracketed pair in `new` or `super-new`, `size` is an *external name* that is symbolically matched to an initialization argument in a class. When `size` appears as an expression within `default-10-fish%`, `size` is an *internal name* that is

lexically scoped. Similarly, a call to an inherited `eat` method uses `eat` as an internal name, whereas a `send` of `eat` uses `eat` as an external name.

The full syntax of the `class` form allows a programmer to specify distinct internal and external names for a class member. Since internal names are local, they can be renamed to avoid shadowing or conflicts. Such renaming is not frequently necessary, but workarounds in the absence of renaming can be especially cumbersome.

13.4 Interfaces

Interfaces are useful for checking that an object or a class implements a set of methods with a particular (implied) behavior. This use of interfaces is helpful even without a static type system (which is the main reason that Java has interfaces).

An interface in PLT Scheme is created using the `interface` form, which merely declares the method names required to implement the interface. An interface can extend other interfaces, which means that implementations of the interface automatically implement the extended interfaces.

```
(interface (superinterface-expr ...) id ...)
```

To declare that a class implements an interface, the `class*` form must be used instead of `class`:

```
(class* superclass-expr (interface-expr ...) decl-or-expr ...)
```

For example, instead of forcing all fish classes to be derived from `fish%`, we can define `fish-interface` and change the `fish%` class to declare that it implements `fish-interface`:

```
(define fish-interface (interface () get-size grow eat))
(define fish% (class* object% (fish-interface) ....))
```

If the definition of `fish%` does not include `get-size`, `grow`, and `eat` methods, then an error is signaled in the evaluation of the `class*` form, because implementing the `fish-interface` interface requires those methods.

The `is-a?` predicate accepts either a class or interface as its first argument and an object as its second argument. When given a class, `is-a?` checks whether the object is an instance of that class or a derived class. When given an interface, `is-a?` checks whether the object's

class implements the interface. In addition, the `implementation?` predicate checks whether a given class implements a given interface.

13.5 Final, Augment, and Inner

As in Java, a method in a `class` form can be specified as *final*, which means that a subclass cannot override the method. A final method is declared using `public-final` or `override-final`, depending on whether the declaration is for a new method or an overriding implementation.

Between the extremes of allowing arbitrary overriding and disallowing overriding entirely, the class system also supports Beta-style *augmentable* methods [Goldberg04]. A method declared with `pubment` is like `public`, but the method cannot be overridden in subclasses; it can be augmented only. A `pubment` method must explicitly invoke an augmentation (if any) using `inner`; a subclass augments the method using `augment`, instead of `override`.

In general, a method can switch between augment and override modes in a class derivation. The `augride` method specification indicates an augmentation to a method where the augmentation is itself overrideable in subclasses (though the superclass's implementation cannot be overridden). Similarly, `overment` overrides a method and makes the overriding implementation augmentable.

13.6 Controlling the Scope of External Names

As noted in §13.3 "Internal and External Names", class members have both internal and external names. A member definition binds an internal name locally, and this binding can be locally renamed. External names, in contrast, have global scope by default, and a member definition does not bind an external name. Instead, a member definition refers to an existing binding for an external name, where the member name is bound to a *member key*; a class ultimately maps member keys to methods, fields, and initialization arguments.

Recall the `hungry-fish%` `class` expression:

```
(define hungry-fish% (class fish% ....
                       (inherit eat)
                       (define/public (eat-more fish1 fish2)
                         (eat fish1) (eat fish2))))
```

During its evaluation, the `hungry-fish%` and `fish%` classes refer to the same global binding of `eat`. At run time, calls to `eat` in `hungry-fish%` are matched with the `eat` method in `fish%` through the shared method key that is bound to `eat`.

The default binding for an external name is global, but a programmer can introduce an external-name binding with the `define-member-name` form.

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

In particular, by using `(generate-member-key)` as the `member-key-expr`, an external name can be localized for a particular scope, because the generated member key is inaccessible outside the scope. In other words, `define-member-name` gives an external name a kind of package-private scope, but generalized from packages to arbitrary binding scopes in Scheme.

For example, the following `fish%` and `pond%` classes cooperate via a `get-depth` method that is only accessible to the cooperating classes:

```
(define-values (fish% pond%) ; two mutually recursive classes
  (let ()
    (define-member-name get-depth (generate-member-key))
    (define fish%
      (class ....
        (define my-depth ....)
        (define my-pond ....)
        (define/public (dive amt)
        (set! my-depth
              (min (+ my-depth amt)
                   (send my-pond get-depth))))))
    (define pond%
      (class ....
        (define current-depth ....)
        (define/public (get-depth) current-depth)))
    (values fish% pond%)))
```

External names are in a namespace that separates them from other Scheme names. This separate namespace is implicitly used for the method name in `send`, for initialization-argument names in `new`, or for the external name in a member definition. The special form `member-name-key` provides access to the binding of an external name in an arbitrary expression position: `(member-name-key id)` produces the member-key binding of `id` in the current scope.

A member-key value is primarily used with a `define-member-name` form. Normally, then, `(member-name-key id)` captures the method key of `id` so that it can be communicated to a use of `define-member-name` in a different scope. This capability turns out to be useful for generalizing mixins, as discussed next.

13.7 Mixins

Since `class` is an expression form instead of a top-level declaration as in Smalltalk and Java, a `class` form can be nested inside any lexical scope, including `lambda`. The result is a *mixin*, i.e., a class extension that is parameterized with respect to its superclass.

For example, we can parameterize the `picky-fish%` class over its superclass to define `picky-mixin`:

```
(define (picky-mixin %)
  (class % (super-new)
    (define/override (grow amt) (super grow (* 3/4 amt)))))
(define picky-fish% (picky-mixin fish%))
```

Many small differences between Smalltalk-style classes and Scheme classes contribute to the effective use of mixins. In particular, the use of `define/override` makes explicit that `picky-mixin` expects a class with a `grow` method. If `picky-mixin` is applied to a class without a `grow` method, an error is signaled as soon as `picky-mixin` is applied.

Similarly, a use of `inherit` enforces a "method existence" requirement when the mixin is applied:

```
(define (hungry-mixin %)
  (class % (super-new)
    (inherit eat)
    (define/public (eat-more fish1 fish2)
      (eat fish1)
      (eat fish2))))
```

The advantage of mixins is that we can easily combine them to create new classes whose implementation sharing does not fit into a single-inheritance hierarchy—without the ambiguities associated with multiple inheritance. Equipped with `picky-mixin` and `hungry-mixin`, creating a class for a hungry, yet picky fish is straightforward:

```
(define picky-hungry-fish%
  (hungry-mixin (picky-mixin fish%)))
```

The use of keyword initialization arguments is critical for the easy use of mixins. For example, `picky-mixin` and `hungry-mixin` can augment any class with suitable `eat` and `grow` methods, because they do not specify initialization arguments and add none in their `super-new` expressions:

```
(define person%
  (class object%
    (init name age)
    ....
    (define/public (eat food) ....)
```

```
    (define/public (grow amt) ....)))
(define child% (hungry-mixin (picky-mixin person%)))
(define oliver (new child% [name "Oliver"][age 6]))
```

Finally, the use of external names for class members (instead of lexically scoped identifiers) makes mixin use convenient. Applying `picky-mixin` to `person%` works because the names `eat` and `grow` match, without any a priori declaration that `eat` and `grow` should be the same method in `fish%` and `person%`. This feature is a potential drawback when member names collide accidentally; some accidental collisions can be corrected by limiting the scope external names, as discussed in §13.6 "Controlling the Scope of External Names".

13.7.1 Mixins and Interfaces

Using `implementation?`, `picky-mixin` could require that its base class implements `grower-interface`, which could be implemented by both `fish%` and `person%`:

```
(define grower-interface (interface () grow))
(define (picky-mixin %)
  (unless (implementation? % grower-interface)
    (error "picky-mixin: not a grower-interface class"))
  (class % ....))
```

Another use of interfaces with a mixin is to tag classes generated by the mixin, so that instances of the mixin can be recognized. In other words, `is-a?` cannot work on a mixin represented as a function, but it can recognize an interface (somewhat like a *specialization interface*) that is consistently implemented by the mixin. For example, classes generated by `picky-mixin` could be tagged with `picky-interface`, enabling the `is-picky?` predicate:

```
(define picky-interface (interface ()))
(define (picky-mixin %)
  (unless (implementation? % grower-interface)
    (error "picky-mixin: not a grower-interface class"))
  (class* % (picky-interface) ....))
(define (is-picky? o)
  (is-a? o picky-interface))
```

13.7.2 The `mixin` Form

To codify the `lambda`-plus-`class` pattern for implementing mixins, including the use of interfaces for the domain and range of the mixin, the class system provides a `mixin` macro:

```
(mixin (interface-expr ...) (interface-expr ...)
  decl-or-expr ...)
```

The first set of `interface-expr`s determines the domain of the mixin, and the second set determines the range. That is, the expansion is a function that tests whether a given base class implements the first sequence of `interface-expr`s and produces a class that implements the second sequence of `interface-expr`s. Other requirements, such as the presence of `inherit`ed methods in the superclass, are then checked for the `class` expansion of the `mixin` form.

Mixins not only override methods and introduce public methods, they can also augment methods, introduce augment-only methods, add an overrideable augmentation, and add an augmentable override — all of the things that a class can do (see §13.5 "Final, Augment, and Inner").

13.7.3 Parameterized Mixins

As noted in §13.6 "Controlling the Scope of External Names", external names can be bound with `define-member-name`. This facility allows a mixin to be generalized with respect to the methods that it defines and uses. For example, we can parameterize `hungry-mixin` with respect to the external member key for `eat`:

```
(define (make-hungry-mixin eat-method-key)
  (define-member-name eat eat-method-key)
  (mixin () () (super-new)
    (inherit eat)
    (define/public (eat-more x y) (eat x) (eat y))))
```

To obtain a particular hungry-mixin, we must apply this function to a member key that refers to a suitable `eat` method, which we can obtain using `member-name-key`:

```
((make-hungry-mixin (member-name-key eat))
 (class object% .... (define/public (eat x) 'yum)))
```

Above, we apply `hungry-mixin` to an anonymous class that provides `eat`, but we can also combine it with a class that provides `chomp`, instead:

```
((make-hungry-mixin (member-name-key chomp))
 (class object% .... (define/public (chomp x) 'yum)))
```

13.8 Traits

A *trait* is similar to a mixin, in that it encapsulates a set of methods to be added to a class. A trait is different from a mixin in that its individual methods can be manipulated with trait operators such as `trait-sum` (merge the methods of two traits), `trait-exclude` (remove a method from a trait), and `trait-alias` (add a copy of a method with a new name; do not redirect any calls to the old name).

The practical difference between mixins and traits is that two traits can be combined, even if they include a common method and even if neither method can sensibly override the other. In that case, the programmer must explicitly resolve the collision, usually by aliasing methods, excluding methods, and merging a new trait that uses the aliases.

Suppose our `fish%` programmer wants to define two class extensions, `spots` and `stripes`, each of which includes a `get-color` method. The fish's spot color should not override the stripe color nor vice-versa; instead, a `spots+stripes-fish%` should combine the two colors, which is not possible if `spots` and `stripes` are implemented as plain mixins. If, however, `spots` and `stripes` are implemented as traits, they can be combined. First, we alias `get-color` in each trait to a non-conflicting name. Second, the `get-color` methods are removed from both and the traits with only aliases are merged. Finally, the new trait is used to create a class that introduces its own `get-color` method based on the two aliases, producing the desired `spots+stripes` extension.

13.8.1 Traits as Sets of Mixins

One natural approach to implementing traits in PLT Scheme is as a set of mixins, with one mixin per trait method. For example, we might attempt to define the spots and stripes traits as follows, using association lists to represent sets:

```
(define spots-trait
  (list (cons 'get-color
              (lambda (%) (class % (super-new)
                                 (define/public (get-color)
                                   'black))))))
(define stripes-trait
  (list (cons 'get-color
              (lambda (%) (class % (super-new)
                                 (define/public (get-color)
                                   'red))))))
```

A set representation, such as the above, allows `trait-sum` and `trait-exclude` as simple manipulations; unfortunately, it does not support the `trait-alias` operator. Although a mixin can be duplicated in the association list, the mixin has a fixed method name, e.g., `get-color`, and mixins do not support a method-rename operation. To support `trait-alias`, we must parameterize the mixins over the external method name in the same way

that `eat` was parameterized in §13.7.3 “Parameterized Mixins”.

To support the `trait-alias` operation, `spots-trait` should be represented as:

```
(define spots-trait
  (list (cons (member-name-key get-color)
              (lambda (get-color-key %)
                (define-member-name get-color get-color-key)
                (class % (super-new)
                  (define/public (get-color) 'black))))))
```

When the `get-color` method in `spots-trait` is aliased to `get-trait-color` and the `get-color` method is removed, the resulting trait is the same as

```
(list (cons (member-name-key get-trait-color)
            (lambda (get-color-key %)
              (define-member-name get-color get-color-key)
              (class % (super-new)
                (define/public (get-color) 'black)))))
```

To apply a trait *T* to a class *C* and obtain a derived class, we use `((trait->mixin T) C)`. The `trait->mixin` function supplies each mixin of *T* with the key for the mixin’s method and a partial extension of *C*:

```
(define ((trait->mixin T) C)
  (foldr (lambda (m %) ((cdr m) (car m) %)) C T))
```

Thus, when the trait above is combined with other traits and then applied to a class, the use of `get-color` becomes a reference to the external name `get-trait-color`.

13.8.2 Inherit and Super in Traits

This first implementation of traits supports `trait-alias`, and it supports a trait method that calls itself, but it does not support trait methods that call each other. In particular, suppose that a spot-fish’s market value depends on the color of its spots:

```
(define spots-trait
  (list (cons (member-name-key get-color) ....)
        (cons (member-name-key get-price)
              (lambda (get-price %) ....
                (class % ....
                  (define/public (get-price)
                    .... (get-color) ....))))))
```

In this case, the definition of `spots-trait` fails, because `get-color` is not in scope for the `get-price` mixin. Indeed, depending on the order of mixin application when the trait is

applied to a class, the `get-color` method may not be available when `get-price` mixin is applied to the class. Therefore adding an `(inherit get-color)` declaration to the `get-price` mixin does not solve the problem.

One solution is to require the use of `(send this get-color)` in methods such as `get-price`. This change works because `send` always delays the method lookup until the method call is evaluated. The delayed lookup is more expensive than a direct call, however. Worse, it also delays checking whether a `get-color` method even exists.

A second, effective, and efficient solution is to change the encoding of traits. Specifically, we represent each method as a pair of mixins: one that introduces the method and one that implements it. When a trait is applied to a class, all of the method-introducing mixins are applied first. Then the method-implementing mixins can use `inherit` to directly access any introduced method.

```
(define spots-trait
  (list (list (local-member-name-key get-color)
              (lambda (get-color get-price %) ....
                (class % ....
                  (define/public (get-color) (void))))
              (lambda (get-color get-price %) ....
                 (class % ....
                   (define/override (get-color) 'black))))
        (list (local-member-name-key get-price)
              (lambda (get-price get-color %) ....
                (class % ....
                  (define/public (get-price) (void))))
              (lambda (get-color get-price %) ....
                (class % ....
                  (inherit get-color)
                  (define/override (get-price)
                    .... (get-color) ....))))))
```

With this trait encoding, `trait-alias` adds a new method with a new name, but it does not change any references to the old method.

13.8.3 The `trait` Form

The general-purpose trait pattern is clearly too complex for a programmer to use directly, but it is easily codified in a `trait` macro:

```
(trait trait-clause ...)
```

The ids in the optional inherit clause are available for direct reference in the method exprs, and they must be supplied either by other traits or the base class to which the trait is ultimately applied.

Using this form in conjunction with trait operators such as trait-sum, trait-exclude, trait-alias, and trait->mixin, we can implement spots-trait and stripes-trait as desired.

```
(define spots-trait
  (trait
    (define/public (get-color) 'black)
    (define/public (get-price) ... (get-color) ...)))

(define stripes-trait
  (trait
    (define/public (get-color) 'red)))

(define spots+stripes-trait
  (trait-sum
   (trait-exclude (trait-alias spots-trait
                               get-color get-spots-color)
                  get-color)
   (trait-exclude (trait-alias stripes-trait
                               get-color get-stripes-color)
                  get-color)
   (trait
     (inherit get-spots-color get-stripes-color)
     (define/public (get-color)
       .... (get-spots-color) .... (get-stripes-color) ....))))
```

14 Units (Components)

15 Threads

15.1 Parameters

A *parameter* holds a kind of global option. For example, there is a parameter that determines the default destination for printed output.

16 Reflection and Dynamic Evaluation

Scheme is a *dynamic* language. It offers numerous facilities for loading, compiling, and even constructing new code at run time.

16.1 Namespaces

Dynamic evaluation requires a *namespace*, which encapsulates two pieces of information:

- A mapping from identifiers to bindings. For example, a namespace might map the identifier `lambda` to the `lambda` form. An "empty" namespace is one that maps every identifier to an uninitialized top-level variable.
- A mapping from module names to module declarations and instances.

The first mapping is used for evaluating expressions in a top-level context, as in `(eval '(lambda (x) (+ x 1)))`. The second mapping is used, for example, by `dynamic-require` to locate a module. The call `(eval '(require scheme/base))` normally uses both pieces: the identifier mapping determines the binding of `require`; if it turns out to mean `require`, then the module mapping is used to locate the `scheme/base` module.

From the perspective of the core Scheme run-time system, all evaluation is reflective. Execution starts with an initial namespace that contains a few primitive modules, and that is further populated by loading files and modules as specified on the command line or as supplied in the REPL. Top-level `require` and `define` forms adjusts the identifier mapping, and module declarations (typically loaded on demand for a `require` form) adjust the module mapping.

Informally, the term *namespace* is sometimes used interchangeably with *environment* or *scope*. In PLT Scheme, the term *namespace* has the more specific, dynamic meaning given above, and it should not be confused with static lexical concepts.

16.2 Creating and Installing Namespaces

A namespace is a first-class value. Some functions, such as `eval`, accept an optional namespace argument. More often, the namespace used by a dynamic operation is the *current namespace* as determined by the `current-namespace` parameter.

The function `make-empty-namespace` creates a new, empty namespace. Since the namespace is truly empty, it cannot at first be used to evaluate any top-level expression—not even `(require scheme)`. In particular,

```
(parameterize ([current-namespace (make-empty-namespace)])
  (namespace-require 'scheme))
```

fails, because the namespace does not include the primitive modules on which `scheme` is built.

To make a namespace useful, some modules much be *attached* from an existing namespace. Attaching a module adjust the mapping of module names to instances by transitively copying entries (the module and all its imports) from an existing namespace's mapping. Normally, instead of just attaching the primitive modules—whose names and organization are subject to change—a higher-level module is attached, such as `scheme` or `scheme/base`.

The `make-base-empty-namespace` function provides a namespace that is empty, except that `scheme/base` is attached. The resulting namespace is still "empty" in the sense that the identifiers-to-bindings part of the namespace has no mappings; only the module mapping has been populated. Nevertheless, with an initial module mapping, further modules can be loaded.

A namespace created with `make-base-empty-namespace` is suitable for many basic dynamic tasks. For example, suppose that a `my-dsl` library implements a domain-specific language in which you want to execute commands from a user-specified file. A namespace created with `make-base-empty-namespace` is enough to get started:

```
(define (run-dsl file)
  (parameterize ([current-namespace (make-base-empty-namespace)])
    (namespace-require 'my-dsl)
    (load file)))
```

Note that the `parameterize` of `current-namespace` does not affect the meaning of identifiers like `namespace-require` within the `parameterize` body. Those identifiers obtain their meaning from the enclosing context (probably a module). Only expressions that are dynamic with respect to this code, such as the content of `load`ed files, are affected by the `parameterize`.

Another subtle point in the above example is the use of `(namespace-require 'my-dsl)` instead of `(eval '(require my-dsl))`. The latter would not work, because `eval` needs to obtain a meaning for `require` in the namespace, and the namespace's identifier mapping is initially empty. The `namespace-require` function, in contrast, directly imports the given module into the current namespace. Starting with `(namespace-require 'scheme/base)` would introduce a binding for `require` and make a subsequent `(eval '(require my-dsl))` work. The above is better, not only because it is more compact, but also because it avoids introducing bindings that are not part of the domain-specific languages.

16.3 Sharing Data and Code Across Namespaces

Modules not attached to a new namespace will be loaded and instantiated afresh if they are demanded by evaluation. For example, `scheme/base` does not include `scheme/class`, and loading `scheme/class` again will create a distinct class datatype:

```
> (require scheme/class)
> (class? object%)
#t
> (class?
   (parameterize ([current-namespace (make-base-empty-namespace)])
     (namespace-require 'scheme/class) ; loads again
     (eval 'object%)))
#f
```

For cases when dynamically loaded code needs to share more code and data with its context, use the `namespace-attach-module` function. The first argument to `namespace-attach-module` is a source namespace from which to draw a module instance; in some cases, the current namespace is known to include the module that needs to be shared:

```
> (require scheme/class)
> (class?
   (let ([ns (make-base-empty-namespace)])
     (namespace-attach-module (current-namespace)
                              'scheme/class
                              ns)
     (parameterize ([current-namespace ns])
       (namespace-require 'scheme/class) ; uses attached
       (eval 'object%))))
#t
```

Within a module, however, the combination of `define-namespace-anchor` and `namespace-anchor->empty-namespace` offers a more reliable method for obtaining a source namespace:

```
#lang scheme/base

(require scheme/class)

(define-namespace-anchor a)

(define (load-plug-in file)
  (let ([ns (make-base-empty-namespace)])
    (namespace-attach-module (namespace-anchor->empty-namespace a)
                             'scheme/class
                             ns)
    (parameterize ([current-namespace ns])
      (dynamic-require file 'plug-in%))))
```

The anchor bound by `namespace-attach-module` connects the the run time of a module with the namespace in which a module is loaded (which might differ from the current namespace). In the above example, since the enclosing module requires `scheme/class`, the namespace produced by `namespace-anchor->empty-namespace` certainly contains

an instance of `scheme/class`. Moreover, that instance is the same as the one imported into the module, so the class datatype is shared.

17 Macros

17.1 Syntax Certificates

A use of a macro can expand into a use of an identifier that is not exported from the module that binds the macro. In general, such an identifier must not be extracted from the expanded expression and used in a different context, because using the identifier in a different context may break invariants of the macro's module.

For example, the following module exports a macro `go` that expands to a use of `unchecked-go`:

```
(module m mzscheme
  (provide go)
  (define (unchecked-go n x)

    (+ n 17))
  (define-syntax (go stx)
    (syntax-case stx ()
     [(_ x)
      #'(unchecked-go 8 x)])))
```

If the reference to `unchecked-go` is extracted from the expansion of `(go 'a)`, then it might be inserted into a new expression, `(unchecked-go #f 'a)`, leading to disaster. The `datum->syntax` procedure can be used similarly to construct references to an unexported identifier, even when no macro expansion includes a reference to the identifier.

To prevent such abuses of unexported identifiers, the expander rejects references to unexported identifiers unless they appear in *certified* syntax objects. The macro expander always certifies a syntax object that is produced by a transformer. For example, when `(go 'a)` is expanded to `(unchecked-go 8 'a)`, a certificate is attached to the result `(unchecked-go 8 'a)`. Extracting just `unchecked-go` removes the identifier from the certified expression, so that the reference is disallowed when it is inserted into `(unchecked-go #f 'a)`.

In addition to checking module references, the macro expander disallows references to local bindings where the binding identifier is less certified than the reference. Otherwise, the expansion of `(go 'a)` could be wrapped with a local binding that redirects `#%app` to `values`, thus obtaining the value of `unchecked-go`. Note that a capturing `#%app` would have to be extracted from the expansion of `(go 'a)`, since lexical scope would prevent an arbitrary `#%app` from capturing. The act of extracting `#%app` removes its certification, whereas the `#%app` within the expansion is still certified; comparing these certifications, the macro expander rejects the local-binding reference, and `unchecked-go` remains protected.

In much the same way that the macro expander copies properties from a syntax transformer's input to its output (see §11.6 "Syntax Object Properties"), the expander copies certificates from a transformer's input to its output. Building on the previous example,

```
(module n mzscheme
  (require m)
  (provide go-more)
  (define y 'hello)
  (define-syntax (go-more stx)
    #'(go y)))
```

the expansion of `(go-more)` introduces a reference to the unexported `y` in `(go y)`, and a certificate allows the reference to `y`. As `(go y)` is expanded to `(unchecked-go 8 y)`, the certificate that allows `y` is copied over, in addition to the certificate that allows the reference to `unchecked-go`.

When a protected identifier becomes inaccessible by direct reference (i.e., when the current code inspector is changed so that it does not control the module's invocation; see §13.9 "Code Inspectors"), the protected identifier is treated like an unexported identifier.

17.1.1 Certificate Propagation

When the result of a macro expansion contains a `quote-syntax` form, the macro expansion's certificate must be attached to the resulting syntax object to support macro-generating macros. In general, when the macro expander encounters `quote-syntax`, it attaches all certificates from enclosing expressions to the quoted syntax constant. However, the certificates are attached to the syntax constant as *inactive* certificates, and inactive certificates do not count directly for certifying identifier access. Inactive certificates become active when the macro expander certifies the result of a macro expansion; at that time, the expander removes all inactive certificates within the expansion result and attaches active versions of the certificates to the overall expansion result.

For example, suppose that the `go` macro is implemented through a macro:

```
(module m mzscheme
  (provide def-go)
  (define (unchecked-go n x)
    (+ n 17))
  (define-syntax (def-go stx)
   (syntax-case stx ()
     [(_ go)
      #'(define-syntax (go stx)
          (syntax-case stx ()
           [(_ x)
            #'(unchecked-go 8 x)]))])))
```

When `def-go` is used inside another module, the generated macro should legally generate expressions that use `unchecked-go`, since `def-go` in `m` had complete control over the generated macro.

```
(module n mzscheme
  (require m)
  (def-go go)
  (go 10))
```

This example works because the expansion of `(def-go go)` is certified to access protected identifiers in `m`, including `unchecked-go`. Specifically, the certified expansion is a definition of the macro `go`, which includes a syntax-object constant `unchecked-go`. Since the enclosing macro declaration is certified, the `unchecked-go` syntax constant gets an inactive certificate to access protected identifiers of `m`. When `(go 10)` is expanded, the inactive certificate on `unchecked-go` is activated for the macro result `(unchecked-go 8 10)`, and the access of `unchecked-go` is allowed.

To see why `unchecked-go` as a syntax constant must be given an inactive certificate instead of an active one, it's helpful to write the `def-go` macro as follows:

```
(define-syntax (def-go stx)
 (syntax-case stx ()
   [(_ go)
    #'(define-syntax (go stx)
        (syntax-case stx ()
         [(_ x)
          (with-syntax ([ug (quote-syntax unchecked-go)])
            #'(ug 8 x))]))]))
```

In this case, `unchecked-go` is clearly quoted as an immediate syntax object in the expansion of `(def-go go)`. If this syntax object were given an active certificate, then it would keep the certificate—directly on the identifier `unchecked-go`—in the result `(unchecked-go 8 10)`. Consequently, the `unchecked-go` identifier could be extracted and used with its certificate intact. Attaching an inactive certificate to `unchecked-go` and activating it only for the complete result `(unchecked-go 8 10)` ensures that `unchecked-go` is used only in the way intended by the implementor of `def-go`.

The `datum->syntax` procedure allows inactive certificates to be transferred from one syntax object to another. Such transfers are allowed because a macro transformer with access to the syntax object could already wrap it with an arbitrary context before activating the certificates. In practice, transferring inactive certificates is useful mainly to macros that implement to new template forms, such as `syntax/loc`.

17.1.2 Internal Certificates

In some cases, a macro implementor intends to allow limited destructuring of a macro result without losing the result's certificate. For example, given the following `define-like-y` macro,

```
(module q mzscheme
```

```
(provide define-like-y)
(define y 'hello)
(define-syntax (define-like-y stx)
  (syntax-case stx ()
    [(_ id) #'(define-values (id) y)])))
```

someone may use the macro in an internal definition:

```
(let ()
  (define-like-y x)
  x)
```

The implementor of the `q` module most likely intended to allow such uses of `define-like-y`. To convert an internal definition into a `letrec` binding, however, the `define` form produced by `define-like-y` must be deconstructed, which would normally lose the certificate that allows the reference to `y`.

The internal use of `define-like-y` is allowed because the macro expander treats specially a transformer result that is a syntax list beginning with `define-values`. In that case, instead of attaching the certificate to the overall expression, the certificate is instead attached to each individual element of the syntax list, pushing the certificates into the second element of the list so that they are attached to the defined identifiers. Thus, a certificate is attached to `define-values`, `x`, and `y` in the expansion result `(define-values (x) y)`, and the definition can be deconstructed for conversion to `letrec`.

Just like the new certificate that is added to a transformer result, old certificates from the input are similarly moved to syntax-list elements when the result starts with `define-values`. Thus, `define-like-y` could have been implemented to produce `(define id y)`, using `define` instead of `define-values`. In that case, the certificate to allow reference to `y` would be attached initially to the expansion result `(define x y)`, but as the `define` is expanded to `define-values`, the certificate would be moved to the parts.

The macro expander treats syntax-list results starting with `define-syntaxes` in the same way that it treats results starting with `define-values`. Syntax-list results starting with `begin` are treated similarly, except that the second element of the syntax list is treated like all the other elements (i.e., the certificate is attached to the element instead of its content). Furthermore, the macro expander applies this special handling recursively, in case a macro produces a `begin` form that contains nested `define-values` forms.

The default application of certificates can be overridden by attaching a `'certify-mode` property (see §11.6 “Syntax Object Properties”) to the result syntax object of a macro transformer. If the property value is `'opaque`, then the certificate is attached to the syntax object and not its parts. If the property value is `'transparent`, then the certificate is attached to the syntax object’s parts. If the property value is `'transparent-binding`, then the certificate is attached to the syntax object’s parts and to the sub-parts of the second part (as for `define-values` and `define-syntaxes`). The `'transparent` and `'transparent-binding` modes triggers recursive property checking at the parts, so that the certificate can

be pushed arbitrarily deep into a transformer's result.

18 Reader Extension

19 Security

20 Memory Management

20.1 Weak Boxes

20.2 Ephemerons

21 Performance

Alan Perlis famously quipped "Lisp programmers know the value of everything and the cost of nothing." A Scheme programmer knows, for example, that a `lambda` anywhere in a program produces a value that is closed over it lexical environment—but how much does allocating that value cost? While most programmers have a reasonable grasp of the cost of various operations and data structures at the machine level, the gap between the Scheme language model and the underlying computing machinery can be quite large.

In this chapter, we narrow the gap by explaining details of the PLT Scheme compiler and run-time system and how they affect the run-time and memory performance of Scheme code.

21.1 The Bytecode and Just-in-Time (JIT) Compilers

Every definition or expression to be evaluated by Scheme is compiled to an internal bytecode format. In interactive mode, this compilation occurs automatically and on-the-fly. Tools like `setup-plt` and `compile-file` marshal compiled bytecode to a file. Most of the time required to compile a file is actually in macro expansion; generating bytecode from fully expanded code is relatively fast.

The bytecode compiler applies all standard optimizations, such as constant propagation, constant folding, inlining, and dead-code elimination. For example, in an environment where `+` has its usual binding, the expression `(let ([x 1] [y (lambda () 4)]) (+ 1 (y)))` is compiled the same as the constant `5`.

On some platforms, bytecode is further compiled to native code via a *just-in-time* or *JIT* compiler. The JIT compiler substantially speeds programs that execute tight loops, arithmetic on small integers, and arithmetic on inexact real numbers. Currently, JIT compilation is supported for x86, x86_64 (a.k.a. AMD64), and 32-bit PowerPC processors. The JIT compiler can be disabled via the `eval-jit-enabled` parameter or the `--no-jit`/`-j` command-line flag.

The JIT compiler works incrementally as functions are applied, but the JIT compiler makes only limited use of run-time information when compiling procedures, since the code for a given module body or `lambda` abstraction is compiled only once. The JIT's granularity of compilation is a single procedure body, not counting the bodies of any lexically nested procedures. The overhead for JIT compilation is normally so small that it is difficult to detect.

21.2 Modules and Performance

The module system aids optimization by helping to ensure that identifiers have the usual bindings. That is, the `+` provided by `scheme/base` can be recognized by the compiler and inlined, which is especially important for JIT-compiled code. In contrast, in a traditional interactive Scheme system, the top-level `+` binding might be redefined, so the compiler cannot assume a fixed `+` binding (unless special flags or declarations act as a poor-man's module system to indicate otherwise).

Even in the top-level environment, importing with `require` enables some inlining optimizations. Although a `+` definition at the top level might shadow an imported `+`, the shadowing definition applies only to expressions evaluated later.

Within a module, inlining and constant-propagation optimizations take additional advantage of the fact that definitions within a module cannot be mutated when no `set!` is visable at compile time. Such optimizations are unavailable in the top-level environment. Although this optimization within modules is important for performance, it hinders some forms of interactive development and exploration. The `compile-enforce-module-constants` parameter disables the JIT compiler's assumptions about module definitions when interactive exploration is more important. See §6.6 "Assignment and Redefinition" for more information.

Currently, the compiler does not attempt to inline or propagate constants across module boundary, except for exports of the built-in modules (such as the one that originally provides `+`).

The later section §21.5 "`letrec` Performance" provides some additional caveats concerning inlining of module bindings.

21.3 Function-Call Optimizations

When the compiler detects a function call to an immediately visible function, it generates more efficient code than for a generic call, especially for tail calls. For example, given the program

```
(letrec ([odd (lambda (x)
                (if (zero? x)
                    #f
                    (even (sub1 x))))]
         [even (lambda (x)
                 (if (zero? x)
                     #t
                     (odd (sub1 x))))])
  (odd 40000000))
```

the compiler can detect the odd–even loop and produce code that runs much faster via loop unrolling and related optimizations.

Within a module form, `defined` variables are lexically scoped like `letrec` bindings, and definitions within a module therefore permit call optimizations, so

```
(define (odd x) ....)
(define (even x) ....)
```

within a module would perform the same as the `letrec` version.

Primitive operations like `pair?`, `car`, and `cdr` are inlined at the machine-code level by the JIT compiler. See also the later section §21.6 "Fixnum and Flonum Optimizations" for information about inlined arithmetic operations.

21.4 Mutation and Performance

Using `set!` to mutate a variable can lead to bad performance. For example, the microbenchmark

```
#lang scheme/base

(define (subtract-one x)
  (set! x (sub1 x))
  x)

(time
  (let loop ([n 4000000])
    (if (zero? n)
        'done
        (loop (subtract-one n)))))
```

runs much more slowly than the equivalent

```
#lang scheme/base

(define (subtract-one x)
  (sub1 x))

(time
  (let loop ([n 4000000])
    (if (zero? n)
        'done
        (loop (subtract-one n)))))
```

In the first variant, a new location is allocated for `x` on every iteration, leading to poor performance. A more clever compiler could unravel the use of `set!` in the first example, but since mutation is discouraged (see §4.9.1 “Guidelines for Using Assignment”), the compiler’s effort is spent elsewhere.

More significantly, mutation can obscure bindings where inlining and constant-propagation might otherwise apply. For example, in

```
(let ([minus1 #f])
  (set! minus1 sub1)
  (let loop ([n 4000000])
    (if (zero? n)
        'done
        (loop (minus1 n)))))
```

the `set!` obscures the fact that `minus1` is just another name for the built-in `sub1`.

21.5 `letrec` Performance

When `letrec` is used to bind only procedures and literals, then the compiler can treat the bindings in an optimal manner, compiling uses of the bindings efficiently. When other kinds of bindings are mixed with procedures, the compiler may be less able to determine the control flow.

For example,

```
(letrec ([loop (lambda (x)
                 (if (zero? x)
                     'done
                     (loop (next x))))]
         [junk (display loop)]
         [next (lambda (x) (sub1 x))])
  (loop 40000000))
```

likely compiles to less efficient code than

```
(letrec ([loop (lambda (x)
                 (if (zero? x)
                     'done
                     (loop (next x))))]
         [next (lambda (x) (sub1 x))])
  (loop 40000000))
```

In the first case, the compiler likely does not know that `display` does not call `loop`. If it did, then `loop` might refer to `next` before the binding is available.

This caveat about `letrec` also applies to definitions of functions and constants within modules. A definition sequence in a module body is analogous to a sequence of `letrec` bindings, and non-constant expressions in a module body can interfere with the optimization of references to later bindings.

21.6 Fixnum and Flonum Optimizations

A *fixnum* is a small exact integer. In this case, "small" depends on the platform. For a 32-bit machine, numbers that can be expressed in 30 bits plus a sign bit are represented as fixnums. On a 64-bit machine, 62 bits plus a sign bit are available.

A *flonum* is used to represent any inexact real number. They correspond to 64-bit IEEE floating-point numbers on all platforms.

Inlined fixnum and flonum arithmetic operations are among the most important advantages of the JIT compiler. For example, when `+` is applied to two arguments, the generated machine code tests whether the two arguments are fixnums, and if so, it uses the machine's instruction to add the numbers (and check for overflow). If the two numbers are not fixnums, then the next check whether whether both are flonums; in that case, the machine's floating-point operations are used directly. For functions that take any number of arguments, such as `+`, inlining is applied only for the two-argument case (except for `-`, whose one-argument case is also inlined).

Flonums are *boxed*, which means that memory is allocated to hold every result of a flonum computation. Fortunately, the generational garbage collector (described later in §21.7 "Memory Management") makes allocation for short-lived results reasonably cheap. Fixnums, in contrast are never boxed, so they are especially cheap to use.

21.7 Memory Management

PLT Scheme is available in two variants: *3m* and *CGC*. The 3m variant uses a modern, *generational garbage collector* that makes allocation relatively cheap for short-lived objects. The CGC variant uses a *conservative garbage collector* which facilitates interaction with C code at the expense of both precision and speed for Scheme memory management. The 3m variant is the standard one.

Although memory allocation is reasonably cheap, avoiding allocation altogether is normally faster. One particular place where allocation can be avoided sometimes is in *closures*, which are the run-time representation of functions that contain free variables. For example,

```
(let loop ([n 40000000][prev-thunk (lambda () #f)])
  (if (zero? n)
      (prev-thunk)
```

```
      (loop (sub1 n)
            (lambda () n))))
```

allocates a closure on every iteration, since `(lambda () n)` effectively saves `n`.

The compiler can eliminate many closures automatically. For example, in

```
(let loop ([n 40000000][prev-val #f])
  (let ([prev-thunk (lambda () n)])
    (if (zero? n)
        prev-val
        (loop (sub1 n) (prev-thunk)))))
```

no closure is ever allocated for `prev-thunk`, because its only application is visible, and so it is inlined. Similarly, in

```
(let n-loop ([n 400000])
  (if (zero? n)
      'done
      (let m-loop ([m 100])
        (if (zero? m)
            (n-loop (sub1 n))
            (m-loop (sub1 m))))))
```

then the expansion of the `let` form to implement `m-loop` involves a closure over `n`, but the compiler automatically converts the closure to pass itself `n` as an argument instead.

22 Running and Creating Executables

22.1 Running MzScheme and MrEd

Depending on command-line arguments, MzScheme runs in interactive mode, module mode, or load mode.

22.1.1 Interactive Mode

When `mzscheme` is run with no command-line arguments (other than confguration options, like `-j`), then it starts a REPL with a `>` prompt:

```
Welcome to MzScheme
>
```

For information on GNU Readline support, see `readline`.

To initialize the REPL's environment, `mzscheme` first requires the `scheme/init` module, which provides all of `scheme`, and also installs `pretty-print` for display results. Finally, `mzscheme` loads the file reported by `(find-system-path 'init-file)`, if it exists, before starting the REPL.

If any command-line arguments are provided (other than configuration options), add `-i` or `--repl` to re-enable the REPL. For example,

```
mzscheme -e '(display "hi\n")'
```

displays "hi" on start-up, but still presents a REPL.

If module-requiring flags appear before `-i`/`--repl`, they cancel the automatic requiring of `scheme/init`. This behavior can be used to initialize the REPL's environment with a different language. For example,

```
mzscheme -l scheme/base -i
```

starts a REPL using a much smaller initial language (that loads much faster). Beware that most modules do not provide the basic syntax of Scheme, including function-call syntax and `require`. For example,

```
mzscheme -l scheme/date -i
```

produces a REPL that fails for every expression, because `scheme/date` provides only a few functions, and not the `#%top-interaction` and `#%app` bindings that are needed to evaluate top-level function calls in the REPL.

If a module-requiring flag appears after `-i`/`--repl` instead of before it, then the module is required after `scheme/init` to augment the initial environment. For example,

```
mzscheme -i -l scheme/date
```

starts a useful REPL with `scheme/date` available in addition to the exports of `scheme`.

22.1.2 Module Mode

If a file argument is supplied to `mzscheme` before any command-line switch (other than configuration options), then the file is required as a module, and (unless `-i`/`--repl` is specified), no REPL is started. For example,

```
mzscheme hello.ss
```

requires the `"hello.ss"` module and then exits. Any argument after the file name, flag or otherwise, is preserved as a command-line argument for use by the required module via `current-command-line-arguments`.

If command-line flags are used, then the `-u` or `--require-script` flag can be used to explicitly require a file as a module. The `-t` or `--require` flag is similar, except that additional command-line flags are processed by `mzscheme`, instead of preserved for the required module. For example,

```
mzscheme -t hello.ss -t goodbye.ss
```

requires the `"hello.ss"` module, then requires the `"goodbye.ss"` module, and then exits.

The `-l` or `--lib` flag is similar to `-t`/`--require`, but it requires a module using a `lib` module path instead of a file path. For example,

```
mzscheme -l compiler
```

is the same as running the `mzc` executable with no arguments, since the `compiler` module is the main `mzc` module.

22.1.3 Load Mode

The `-f` or `--load` flag supports `load`ing top-level expressions in a file directly, as opposed to expressions within a module file. This evaluation is like starting a REPL and typing the expressions directly, except that the results are not printed. For example,

```
mzscheme -f hi.ss
```

`load`s `"hi.ss"` and exits. Note that load mode is generally a bad idea, for the reasons explained in §1.3 “A Note to Readers with Scheme/Lisp Experience”; using module mode is typically better.

The `-e` or `--eval` flag accepts an expression to evaluate directly. Unlike file loading, the result of the expression is printed, as in a REPL. For example,

```
mzscheme -e '(current-seconds)'
```

prints the number of seconds since January 1, 1970.

For file loading and expression evaluation, the top-level environment is created in the same way for interactive mode: `scheme/init` is required unless another module is specified first. For example,

```
mzscheme -l scheme/base -e '(current-seconds)'
```

likely runs faster, because it initializes the environment for evaluation using the smaller `scheme/base` language, instead of `scheme/init`.

22.2 Creating Stand-Alone Executables

22.3 Unix Scripts

Under Unix and Mac OS X, a Scheme file can be turned into an executable script using the shell's `#!` convention. The first two characters of the file must be `#!`; the next character must be either a space or `/`, and the remainder of the first line must be a command to execute the script. For some platforms, the total length of the first line is restricted to 32 characters, and sometimes the space is required.

The simplest script format uses an absolute path to a MzScheme executable followed by a module declaration. For example, if `mzscheme` is installed in `"/usr/local/bin"`, then a file containing the following text acts as a "hello world" script:

```
#! /usr/local/bin/mzscheme
#lang scheme/base
"Hello, world!"
```

In particular, if the above is put into a file `"hello"` and the file is made executable (e.g., with `chmod a+x hello`), then typing `./hello` at the shell prompt produces the output `"Hello, world!"`.

The above script works because the operating system automatically puts the path to the script as the argument to the program started by the `#!` line, and because `mzscheme` treats a single non-flag argument as a file containing a module to run.

Instead of specifying a complete path to the `mzscheme` executable, a popular alternative is to require that `mzscheme` is in the user's command path, and then "trampoline" using `/usr/bin/env`:

```
#! /usr/bin/env mzscheme
#lang scheme/base
"Hello, world!"
```

In either case, command-line arguments to a script are available via `current-command-line-arguments`:

```
#! /usr/bin/env mzscheme
#lang scheme/base
(printf "Given arguments: ~s\n"
        (current-command-line-arguments))
```

If the name of the script is needed, it is available via `(find-system-path 'run-file)`, instead of via `(current-command-line-arguments)`.

Usually, then best way to handle command-line arguments is to parse them using the `command-line` form provided by `scheme`. The `command-line` form extracts command-line arguments from `(current-command-line-arguments)` by default:

```
#! /usr/bin/env mzscheme
#lang scheme

(define verbose? (make-parameter #f))

(define greeting
  (command-line
   #:once-each
   [("-v") "Verbose mode" (verbose? #t)]
   #:args
   (str) str))

(printf "~a~a\n"
        greeting
        (if (verbose?) " to you, too!" ""))
```

Try running the above script with the `--help` flag to see what command-line arguments are allowed by the script.

An even more general trampoline uses `/bin/sh` plus some lines that are comments in one language and expressions in the other. This trampoline is more complicated, but it provides more control over command-line arguments to `"mzscheme"`:

```
#! /bin/sh
#|
exec mzscheme -cu "$0" ${1+"$@"}
|#
#lang scheme/base
```

```
(printf "This script started slowly, because the use of\n")
(printf "bytecode files has been disabled via -c.\n")
(printf "Given arguments: ~s\n"
        (current-command-line-arguments))
```

Note that `#!` starts a line comment in Scheme, and `#|`...`|#` forms a block comment. Meanwhile, `#` also starts a shell-script comment, while `exec mzscheme` aborts the shell script to start `mzscheme`. That way, the script file turns out to be valid input to both `/bin/sh` and `mzscheme`.

23 Configuration and Compilation

- **setup-plt**, a command-line tool for installation tasks
- **planet**, a command-line tool for managing packages that are normally downloaded automatically, on demand
- **mzc**, a command-line tool for miscellaneous tasks, such as compiling Scheme source, compiling C-implemented extensions to the run-time system, generating executables, and building distribution packages

24 More Libraries

§“**GUI**: PLT Graphics Toolkit” describes the PLT Scheme graphics toolbox, whose core is implemented by the MrEd executable.

§“**FFI**: PLT Scheme Foreign Interface” describes tools for using Scheme to access libraries that are normally used by C programs.

§“**Web Server**: PLT HTTP Server” describes the PLT Scheme web server, which supports servlets implemented in Scheme.

PLT Scheme Documentation lists documentation for many other installed libraries. Run `plt-help` to find documentation for libraries that are installed on your system and specific to your user account.

PLaneT offers even more downloadable packages contributed by PLT Scheme users.

Bibliography

[Goldberg04] David Goldberg, Robert Bruce Findler, and Matthew Flatt, “Super and Inner—Together at Last!,” Object-Or

[Flatt06] Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen, “Scheme with Classes, Mixins, and Traits (inv

Index