Teach Yourself Scheme in Fixnum Days

© Dorai Sitaram, 1998–2000
All Rights Reserved
ds26@gte.com
Document will grow

Contents

Preface

This is an introduction to the Scheme programming language. It is intended as a quick-start guide, something a novice can use to get a non-trivial working knowledge of the language, before moving on to more comprehensive and in-depth texts.

The text describes *an* approach to writing a crisp and utilitarian Scheme. Although we will not cover Scheme from `abs` to `zero?`, we will not shy away from those aspects of the language that are difficult, messy, nonstandard, or unusual, but nevertheless useful and usable. Such aspects include `call-with-current-continuation`, system interface, and dialect diversity. Our discussions will be informed by our focus on problem-solving, not by a quest for metalinguistic insight. I have therefore left out many of the staples of traditional Scheme tutorials. There will be no in-depth pedagogy; no dwelling on the semantic appeal of Scheme; no metacircular interpreters; no discussion of the underlying implementation; and no evangelizing about Scheme's virtues. This is not to suggest that these things are unimportant. However, they are arguably not immediately relevant to someone seeking a quick introduction.

How quick though? I do not know if one can teach oneself Scheme in 21 days[1], although I have heard it said that the rudiments of Scheme should be a matter of an afternoon's study. The Scheme standard [KCR98] itself, for all its exacting comprehensiveness, is a mere fifty pages long. It may well be that the insight, when it comes, will arrive in its entirety in one afternoon, though there is no telling how many afternoons of mistries must precede it. Until that zen moment, here is my gentle introduction.

—d

[1] A *fixnum* is a machine's idea of a "small" integer. Every machine has its own idea of how big a fixnum can be.

Chapter 1

Enter Scheme

The canonical first program is the one that says `"Hello, World!"` on the console. Using your favourite editor, create a file called `hello.scm` with the following contents:

```
;The first program

(begin
  (display "Hello, World!")
  (newline))
```

The first line is a comment. When Scheme sees a semicolon, it ignores it and all the following text on the line.

The `begin`-*form* is Scheme's way of introducing a sequence of *subforms*. In this case there are two subforms. The first is a call to the `display` procedure that outputs its argument (the string `"Hello, World!"`) to the console (or "standard output"). It is followed by a `newline` procedure call, which outputs a carriage return.

To run this program, first start your Scheme. This is usually done by typing the name of your Scheme executable at the operating-system command line. Eg, in the case of MzScheme [Fla], you type

```
mzscheme
```

at the operating-system prompt.

This invokes the Scheme *listener*, which *read*s your input, *eval*uates it, *print*s the result (if any), and then waits for more input from you. For this reason, it is often called the *read-eval-print loop*. Note that this is not much different from your operating-system command line, which also reads your commands, executes them, and then waits for more. Like the operating system, the Scheme listener has its own prompt — usually this is `>`, but could be something else.

At the listener prompt, *load* the file `hello.scm`. This is done by typing

```
(load "hello.scm")
```

Scheme will now execute the contents of `hello.scm`, outputting `Hello, World!` followed by a carriage return. After this, you will get the listener prompt again, waiting for more input from you.

Since you have such an eager listener, you need not always write your programs in a file and load them. Sometimes, it is easier, especially when you are in an exploring mood, to simply type expressions directly at the listener prompt and see what happens. For example, typing the form

```
(begin (display "Hello, World!")
       (newline))
```

at the Scheme prompt produces

```
Hello, World!
```

Actually, you could simply have typed the form `"Hello, World!"` at the listener, and you would have obtained as result the string

```
"Hello, World!"
```

because that is the result of the listener evaluating `"Hello, World!"`.

Other than the fact that the second approach produces a result with double-quotes around it, there is one other significant difference between the last two programs. The first (ie, the one with the `begin`) does not evaluate to anything — the `Hello, World!` it emits is a *side-effect* produced by the `display` and `newline` procedures writing to the standard output. In the second program, the form `"Hello, World!"` *evaluates* to the result, which in this case is the same string as the form.

Henceforth, we will use the notation ⇒ to denote evaluation. Thus

```
E ⇒ v
```

indicates that the form `E` evaluates to a result value of `v`. Eg,

```
(begin
  (display "Hello, World!")
  (newline))
⇒
```

(ie, nothing or void), although it has the side-effect of writing

```
Hello, World!
```

to the standard output. On the other hand,

```
"Hello, World!"
⇒ "Hello, World!"
```

In either case, we are still at the listener. To exit, type

```
(exit)
```

and this will land you back at the operating-system command-line (which, as we've seen, is also a kind of listener).

The listener is convenient for interactive testing of programs and program fragments. However it is by no means necessary. You may certainly stick to the tradition of creating programs in their entirety in files, and having Scheme execute them without any explicit "listening". In MzScheme, for instance, you could say (at the operating-system prompt)

```
mzscheme -r hello.scm
```

and this will produce the greeting without making you deal with the listener. After the greeting, `mzscheme` will return you to the operating-system prompt. This is almost as if you said

```
echo Hello, World!
```

You could even make `hello.scm` seem like an operating-system command (a shell script or a batch file), but that will have to wait till chapter 16.

Chapter 2

Data types

A *data type* is a collection of related values. These collections need not be disjoint, and they are often hierarchical. Scheme has a rich set of data types: some are simple (indivisible) data types and others are compound data types made by combining other data types.

2.1 Simple data types

The simple data types of Scheme include booleans, numbers, characters, and symbols.

2.1.1 Booleans

Scheme's booleans are `#t` for true and `#f` for false. Scheme has a predicate procedure called `boolean?` that checks if its argument is boolean.

```
(boolean? #t)              ⇒ #t
(boolean? "Hello, World!") ⇒ #f
```

The procedure `not` negates its argument, considered as a boolean.

```
(not #f)              ⇒ #t
(not #t)              ⇒ #f
(not "Hello, World!") ⇒ #f
```

The last expression illustrates a Scheme convenience: In a context that requires a boolean, Scheme will treat any value that is not `#f` as a true value.

2.1.2 Numbers

Scheme numbers can be integers (eg, `42`), rationals (`22/7`), reals (`3.1416`), or complex (`2+3i`). An integer is a rational is a real is a complex number is a number. Predicates exist for testing the various kinds of numberness:

```
(number? 42)        ⇒ #t
(number? #t)        ⇒ #f
(complex? 2+3i)     ⇒ #t
(real? 2+3i)        ⇒ #f
(real? 3.1416)      ⇒ #t
(real? 42)          ⇒ #t
(rational? 3.1416)  ⇒ #f
(rational? 22/7)    ⇒ #t
(integer? 22/7)     ⇒ #f
(integer? 42)       ⇒ #t
```

Scheme integers need not be specified in decimal (base 10) format. They can be specified in binary by prefixing the numeral with `#b`. Thus `#b1100` is the number twelve. The octal prefix is `#o` and the hex prefix is `#x`. (The optional decimal prefix is `#d`.)

Numbers can tested for equality using the general-purpose equality predicate `eqv?`.

```
(eqv? 42 42)   ⇒ #t
(eqv? 42 #f)   ⇒ #f
(eqv? 42 42.0) ⇒ #f
```

However, if you know that the arguments to be compared are numbers, the special number-equality predicate `=` is more apt.

```
(= 42 42)   ⇒ #t
(= 42 #f)   →ERROR!!!
(= 42 42.0) ⇒ #t
```

Other number comparisons are also possible:

```
(< 3 2)    ⇒ #f
(>= 4.5 3) ⇒ #t
```

Arithmetic procedures `+`, `-`, `*`, `/` have the expected behavior:

```
(+ 1 2 3) ⇒ 6
(- 5.3 2) ⇒ 3.3
(- 5 2 1) ⇒ 2
(* 1 2 3) ⇒ 6
(/ 6 3)   ⇒ 2
(/ 22 7)  ⇒ 22/7
```

For a single argument, `-` and `/` return the negation and the reciprocal respectively:

```
(- 4) ⇒ -4
(/ 4) ⇒ 1/4
```

The procedures `max` and `min` return the maximum and minimum respectively of the number arguments supplied to them. Any number of arguments can be so supplied.

```
(max 1 3 4 2 3) ⇒ 4
(min 1 3 4 2 3) ⇒ 1
```

This is just the tip of the iceberg. Scheme provides a large and comprehensive suite of arithmetic and trigonometric procedures. Consult R5RS [KCR98].

2.1.3 Characters

Scheme character data are represented by prefixing the character with `#\`. Thus, `#\c` is the character `c`. Some non-graphic characters have more descriptive names, eg, `#\newline`, `#\tab`. The character for space can be written `#\ `, or more readably, `#\space`.

The character predicate is `char?`:

```
(char? #\c) ⇒ #t
(char? 1)   ⇒ #f
(char? #\;) ⇒ #t
```

Note that a semicolon character datum does not trigger a comment.

The character data type has its set of comparison predicates:

```
(char=? #\a #\a)  ⇒ #t
(char<? #\a #\b)  ⇒ #t
(char>=? #\a #\b) ⇒ #f
```

To make the comparisons case-insensitive, use `char-ci` instead of `char` in the procedure name:

```
(char-ci=? #\a #\A) ⇒ #t
(char-ci<? #\a #\B) ⇒ #t
```

The case conversion procedures are `char-downcase` and `char-upcase`:

```
(char-downcase #\A) ⇒ #\a
(char-upcase #\a)   ⇒ #\A
```

2.1.4 Symbols

The simple data types we saw above are *self-evaluating*. Ie, if you typed any object from these data types to the listener, the evaluated result returned by the listener will be the same as what you typed in.

```
#t  ⇒ #t
42  ⇒ 42
#\c ⇒ #\c
```

Symbols don't behave the same way. This is because symbols are used by Scheme programs as *identifiers* for *variables*, and thus will evaluate to the value that the variable holds. Nevertheless, symbols are a simple data type, and symbols are legitimate values that Scheme can traffic in, along with characters, numbers, and the rest.

To specify a symbol without making Scheme think it is a variable, you should *quote* the symbol:

```
(quote xyz)
⇒ xyz
```

Since this type of quoting is very common in Scheme, a convenient abbreviation is provided. The expression

```
'E
```

will be treated by Scheme as equivalent to

```
(quote E)
```

Scheme symbols are named by a sequence of characters. About the only limitation on a symbol's name are that it shouldn't be mistakable for some other data, eg, characters or booleans or numbers or compound data. Thus, `this-is-a-symbol`, `i18n`, `<=>`, and `$!#*` are all symbols; `16`, `-i` (a complex number!), `#t`, `"this-is-a-string"`, and `(barf)` (a list) are not. The predicate for checking symbolness is called `symbol?`:

```
(symbol? 'xyz) ⇒ #t
(symbol? 42)   ⇒ #f
```

Scheme symbols are normally case-insensitive. Thus the symbols `Calorie` and `calorie` are identical:

```
(eqv? 'Calorie 'calorie)
⇒ #t
```

We can use the symbol `xyz` as a global variable by using the form `define`:

```
(define xyz 9)
```

This says the variable `xyz` holds the value `9`. If we feed `xyz` to the listener, the result will be the value held by `xyz`:

```
xyz
⇒ 9
```

We can use the form `set!` to *change* the value held by a variable:

```
(set! xyz #\c)
```

Now

```
xyz
⇒ #\c
```

2.2 Compound data types

Compound data types are built by combining values from other data types in structured ways.

2.2.1 Strings

Strings are sequences of characters (not to be confused with symbols, which are simple data that have a sequence of characters as their name). You can specify strings by enclosing the constituent characters in double-quotes. Strings evaluate to themselves.

```
"Hello, World!"
⇒ "Hello, World!"
```

The procedure `string` takes a bunch of characters and returns the string made from them:

```
(string #\h #\e #\l #\l #\o)
⇒ "hello"
```

Let us now define a global variable `greeting`.

```
(define greeting "Hello; Hello!")
```

Note that a semicolon inside a string datum does not trigger a comment.

The characters in a given string can be individually accessed and modified. The procedure `string-ref` takes a string and a (0-based) index, and returns the character at that index:

```
(string-ref greeting 0)
⇒ #\H
```

The procedure `string-set!` replaces the character at an index:

```
(string-set! greeting 1 #\a)

greeting
⇒ "Hallo; Hello!"
```

New strings can be created by appending other strings:

```
(string-append "E "
               "Pluribus "
               "Unum")
⇒ "E Pluribus Unum"
```

You can make a string of a specified length, and fill it with `string-set!` later.

```
(define a-3-char-long-string (make-string 3))
```

The predicate for checking stringness is `string?`.

2.2.2 Vectors

Vectors are sequences like strings, but their elements can be anything, not just characters. Indeed, the elements can be vectors themselves, which is a good way to generate multidimensional vectors.

Here's a way to create a vector of the first five integers:

```
(vector 0 1 2 3 4)
⇒ #(0 1 2 3 4)
```

Note Scheme's representation of a vector value: a `#` character followed by the vector's contents enclosed in parentheses.

In analogy with `make-string`, the procedure `make-vector` makes a vector of a specific length:

```
(define v (make-vector 5))
```

The procedures `vector-ref` and `vector-set!` access and modify vector elements. The predicate for checking if something is a vector is `vector?`.

2.2.3 Dotted pairs and lists

A *dotted pair* is a compound value made by combining any two arbitrary values into an ordered couple. The first element is called the *car*, the second element is called the *cdr*, and the combining procedure is `cons`.

```
(cons 1 #t)
⇒ (1 . #t)
```

Dotted pairs are not self-evaluating, and so to specify them directly as data (ie, without producing them via a `cons`-call), one must explicitly quote them:

```
'(1 . #t) ⇒ (1 . #t)

(1 . #t)  →ERROR!!!
```

The accessor procedures are `car` and `cdr`:

```
(define x (cons 1 #t))

(car x)
⇒ 1

(cdr x)
⇒ #t
```

The elements of a dotted pair can be replaced by the mutator procedures `set-car!` and `set-cdr!`:

```
(set-car! x 2)

(set-cdr! x #f)

x
⇒ (2 . #f)
```

Dotted pairs can contain other dotted pairs.

```
(define y (cons (cons 1 2) 3))

y
⇒ ((1 . 2) . 3)
```

The `car` of the `car` of this list is `1`. The `cdr` of the `car` of this list is `2`. Ie,

```
(car (car y))
⇒ 1

(cdr (car y))
⇒ 2
```

Scheme provides procedure abbreviations for cascaded compositions of the `car` and `cdr` procedures. Thus, `caar` stands for "`car` of `car` of", and `cdar` stands for "`cdr` of `car` of", etc.

```
(caar y)
⇒ 1

(cdar y)
⇒ 2
```

`c...r`-style abbreviations for upto four cascades are guaranteed to exist. Thus, `cadr`, `cdadr`, and `cdaddr` are all valid. `cdadadr` might be pushing it.

When nested dotting occurs along the second element, Scheme uses a special notation to represent the resulting expression:

```
(cons 1 (cons 2 (cons 3 (cons 4 5))))
⇒ (1 2 3 4 . 5)
```

Ie, `(1 2 3 4 . 5)` is an abbreviation for `(1 . (2 . (3 . (4 . 5))))`. The last cdr of this expression is `5`.

Scheme provides a further abbreviation if the last cdr is a special object called the *empty list*, which is represented by the expression `()`. The empty list is not considered self-evaluating, and so one should quote it when supplying it as a value in a program:

```
'() ⇒ ()
```

The abbreviation for a dotted pair of the form `(1 . (2 . (3 . (4 . ()))))` is

```
(1 2 3 4)
```

This special kind of nested dotted pair is called a *list*. This particular list is four elements long. It could have been created by saying

```
(cons 1 (cons 2 (cons 3 (cons 4 '()))))
```

but Scheme provides a procedure called `list` that makes list creation more convenient. `list` takes any number of arguments and returns the list containing them:

```
(list 1 2 3 4)
⇒ (1 2 3 4)
```

Indeed, if we know all the elements of a list, we can use `quote` to specify the list:

```
'(1 2 3 4)
⇒ (1 2 3 4)
```

List elements can be accessed by index.

```
(define y (list 1 2 3 4))

(list-ref y 0) ⇒ 1
(list-ref y 3) ⇒ 4

(list-tail y 1) ⇒ (2 3 4)
(list-tail y 3) ⇒ (4)
```

`list-tail` returns the *tail* of the list starting from the given index.

The predicates `pair?`, `list?`, and `null?` check if their argument is a dotted pair, list, or the empty list, respectively:

```
(pair? '(1 . 2)) ⇒ #t
(pair? '(1 2))   ⇒ #t
(pair? '())      ⇒ #f
(list? '())      ⇒ #t
(null? '())      ⇒ #t
(list? '(1 2))   ⇒ #t
(list? '(1 . 2)) ⇒ #f
(null? '(1 2))   ⇒ #f
(null? '(1 . 2)) ⇒ #f
```

2.2.4 Conversions between data types

Scheme offers many procedures for converting among the data types. We already know how to convert between the character cases using `char-downcase` and `char-upcase`. Characters can be converted into integers using `char->integer`, and integers can be converted into characters using `integer->char`. (The integer corresponding to a character is usually its ascii code.)

```
(char->integer #\d) ⇒ 100
(integer->char 50)  ⇒ #\2
```

Strings can be converted into the corresponding list of characters.

```
(string->list "hello") ⇒ (#\h #\e #\l #\l #\o)
```

Other conversion procedures in the same vein are `list->string`, `vector->list`, and `list->vector`.

Numbers can be converted to strings:

```
(number->string 16) ⇒ "16"
```

Strings can be converted to numbers. If the string corresponds to no number, `#f` is returned.

```
(string->number "16")
⇒ 16

(string->number "Am I a hot number?")
⇒ #f
```

`string->number` takes an optional second argument, the radix.

```
(string->number "16" 8) ⇒ 14
```

because `16` in base 8 is the number fourteen.

Symbols can be converted to strings, and vice versa:

```
(symbol->string 'symbol)
⇒ "symbol"

(string->symbol "string")
⇒ string
```

2.3 Other data types

Scheme contains some other data types. One is the *procedure*. We have already seen many procedures, eg, `display`, `+`, `cons`. In reality, these are variables holding the procedure values, which are themselves not visible as are numbers or characters:

```
cons
⇒ <procedure>
```

The procedures we have seen thus far are *primitive* procedures, with standard global variables holding them. Users can create additional procedure values.

Yet another data type is the *port*. A port is the conduit through which input and output is performed. Ports are usually associated with files and consoles.

In our "Hello, World!" program, we used the procedure `display` to write a string to the console. `display` can take two arguments, one the value to be displayed, and the other the output port it should be displayed on.

In our program, `display`'s second argument was implicit. The default output port used is the standard output port. We can get the current standard output port via the procedure-call `(current-output-port)`. We could have been more explicit and written

```
(display "Hello, World!" (current-output-port))
```

2.4 S-expressions

All the data types discussed here can be lumped together into a single all-encompassing data type called the *s-expression* (*s* for *symbolic*). Thus `42`, `#\c`, `(1 . 2)`, `#(a b c)`, `"Hello"`, `(quote xyz)`, `(string->number "16")`, and `(begin (display "Hello, World!") (newline))` are all s-expressions.

Chapter 3

Forms

The reader will have noted that the Scheme example programs provided thus far are also s-expressions. This is true of all Scheme programs: Programs are data.

Thus, the character datum `#\c` is a program, or a *form*. We will use the more general term *form* instead of *program*, so that we can deal with program fragments too.

Scheme evaluates the form `#\c` to the value `#\c`, because `#\c` is self-evaluating. Not all s-expressions are self-evaluating. For instance the symbol s-expression `xyz` evaluates to the value held by the *variable* `xyz`. The list s-expression `(string->number "16")` evaluates to the number 16.

Not all s-expressions are valid programs. If you typed the dotted-pair s-expression `(1 . 2)` at the Scheme listener, you will get an error.

Scheme evaluates a list form by examining the first element, or *head*, of the form. If the head evaluates to a procedure, the rest of the form is evaluated to get the procedure's arguments, and the procedure is *applied* to the arguments.

If the head of the form is a *special form*, the evaluation proceeds in a manner idiosyncratic to that form. Some special forms we have already seen are `begin`, `define`, and `set!`. `begin` causes its subforms to be evaluated in order, the result of the entire form being the result of the last subform. `define` introduces and initializes a variable. `set!` changes the binding of a variable.

3.1 Procedures

We have seen quite a few primitive Scheme procedures, eg, `cons`, `string->list`, and the like. Users can create their own procedures using the special form `lambda`. For example, the following defines a procedure that adds `2` to its argument:

```
(lambda (x) (+ x 2))
```

The first subform, `(x)`, is the list of parameters. The remaining subform(s) constitute the procedure's body. This procedure can be called on an argument, just like a primitive procedure:

```
((lambda (x) (+ x 2)) 5)
⇒ 7
```

If we wanted to call this same procedure many times, we could create a replica using `lambda` each time, but we can do better. We can use a variable to hold the procedure value:

```
(define add2
  (lambda (x) (+ x 2)))
```

We can then use the variable `add2` each time we need a procedure for adding 2 to its argument:

```
(add2 4) ⇒ 6
(add2 9) ⇒ 11
```

3.1.1 Procedure parameters

The parameters of a `lambda`-procedure are specified by its first subform (the form immediately following the head, the symbol `lambda`). `add2` is a single-argument — or *unary* — procedure, and so its parameter list is the singleton list `(x)`. The symbol `x` acts as a variable holding the procedure's argument. Each occurrence of `x` in the procedure's body refers to the procedure's argument. The variable `x` is said to be *local* to the procedure's body.

We can use 2-element lists for 2-argument procedures, and in general, n-element lists for n-argument procedures. The following is a 2-argument procedure that calculates the area of a rectangle. Its two arguments are the length and breadth of the rectangle.

```
(define area
  (lambda (length breadth)
    (* length breadth)))
```

Notice that `area` multiplies its arguments, and so does the primitive procedure `*`. We could have simply said:

```
(define area *)
```

3.1.2 Variable number of arguments

Some procedures can be called at different times with different numbers of arguments. To do this, the `lambda` parameter list is replaced by a single symbol. This symbol acts as a variable that is bound to the list of the arguments that the procedure is called on.

In general, the `lambda` parameter list can be a list of the form `(x ...)`, a symbol, or a dotted pair of the form `(x ... . z)`. In the dotted-pair case, all the variables before the dot are bound to the corresponding arguments in the procedure call, with the single variable after the dot picking up all all the remaining arguments as one list.

3.2 apply

The Scheme procedure `apply` lets us call a procedure on a *list* of its arguments.

```
(define x '(1 2 3))

(apply + x)
⇒ 6
```

In general, `apply` takes a procedure, followed by a variable number of other arguments, the last of which must be a list. It constructs the argument list by prefixing the last argument with all the other (intervening) arguments. It then returns the result of calling the procedure on this argument list. Eg,

```
(apply + 1 2 3 x)
⇒ 12
```

3.3 Sequencing

We used the `begin` special form to bunch together a group of subforms that need to be evaluated in sequence. Many Scheme forms have *implicit* `begin`s. For example, let's define a 3-argument procedure that displays its three arguments, with spaces between them. A possible definition is:

```
(define display3
  (lambda (arg1 arg2 arg3)
    (begin
      (display arg1)
      (display " ")
      (display arg2)
      (display " ")
      (display arg3)
      (newline))))
```

In Scheme, `lambda`-bodies are implicit `begin`s. Thus, the `begin` in `display3`'s body isn't needed, although it doesn't hurt. `display3`, more simply, is:

```
(define display3
  (lambda (arg1 arg2 arg3)
    (display arg1)
    (display " ")
    (display arg2)
    (display " ")
    (display arg3)
    (newline)))
```

Chapter 4

Conditionals

Like all languages, Scheme provides *conditionals*. The basic form is the `if`:

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

If `test-expression` evaluates to true (ie, any value other than `#f`), the `then-expression` is evaluated. If not, the `else-expression` is evaluated. `else-expression` is optional.

```
(if (= (+ 2 3) 5)
    'schemes-addition-works!)
=> schemes-addition-works!

(if (< 3 2)
    'three-is-less-than-two
    'three-isnt-less-than-two)
=> three-isnt-less-than-two
```

Scheme provides some other conditional forms for convenience. They can all be defined as macros (chap 8) that expand into `if`-expressions.

4.1 'when' and 'unless'

`when` and `unless` are convenient conditionals to use when only one branch (the "then" or the "else" branch) of the basic conditional is needed.

```
(when (< (pressure tube) 60)
   (open-valve tube)
   (attach floor-pump tube)
   (depress floor-pump 5)
   (detach floor-pump tube)
   (close-valve tube))
```

Assuming `pressure` of `tube` is less than `60`, this conditional will attach `floor-pump` to `tube` and `depress` it `5` times. (`attach` and `depress` are some suitable procedures.)

The same program using `if` would be:

```
(if (< (pressure tube) 60)
    (begin
      (open-valve tube)
      (attach floor-pump tube)
      (depress floor-pump 5)
      (detach floor-pump tube)
      (close-valve tube)))
```

Note that `when`'s branch is an implicit `begin`, whereas `if` requires an explicit `begin` if either of its branches has more than one form.

The same behavior can be written using `unless` as follows:

```
(unless (>= (pressure tube) 60)
   (open-valve tube)
   (attach floor-pump tube)
   (depress floor-pump 5)
   (detach floor-pump tube)
   (close-valve tube))
```

Not all Schemes provide `when` and `unless`. If your Scheme does not have them, you can define them as macros (see chap 8).

4.2 cond

The `cond` form is convenient for expressing nested `if`-expressions, where each "else" branch but the last introduces a new `if`. Thus, the form

```
(if (char<? c #\c) -1
    (if (char=? c #\c) 0
        1))
```

can be rewritten using `cond` as:

```
(cond ((char<? c #\c) -1)
      ((char=? c #\c) 0)
      (else 1))
```

The `cond` is thus a *multi-branch* conditional. Each clause has a test and an associated action. The first test that succeeds triggers its associated action. The final `else` clause is chosen if no other test succeeded.

The `cond` actions are implicit `begin`s.

4.3 case

A special case of the `cond` can be compressed into a `case` expression. This is when every test is a membership test.

```
(case c
  ((#\a) 1)
  ((#\b) 2)
  ((#\c) 3)
  (else 4))
⇒ 3
```

The clause whose head contains the value of `c` is chosen.

4.4 'and' and 'or'

Scheme provides special forms for boolean conjunction ("and") and disjunction ("or"). (We have already seen (sec 2.1.1) Scheme's boolean negation `not`, which is a procedure.)

The special form `and` returns a true value if all its subforms are true. The actual value returned is the value of the final subform. If any of the subforms are false, `and` returns `#f`.

```
(and 1 2)  ⇒ 2
(and #f 1) ⇒ #f
```

The special form `or` returns the value of its first true subform. If all the subforms are false, `or` returns `#f`.

```
(or 1 2)  ⇒ 1
(or #f 1) ⇒ 1
```

Both `and` and `or` evaluate their subforms left-to-right. As soon as the result can be determined, `and` and `or` will ignore the remaining subforms.

```
(and 1 #f expression-guaranteed-to-cause-error)
⇒ #f

(or 1 #f expression-guaranteed-to-cause-error)
⇒ 1
```

Chapter 5

Lexical variables

Scheme's variables have lexical scope, ie, they are visible only to forms within a certain contiguous stretch of program text. The *global* variables we have seen thus far are no exception: Their scope is all program text, which is certainly contiguous.

We have also seen some examples of *local* variables. These were the `lambda` parameters, which get *bound* each time the procedure is called, and whose scope is that procedure's body. Eg,

```
(define x 9)
(define add2 (lambda (x) (+ x 2)))

x         ⇒ 9

(add2 3) ⇒ 5
(add2 x) ⇒ 11

x         ⇒ 9
```

Here, there is a global `x`, and there is also a local `x`, the latter introduced by procedure `add2`. The global `x` is always `9`. The local `x` gets bound to `3` in the first call to `add2` and to the value of the global `x`, ie, `9`, in the second call to `add2`. When the procedure calls return, the global `x` continues to be `9`.

The form `set!` modifies the lexical binding of a variable.

```
(set! x 20)
```

modifies the global binding of `x` from `9` to `20`, because that is the binding of `x` that is visible to `set!`. If the `set!` was inside `add2`'s body, it would have modified the local `x`:

```
(define add2
  (lambda (x)
    (set! x (+ x 2))
    x))
```

The `set!` here adds `2` to the local variable `x`, and returns that value. (In terms of effect, this procedure is indistinguishable from the previous `add2`.) We can call `add2` on the global `x`, as before:

```
(add2 x) ⇒ 22
```

(Remember global `x` is now `20`, not `9`!)

The `set!` inside `add2` affects only the local variable used by `add2`. Although the local variable `x` got its binding from the global `x`, the latter is unaffected by the `set!` to the local `x`.

```
x ⇒ 20
```

Note that we had all this discussion because we used the same identifier for a local variable and a global variable. In any text, an identifier named `x` refers to the lexically closest variable named `x`. This will *shadow* any outer or global `x`'s. Eg, in `add2`, the parameter `x` shadows the global `x`.

A procedure's body can access and modify variables in its surrounding scope provided the procedure's parameters don't shadow them. This can give some interesting programs. Eg,

```
(define counter 0)

(define bump-counter
  (lambda ()
    (set! counter (+ counter 1))
    counter))
```

The procedure `bump-counter` is a zero-argument procedure (also called a *thunk*). It introduces no local variables, and thus cannot shadow anything. Each time it is called, it modifies the *global* variable `counter` — it increments it by 1 — and returns its current value. Here are some successive calls to `bump-counter`:

```
(bump-counter) ⇒ 1
(bump-counter) ⇒ 2
(bump-counter) ⇒ 3
```

5.1 'let' and 'let*'

Local variables can be introduced without explicitly creating a procedure. The special form `let` introduces a list of local variables for use within its body:

```
(let ((x 1)
      (y 2)
      (z 3))
  (list x y z))
⇒ (1 2 3)
```

As with `lambda`, within the `let`-body, the local `x` (bound to `1`) shadows the global `x` (which is bound to `20`).

The local variable initializations — `x` to `1`; `y` to `2`; `z` to `3` — are not considered part of the `let` body. Therefore, a reference to `x` in the initialization will refer to the global, not the local `x`:

```
(let ((x 1)
      (y x))
  (+ x y))
⇒ 21
```

This is because `x` is bound to `1`, and `y` is bound to the *global* `x`, which is `20`.

Sometimes, it is convenient to have `let`'s list of lexical variables be introduced in sequence, so that the initialization of a later variable occurs in the *lexical scope* of earlier variables. The form `let*` does this:

```
(let* ((x 1)
       (y x))
  (+ x y))
⇒ 2
```

The `x` in `y`'s initialization refers to the `x` just above. The example is entirely equivalent to — and is in fact intended to be a convenient abbreviation for — the following program with nested `let`s:

```
(let ((x 1))
  (let ((y x))
    (+ x y)))
⇒ 2
```

The values bound to lexical variables can be procedures:

```
(let ((cons (lambda (x y) (+ x y))))
  (cons 1 2))
⇒ 3
```

Inside this `let` body, the lexical variable `cons` adds its arguments. Outside, `cons` continues to create dotted pairs.

5.2 fluid-let

A lexical variable is visible throughout its scope, provided it isn't shadowed. Sometimes, it is helpful to *temporarily* set a lexical variable to a certain value. For this, we use the form `fluid-let`.[2]

```
(fluid-let ((counter 99))
  (display (bump-counter)) (newline)
  (display (bump-counter)) (newline)
  (display (bump-counter)) (newline))
```

This looks similar to a `let`, but instead of shadowing the global variable `counter`, it temporarily sets it to `99` before continuing with the `fluid-let` body. Thus the `display`s in the body produce

```
100
101
102
```

After the `fluid-let` expression has evaluated, the global `counter` reverts to the value it had before the `fluid-let`.

```
counter ⇒ 3
```

Note that `fluid-let` has an entirely different effect from `let`. `fluid-let` does not introduce new lexical variables like `let` does. It modifies the bindings of *existing* lexical variables, and the modification ceases as soon as the `fluid-let` does.

To drive home this point, consider the program

```
(let ((counter 99))
  (display (bump-counter)) (newline)
  (display (bump-counter)) (newline)
  (display (bump-counter)) (newline))
```

which substitutes `let` for `fluid-let` in the previous example. The output is now

```
4
5
6
```

Ie, the global `counter`, which is initially `3`, is updated by each call to `bump-counter`. The new lexical variable `counter`, with its initialization of `99`, has no impact on the calls to `bump-counter`, because although the calls to `bump-counter` are within the scope of this local `counter`, the body of `bump-counter` isn't. The latter continues to refer to the *global* `counter`, whose final value is `6`.

[2] `fluid-let` is a nonstandard special form. See sec 8.3 for a definition of `fluid-let` in Scheme.

```
counter ⇒ 6
```

Chapter 6

Recursion

A procedure body can contain calls to other procedures, not least itself:

```
(define factorial
  (lambda (n)
    (if (= n 0) 1
        (* n (factorial (- n 1))))))
```

This *recursive* procedure calculates the *factorial* of a number. If the number is `0`, the answer is `1`. For any other number `n`, the procedure uses itself to calculate the factorial of `n - 1`, multiplies that subresult by `n`, and returns the product.

Mutually recursive procedures are also possible. The following predicates for evenness and oddness use each other:

```
(define is-even?
  (lambda (n)
    (if (= n 0) #t
        (is-odd? (- n 1)))))

(define is-odd?
  (lambda (n)
    (if (= n 0) #f
        (is-even? (- n 1)))))
```

6.1 letrec

If we wanted the above procedures as local variables, we could try to use a `let` form:

```
(let ((local-even? (lambda (n)
                     (if (= n 0) #t
                         (local-odd? (- n 1)))))
      (local-odd? (lambda (n)
                    (if (= n 0) #f
                        (local-even? (- n 1))))))
  (list (local-even? 23) (local-odd? 23)))
```

This won't quite work, because the occurrences of `local-even?` and `local-odd?` in the initializations don't refer to the lexical variables themselves. Changing the `let` to a `let*` won't work either, for while the `local-even?` inside `local-odd?`'s body refers to the correct procedure value, the `local-odd?` in `local-even?`'s body still points elsewhere.

To solve problems like this, Scheme provides the form `letrec`:

```
(letrec ((local-even? (lambda (n)
                        (if (= n 0) #t
                            (local-odd? (- n 1)))))
         (local-odd? (lambda (n)
                       (if (= n 0) #f
```

```
                              (local-even? (- n 1))))))
  (list (local-even? 23) (local-odd? 23)))
```

The lexical variables introduced by a `letrec` are visible not only in the `letrec`-body but also within all the initializations. `letrec` is thus tailor-made for defining recursive and mutually recursive local procedures.

6.2 Named 'let'

A recursive procedure defined using `letrec` can describe loops. Let's say we want to display a countdown from `10`:

```
(letrec ((countdown (lambda (i)
                      (if (= i 0) 'liftoff
                          (begin
                            (display i)
                            (newline)
                            (countdown (- i 1)))))))
  (countdown 10))
```

This outputs on the console the numbers `10` down to `1`, and returns the result `liftoff`.

Scheme allows a variant of `let` called *named* `let` to write this kind of loop more compactly:

```
(let countdown ((i 10))
  (if (= i 0) 'liftoff
      (begin
        (display i)
        (newline)
        (countdown (- i 1)))))
```

Note the presence of a variable identifying the loop immediately after the `let`. This program is equivalent to the one written with `letrec`. You may consider the named `let` to be a macro (chap 8) expanding to the `letrec` form.

6.3 Iteration

`countdown` defined above is really a recursive procedure. Scheme can define loops only through recursion. There are no special looping or iteration constructs.

Nevertheless, the loop as defined above is a *genuine* loop, in exactly the same way that other languages bill their loops. Ie, Scheme takes special care to ensure that recursion of the type used above will not generate the procedure call/return overhead.

Scheme does this by a process called *tail-call elimination*. If you look closely at the `countdown` procedure, you will note that when the recursive call occurs in `countdown`'s body, it is the *tail call*, or the very last thing done — each invocation of `countdown` either does not call itself, or when it does, it does so as its very last act. To a Scheme implementation, this makes the recursion indistinguishable from iteration. So go ahead, use recursion to write loops. It's safe.

Here's another example of a useful tail-recursive procedure:

```
(define list-position
  (lambda (o l)
    (let loop ((i 0) (l l))
      (if (null? l) #f
          (if (eqv? (car l) o) i
              (loop (+ i 1) (cdr l)))))))
```

`list-position` finds the index of the first occurrence of the object `o` in the list `l`. If the object is not found in the list, the procedure returns `#f`.

Here's yet another tail-recursive procedure, one that reverses its argument list "in place", ie, by mutating the contents of the existing list, and without allocating a new one:

```
(define reverse!
  (lambda (s)
    (let loop ((s s) (r '()))
      (if (null? s) r
          (let ((d (cdr s)))
            (set-cdr! s r)
            (loop d s))))))
```

(`reverse!` is a useful enough procedure that it is provided primitively in many Scheme dialects, eg, MzScheme and Guile.)

6.4 Mapping a procedure across a list

A special kind of iteration involves repeating the same action for each element of a list. Scheme offers two procedures for this situation: `map` and `for-each`.

The `map` procedure applies a given procedure to every element of a given list, and returns the list of the results. Eg,

```
(map add2 '(1 2 3))
⇒ (3 4 5)
```

The `for-each` procedure also applies a procedure to each element in a list, but returns void. The procedure application is done purely for any side-effects it may cause. Eg,

```
(for-each display
  (list "one " "two " "buckle my shoe"))
```

has the side-effect of displaying the strings (in the order they appear) on the console.

The procedures applied by `map` and `for-each` need not be one-argument procedures. For example, given an `n`-argument procedure, `map` takes `n` lists and applies the procedure to every set of `n` of arguments selected from across the lists. Eg,

```
(map cons '(1 2 3) '(10 20 30))
⇒ ((1 . 10) (2 . 20) (3 . 30))

(map + '(1 2 3) '(10 20 30))
⇒ (11 22 33)
```

Chapter 7

I/O

Scheme has input/output (I/O) procedures that will let you read from an input port or write to an output port. Ports can be associated with the console, files or strings.

7.1 Reading

Scheme's reader procedures take an optional input port argument. If the port is not specified, the current input port (usually the console) is assumed.

Reading can be character-, line- or s-expression-based. Each time a read is performed, the port's state changes so that the next read will read material following what was already read. If the port has no more material to be read, the reader procedure returns a specific datum called the end-of-file or eof object. This datum is the only value that satisfies the `eof-object?` predicate.

The procedure `read-char` reads the next character from the port. `read-line` reads the next line, returning it as a string (the final newline is not included). The procedure `read` reads the next s-expression.

7.2 Writing

Scheme's writer procedures take the object that is to be written and an optional output port argument. If the port is not specified, the current output port (usually the console) is assumed.

Writing can be character- or s-expression-based.

The procedure `write-char` writes the given character (without the `#\`) to the port.

The procedures `write` and `display` both write the given s-expression to the port, with one difference: `write` attempts to use a machine-readable format and `display` doesn't. Eg, `write` uses double quotes for strings and the `#\` syntax for characters. `display` doesn't.

7.3 File ports

Scheme's I/O procedures do not need a port argument if the port happens to be standard input or standard output. However, if you need these ports explicitly, the zero-argument procedures `current-input-port` and `current-output-port` furnish them. Thus,

```
(display 9)
(display 9 (current-output-port))
```

have the same behavior.

A port is associated with a file by *opening* the file. The procedure `open-input-file` takes a filename argument and returns a new input port associated with it. The procedure `open-output-file` takes a filename argument and returns a new output port associated with it. It is an error to open an input file that doesn't exist, or to open an output file that already exists.

After you have performed I/O on a port, you should close it with `close-input-port` or `close-output-port`.

In the following, assume the file `hello.txt` contains the single word `hello`.

```
(define i (open-input-file "hello.txt"))

(read-char i)
⇒ #\h

(define j (read i))

j
⇒ ello
```

Assume the file `greeting.txt` does not exist before the following programs are fed to the listener:

```
(define o (open-output-file "greeting.txt"))

(display "hello" o)
(write-char #\space o)
(display 'world o)
(newline o)

(close-output-port o)
```

The file `greeting.txt` will now contain the line:

```
hello world
```

7.3.1 Automatic opening and closing of file ports

Scheme supplies the procedures `call-with-input-file` and `call-with-output-file` that will take care of opening a port and closing it after you're done with it.

The procedure `call-with-input-file` takes a filename argument and a procedure. The procedure is applied to an input port opened on the file. When the procedure completes, its result is returned after ensuring that the port is closed.

```
(call-with-input-file "hello.txt"
  (lambda (i)
    (let* ((a (read-char i))
           (b (read-char i))
           (c (read-char i)))
      (list a b c))))
⇒ (#\h #\e #\l)
```

The procedure `call-with-output-file` does the analogous services for an output file.

7.4 String ports

It is often convenient to associate ports with strings. Thus, the procedure `open-input-string` associates a port with a given string. Reader procedures on this port will read off the string:

```
(define i (open-input-string "hello world"))

(read-char i)
⇒ #\h

(read i)
⇒ ello

(read i)
⇒ world
```

The procedure `open-output-string` creates an output port that will eventually be used to create a string:

```
(define o (open-output-string))

(write 'hello o)
(write-char #\, o)
(display " " o)
(display "world" o)
```

You can now use the procedure `get-output-string` to get the accumulated string in the string port `o`:

```
(get-output-string o)
⇒ "hello, world"
```

String ports need not be explicitly closed.

7.5 Loading files

We have already seen the procedure `load` that loads files containing Scheme code. *Loading* a file consists in evaluating in sequence every Scheme form in the file. The pathname argument given to `load` is reckoned relative to the current working directory of Scheme, which is normally the directory in which the Scheme executable was called.

Files can load other files, and this is useful in a large program spanning many files. Unfortunately, unless full pathnames are used, the argument file of a `load` is dependent on Scheme's current directory. Supplying full pathnames is not always convenient, because we would like to move the program files as a unit (preserving their relative pathnames), perhaps to many different machines.

MzScheme provides the `load-relative` procedure that greatly helps in fixing the files to be loaded. `load-relative`, like `load`, takes a pathname argument. When a `load-relative` call occurs in a file `foo.scm`, the path of its argument is reckoned from the directory of the calling file `foo.scm`. In particular, this pathname is reckoned independent of Scheme's current directory, and thus allows convenient multifile program development.

Chapter 8

Macros

Users can create their own special forms by defining *macros*. A macro is a symbol that has a *transformer procedure* associated with it. When Scheme encounters a macro-expression — ie, a form whose head is a macro —, it applies the macro's transformer to the subforms in the macro-expression, and evaluates the result of the transformation.

Ideally, a macro specifies a purely textual transformation from code text to other code text. This kind of transformation is useful for abbreviating an involved and perhaps frequently occurring textual pattern.

A macro is *defined* using the special form `define-macro` (but see sec A.4). For example, if your Scheme lacks the conditional special form `when`, you could define `when` as the following macro:

```
(define-macro when
  (lambda (test . branch)
    (list 'if test
      (cons 'begin branch))))
```

This defines a `when`-transformer that would convert a `when`-expression into the equivalent `if`-expression. With this macro definition in place, the `when`-expression

```
(when (< (pressure tube) 60)
   (open-valve tube)
   (attach floor-pump tube)
   (depress floor-pump 5)
   (detach floor-pump tube)
   (close-valve tube))
```

will be converted to another expression, the result of applying the `when`-transformer to the `when`-expression's subforms:

```
(apply
  (lambda (test . branch)
    (list 'if test
      (cons 'begin branch)))
  '((< (pressure tube) 60)
      (open-valve tube)
      (attach floor-pump tube)
      (depress floor-pump 5)
      (detach floor-pump tube)
      (close-valve tube)))
```

The transformation yields the list

```
(if (< (pressure tube) 60)
    (begin
      (open-valve tube)
      (attach floor-pump tube)
      (depress floor-pump 5)
```

```
        (detach floor-pump tube)
        (close-valve tube)))
```

Scheme will then evaluate this expression, as it would any other.

As an additional example, here is the macro-definition for `when`'s counterpart `unless`:

```
(define-macro unless
  (lambda (test . branch)
    (list 'if
          (list 'not test)
          (cons 'begin branch))))
```

Alternatively, we could invoke `when` inside `unless`'s definition:

```
(define-macro unless
  (lambda (test . branch)
    (cons 'when
          (cons (list 'not test) branch))))
```

Macro expansions can refer to other macros.

8.1 Specifying the expansion as a template

A macro transformer takes some s-expressions and produces an s-expression that will be used as a form. Typically this output is a list. In our `when` example, the output list is created using

```
(list 'if test
  (cons 'begin branch))
```

where `test` is bound to the macro's first subform, ie,

```
(< (pressure tube) 60)
```

and `branch` to the rest of the macro's subforms, ie,

```
((open-valve tube)
 (attach floor-pump tube)
 (depress floor-pump 5)
 (detach floor-pump tube)
 (close-valve tube))
```

Output lists can be quite complicated. It is easy to see that a more ambitious macro than `when` could lead to quite an elaborate construction process for the output list. In such cases, it is more convenient to specify the macro's output form as a *template*, with the macro arguments inserted at appropriate places to fill out the template for each particular use of the macro. Scheme provides the *backquote* syntax to specify such templates. Thus the expression

```
(list 'IF test
  (cons 'BEGIN branch))
```

is more conveniently written as

```
`(IF ,test
  (BEGIN ,@branch))
```

We can refashion the `when` macro-definition as:

```
(define-macro when
  (lambda (test . branch)
    `(IF ,test
         (BEGIN ,@branch))))
```

Note that the template format, unlike the earlier list construction, gives immediate visual indication of the shape of the output list. The backquote (`` ` ``) introduces a template for a list. The elements of the template appear *verbatim* in the resulting list, *except* when they are prefixed by a *comma* (‘`,`’) or a *comma-splice* (‘`,@`’). (For the purpose of illustration, we have written the verbatim elements of the template in UPPER-CASE.)

The comma and the comma-splice are used to insert the macro arguments into the template. The comma inserts the result of evaluating its following expression. The comma-splice inserts the result of evaluating its following expression after *splicing* it, ie, it removes the outermost set of parentheses. (This implies that an expression introduced by comma-splice *must* be a list.)

In our example, given the values that `test` and `branch` are bound to, it is easy to see that the template will expand to the required

```
(IF (< (pressure tube) 60)
    (BEGIN
      (open-valve tube)
      (attach floor-pump tube)
      (depress floor-pump 5)
      (detach floor-pump tube)
      (close-valve tube)))
```

8.2 Avoiding variable capture inside macros

A two-argument disjunction form, `my-or`, could be defined as follows:

```
(define-macro my-or
  (lambda (x y)
    `(if ,x ,x ,y)))
```

`my-or` takes two arguments and returns the value of the first of them that is true (ie, non-`#f`). In particular, the second argument is evaluated only if the first turns out to be false.

```
(my-or 1 2)
⇒ 1

(my-or #f 2)
⇒ 2
```

There is a problem with the `my-or` macro as it is written. It re-evaluates the first argument if it is true: once in the `if`-test, and once again in the “then” branch. This can cause undesired behavior if the first argument were to contain side-effects, eg,

```
(my-or
  (begin
    (display "doing first argument")
     (newline)
     #t)
  2)
```

displays "doing first argument" twice.

This can be avoided by storing the if-test result in a local variable:

```
(define-macro my-or
  (lambda (x y)
    `(let ((temp ,x))
       (if temp temp ,y))))
```

This is almost OK, except in the case where the second argument happens to contain the same identifier temp as used in the macro definition. Eg,

```
(define temp 3)

(my-or #f temp)
⇒ #f
```

Surely it should be 3! The fiasco happens because the macro uses a local variable temp to store the value of the first argument (#f) and the variable temp in the second argument got *captured* by the temp introduced by the macro.

To avoid this, we need to be careful in choosing local variables inside macro definitions. We could choose outlandish names for such variables and hope fervently that nobody else comes up with them. Eg,

```
(define-macro my-or
  (lambda (x y)
    `(let ((+temp ,x))
       (if +temp +temp ,y))))
```

This will work given the tacit understanding that +temp will not be used by code outside the macro. This is of course an understanding waiting to be disillusioned.

A more reliable, if verbose, approach is to use *generated symbols* that are guaranteed not to be obtainable by other means. The procedure gensym generates unique symbols each time it is called. Here is a safe definition for my-or using gensym:

```
(define-macro my-or
  (lambda (x y)
    (let ((temp (gensym)))
      `(let ((,temp ,x))
         (if ,temp ,temp ,y)))))
```

In the macros defined in this document, in order to be concise, we will not use the gensym approach. Instead, we will consider the point about variable capture as having been made, and go ahead with the less cluttered +-as-prefix approach. We will leave it to the astute reader to remember to convert these +-identifiers into gensyms in the manner outlined above.

8.3 fluid-let

Here is a definition of a rather more complicated macro, fluid-let (sec 5.2). fluid-let specifies temporary bindings for a set of already existing lexical variables. Given a fluid-let expression such as

```
(fluid-let ((x 9) (y (+ y 1)))
  (+ x y))
```

we want the expansion to be

```
(let ((OLD-X x) (OLD-Y y))
  (set! x 9)
  (set! y (+ y 1))
  (let ((RESULT (begin (+ x y))))
    (set! x OLD-X)
    (set! y OLD-Y)
    RESULT))
```

where we want the identifiers `OLD-X`, `OLD-Y`, and `RESULT` to be symbols that will not capture variables in the expressions in the `fluid-let` form.

Here is how we go about fashioning a `fluid-let` macro that implements what we want:

```
(define-macro fluid-let
  (lambda (xexe . body)
    (let ((xx (map car xexe))
          (ee (map cadr xexe))
          (old-xx (map (lambda (ig) (gensym)) xexe))
          (result (gensym)))
      `(let ,(map (lambda (old-x x) `(,old-x ,x))
                  old-xx xx)
         ,@(map (lambda (x e)
                  `(set! ,x ,e))
                xx ee)
         (let ((,result (begin ,@body)))
           ,@(map (lambda (x old-x)
                    `(set! ,x ,old-x))
                  xx old-xx)
           ,result)))))
```

The macro's arguments are: `xexe`, the list of variable/expression pairs introduced by the `fluid-let`; and `body`, the list of expressions in the body of the `fluid-let`. In our example, these are `((x 9) (y (+ y 1)))` and `((+ x y))` respectively.

The macro body introduces a bunch of local variables: `xx` is the list of the variables extracted from the variable/expression pairs. `ee` is the corresponding list of expressions. `old-xx` is a list of fresh identifiers, one for each variable in `xx`. These are used to store the *incoming* values of the `xx`, so we can revert the `xx` back to them once the `fluid-let` body has been evaluated. `result` is another fresh identifier, used to store the value of the `fluid-let` body. In our example, `xx` is `(x y)` and `ee` is `(9 (+ y 1))`. Depending on how your system implements `gensym`, `old-xx` might be the list `(GEN-63 GEN-64)`, and `result` might be `GEN-65`.

The output list is created by the macro for our given example looks like

```
(let ((GEN-63 x) (GEN-64 y))
  (set! x 9)
  (set! y (+ y 1))
  (let ((GEN-65 (begin (+ x y))))
    (set! x GEN-63)
    (set! y GEN-64)
    GEN-65))
```

which matches our requirement.

Chapter 9

Structures

Data that are naturally grouped are called *structures.* One can use Scheme's compound data types, eg, vectors or lists, to represent structures. Eg, let's say we are dealing with grouped data relevant to a (botanical) *tree.* The individual elements of the data, or *fields*, could be: *height*, *girth*, *age*, *leaf-shape*, and *leaf-color*, making a total of 5 fields. Such data could be represented as a 5-element vector. The fields could be accessed using `vector-ref` and modified using `vector-set!`. Nevertheless, we wouldn't want to be saddled with the burden of remembering which vector index corresponds to which field. That would be a thankless and error-prone activity, especially if fields get excluded or included over the course of time.

We will therefore use a Scheme macro `defstruct` to define a structure data type, which is basically a vector, but which comes with an appropriate suite of procedures for creating instances of the structure, and for accessing and modifying its fields. Thus, our `tree` structure could be defined as:

```
(defstruct tree height girth age leaf-shape leaf-color)
```

This gives us a constructor procedure named `make-tree`; accessor procedures for each field, named `tree.height`, `tree.girth`, etc; and modifier procedures for each field, named `set!tree.height`, `set!tree.girth`, etc. The constructor is used as follows:

```
(define coconut
  (make-tree 'height 30
             'leaf-shape 'frond
             'age 5))
```

The constructor's arguments are in the form of twosomes, a field name followed by its initialization. The fields can occur in any order, and may even be missing, in which case their value is undefined.

The accessor procedures are invoked as follows:

```
(tree.height coconut) ⇒ 30
(tree.leaf-shape coconut) ⇒ frond
(tree.girth coconut) ⇒ <undefined>
```

The `tree.girth` accessor returns an undefined value, because we did not specify `girth` for the `coconut` `tree`.

The modifier procedures are invoked as follows:

```
(set!tree.height coconut 40)
(set!tree.girth coconut 10)
```

If we now access these fields using the corresponding accessors, we will get the new values:

```
(tree.height coconut) ⇒ 40
(tree.girth coconut) ⇒ 10
```

9.1 Default initializations

We can have some initializations done during the definition of the structure itself, instead of per instance. Thus, we could postulate that `leaf-shape` and `leaf-color` are by default `frond` and `green` respectively. We can always override these defaults by providing explicit initialization in the `make-tree` call, or by using a field modifier after the structure instance has been created:

```
(defstruct tree height girth age
                (leaf-shape 'frond)
                (leaf-color 'green))

(define palm (make-tree 'height 60))

(tree.height palm)
⇒ 60

(tree.leaf-shape palm)
⇒ frond

(define plantain
  (make-tree 'height 7
             'leaf-shape 'sheet))

(tree.height plantain)
⇒ 7

(tree.leaf-shape plantain)
⇒ sheet

(tree.leaf-color plantain)
⇒ green
```

9.2 'defstruct' definition

The `defstruct` macro definition follows:

```
(define-macro defstruct
  (lambda (s . ff)
    (let ((s-s (symbol->string s)) (n (length ff)))
      (let* ((n+1 (+ n 1))
             (vv (make-vector n+1)))
        (let loop ((i 1) (ff ff))
          (if (<= i n)
            (let ((f (car ff)))
              (vector-set! vv i
                (if (pair? f) (cadr f) '(if #f #f)))
              (loop (+ i 1) (cdr ff)))))
        (let ((ff (map (lambda (f) (if (pair? f) (car f) f))
                       ff)))
          `(begin
             (define ,(string->symbol
                       (string-append "make-" s-s))
               (lambda fvfv
                 (let ((st (make-vector ,n+1)) (ff ',ff))
```

```
      (vector-set! st 0 ',s)
      ,@(let loop ((i 1) (r '()))
          (if (>= i n+1) r
              (loop (+ i 1)
                    (cons `(vector-set! st ,i
                                ,(vector-ref vv i))
                          r))))
      (let loop ((fvfv fvfv))
        (if (not (null? fvfv))
            (begin
              (vector-set! st
                  (+ (list-position (car fvfv) ff)
                     1)
                (cadr fvfv))
              (loop (cddr fvfv)))))
      st)))
,@(let loop ((i 1) (procs '()))
    (if (>= i n+1) procs
        (loop (+ i 1)
              (let ((f (symbol->string
                         (list-ref ff (- i 1)))))
                (cons
                 `(define ,(string->symbol
                             (string-append
                              s-s "." f))
                    (lambda (x) (vector-ref x ,i)))
                 (cons
                  `(define ,(string->symbol
                              (string-append
                               "set!" s-s "." f))
                     (lambda (x v)
                       (vector-set! x ,i v)))
                  procs))))))
(define ,(string->symbol (string-append s-s "?"))
  (lambda (x)
    (and (vector? x)
         (eqv? (vector-ref x 0) ',s))))))))))
```

Chapter 10

Alists and tables

An *association list*, or *alist*, is a Scheme list of a special format. Each element of the list is a cons cell, the car of which is called a *key*, the cdr being the *value* associated with the key. Eg,

```
((a . 1) (b . 2) (c . 3))
```

The procedure call `(assv k al)` finds the cons cell associated with key `k` in alist `al`. The keys of the alist are compared against the given `k` using the equality predicate `eqv?`. In general, though we may want a different predicate for key comparison. For instance, if the keys were case-insensitive strings, the predicate `eqv?` is not very useful.

We now define a structure called `table`, which is a souped-up alist that allows user-defined predicates on its keys. Its fields are `equ` and `alist`.

```
(defstruct table (equ eqv?) (alist '()))
```

(The default predicate is `eqv?` — as for an ordinary alist — and the alist is initially empty.)

We will use the procedure `table-get` to get the value (as opposed to the cons cell) associated with a given key. `table-get` takes a table and key arguments, followed by an optional default value that is returned if the key was not found in the table:

```
(define table-get
  (lambda (tbl k . d)
    (let ((c (lassoc k (table.alist tbl) (table.equ tbl))))
      (cond (c (cdr c))
            ((pair? d) (car d))))))
```

The procedure `lassoc`, used in `table-get`, is defined as:

```
(define lassoc
  (lambda (k al equ?)
    (let loop ((al al))
      (if (null? al) #f
          (let ((c (car al)))
            (if (equ? (car c) k) c
                (loop (cdr al))))))))
```

The procedure `table-put!` is used to update a key's value in the given table:

```
(define table-put!
  (lambda (tbl k v)
    (let ((al (table.alist tbl)))
      (let ((c (lassoc k al (table.equ tbl))))
        (if c (set-cdr! c v)
            (set!table.alist tbl (cons (cons k v) al)))))))
```

The procedure `table-for-each` calls the given procedure on every key/value pair in the table

```
(define table-for-each
  (lambda (tbl p)
    (for-each
     (lambda (c)
       (p (car c) (cdr c)))
     (table.alist tbl))))
```

Chapter 11

System interface

Useful Scheme programs often need to interact with the underlying operating system.

11.1 Checking for and deleting files

`file-exists?` checks if its argument string names a file. `delete-file` deletes its argument file. These procedures are not part of the Scheme standard, but are available in most implementations. These procedures work reliably only for files that are not directories. (Their behavior on directories is dialect-specific.)

`file-or-directory-modify-seconds` returns the time when its argument file or directory was last modified. Time is reckoned in seconds from 12 AM GMT, 1 January 1970. Eg,

```
(file-or-directory-modify-seconds "hello.scm")
⇒ 893189629
```

assuming that the file `hello.scm` was last messed with sometime on 21 April 1998.

11.2 Calling operating-system commands

The `system` procedure executes its argument string as an operating-system command. It returns true if the command executed successfully with an exit status 0, and false if it failed to execute or exited with a non-zero status. Any output generated by the command goes to standard output.

```
(system "ls")
;lists current directory

(define fname "spot")

(system (string-append "test -f " fname))
;tests if file ‘spot’ exists

(system (string-append "rm -f " fname))
;removes ‘spot’
```

The last two forms are equivalent to

```
(file-exists? fname)

(delete-file fname)
```

11.3 Environment variables

The `getenv` procedure returns the setting of an operating-system environment variable. Eg,

```
(getenv "HOME")
⇒ "/home/dorai"

(getenv "SHELL")
⇒ "/bin/bash"
```

Chapter 12

Objects and classes

A *class* describes a collection of *objects* that share behavior. The objects described by a class are called the *instances* of the class. The class specifies the names of the *slots* that the instance has, although it is up to the instance to populate these slots with particular values. The class also specifies the *methods* that can be applied to its instances. Slot values can be anything, but method values must be procedures.

Classes are hierarchical. Thus, a class can be a *subclass* of another class, which is called its *superclass*. A subclass not only has its own *direct* slots and methods, but also inherits all the slots and methods of its superclass. If a class has a slot or method that has the same name as its superclass's, then the subclass's slot or method is the one that is retained.

12.1 A simple object system

Let us now implement a basic object system in Scheme. We will allow only one superclass per class (*single inheritance*). If we don't want to specify a superclass, we will use `#t` as a "zero" superclass, one that has neither slots nor methods. The superclass of `#t` is deemed to be itself.

As a first approximation, it is useful to define classes using a struct called `standard-class`, with fields for the slot names, the superclass, and the methods. The first two fields we will call `slots` and `superclass` respectively. We will use *two* fields for methods, a `method-names` field that will hold the list of names of the class's methods, and a `method-vector` field that will hold the vector of the values of the class's methods.[3] Here is the definition of the `standard-class`:

```
(defstruct standard-class
  slots superclass method-names method-vector)
```

We can use `make-standard-class`, the maker procedure of `standard-class`, to create a new class. Eg,

```
(define trivial-bike-class
  (make-standard-class
   'superclass #t
   'slots '(frame parts size)
   'method-names '()
   'method-vector #()))
```

This is a very simple class. More complex classes will have non-trivial superclasses and methods, which will require a lot of standard initialization that we would like to hide within the class creation process. We will therefore define a macro called `create-class` that will make the appropriate call to `make-standard-class`.

[3] We could in theory define methods also as slots (whose values happen to be procedures), but there is a good reason not to. The instances of a class share methods but in general differ in their slot values. In other words, methods can be included in the class definition and don't have to be allocated per instance as slots have to be.

```
(define-macro create-class
  (lambda (superclass slots . methods)
    `(create-class-proc
       ,superclass
       (list ,@(map (lambda (slot) `',slot) slots))
       (list ,@(map (lambda (method) `',(car method)) methods))
       (vector ,@(map (lambda (method) `,(cadr method)) methods)))))
```

We will defer the definition of the `create-class-proc` procedure to later.

The procedure `make-instance` creates an *instance* of a class by generating a fresh vector based on information enshrined in the class. The format of the instance vector is very simple: Its first element will refer to the class, and its remaining elements will be slot values. `make-instance`'s arguments are the class followed by a sequence of twosomes, where each twosome is a slot name and the value it assumes in the instance.

```
(define make-instance
  (lambda (class . slot-value-twosomes)

    ;Find `n', the number of slots in `class'.
    ;Create an instance vector of length `n + 1',
    ;because we need one extra element in the instance
    ;to contain the class.

    (let* ((slotlist (standard-class.slots class))
           (n (length slotlist))
           (instance (make-vector (+ n 1))))
      (vector-set! instance 0 class)

      ;Fill each of the slots in the instance
      ;with the value as specified in the call to
      ;`make-instance'.

      (let loop ((slot-value-twosomes slot-value-twosomes))
        (if (null? slot-value-twosomes) instance
            (let ((k (list-position (car slot-value-twosomes)
                                    slotlist)))
              (vector-set! instance (+ k 1)
                (cadr slot-value-twosomes))
              (loop (cddr slot-value-twosomes))))))))
```

Here is an example of instantiating a class:

```
(define my-bike
  (make-instance trivial-bike-class
                 'frame 'cromoly
                 'size '18.5
                 'parts 'alivio))
```

This binds `my-bike` to the instance

```
#(<trivial-bike-class> cromoly 18.5 alivio)
```

where `<trivial-bike-class>` is a Scheme datum (another vector) that is the value of `trivial-bike-class`, as defined above.

The procedure `class-of` returns the class of an instance:

```
(define class-of
  (lambda (instance)
    (vector-ref instance 0)))
```

This assumes that `class-of`'s argument will be a class instance, ie, a vector whose first element points to some instantiation of the `standard-class`. We probably want to make `class-of` return an appropriate value for any kind of Scheme object we feed to it.

```
(define class-of
  (lambda (x)
    (if (vector? x)
        (let ((n (vector-length x)))
          (if (>= n 1)
              (let ((c (vector-ref x 0)))
                (if (standard-class? c) c #t))
              #t))
        #t)))
```

The class of a Scheme object that isn't created using `standard-class` is deemed to be `#t`, the zero class.

The procedures `slot-value` and `set!slot-value` access and mutate the values of a class instance:

```
(define slot-value
  (lambda (instance slot)
    (let* ((class (class-of instance))
           (slot-index
             (list-position slot (standard-class.slots class))))
      (vector-ref instance (+ slot-index 1)))))

(define set!slot-value
  (lambda (instance slot new-val)
    (let* ((class (class-of instance))
           (slot-index
             (list-position slot (standard-class.slots class))))
      (vector-set! instance (+ slot-index 1) new-val))))
```

We are now ready to tackle the definition of `create-class-proc`. This procedure takes a superclass, a list of slots, a list of method names, and a vector of methods and makes the appropriate call to `make-standard-class`. The only tricky part is the value to be given to the `slots` field. It can't be just the slots argument supplied via `create-class`, for a class must include the slots of its superclass as well. We must append the supplied slots to the superclass's slots, making sure that we don't have duplicate slots.

```
(define create-class-proc
  (lambda (superclass slots method-names method-vector)
    (make-standard-class
     'superclass superclass
     'slots
     (let ((superclass-slots
            (if (not (eqv? superclass #t))
                (standard-class.slots superclass)
                '())))
       (if (null? superclass-slots) slots
           (delete-duplicates
```

```
                (append slots superclass-slots))))
          'method-names method-names
          'method-vector method-vector)))
```

The procedure `delete-duplicates` called on a list `s`, returns a new list that only includes the *last* occurrence of each element of `s`.

```
(define delete-duplicates
  (lambda (s)
    (if (null? s) s
        (let ((a (car s)) (d (cdr s)))
          (if (memv a d) (delete-duplicates d)
              (cons a (delete-duplicates d)))))))
```

Now to the application of methods. We invoke the method on an instance by using the procedure `send`. `send`'s arguments are the method name, followed by the instance, followed by any arguments the method has in addition to the instance itself. Since methods are stored in the instance's class instead of the instance itself, `send` will search the instance's class for the method. If the method is not found there, it is looked for in the class's superclass, and so on further up the superclass chain:

```
(define send
  (lambda (method instance . args)
    (let ((proc
           (let loop ((class (class-of instance)))
             (if (eqv? class #t) (error 'send)
                 (let ((k (list-position
                           method
                           (standard-class.method-names class))))
                   (if k
                       (vector-ref (standard-class.method-vector class)
k)
                       (loop (standard-class.superclass class))))))))
      (apply proc instance args))))
```

We can now define some more interesting classes:

```
(define bike-class
  (create-class
   #t
   (frame size parts chain tires)
   (check-fit (lambda (me inseam)
                (let ((bike-size (slot-value me 'size))
                      (ideal-size (* inseam 3/5)))
                  (let ((diff (- bike-size ideal-size)))
                    (cond ((<= -1 diff 1) 'perfect-fit)
                          ((<= -2 diff 2) 'fits-well)
                          ((< diff -2) 'too-small)
                          ((> diff 2) 'too-big))))))))
```

Here, `bike-class` includes a method `check-fit`, that takes a bike and an inseam measurement and reports on the fit of the bike for a person of that inseam.

Let's redefine `my-bike`:

```
(define my-bike
  (make-instance bike-class
                 'frame 'titanium ; I wish
                 'size 21
                 'parts 'ultegra
                 'chain 'sachs
                 'tires 'continental))
```

To check if this will fit someone with inseam 32:

```
(send 'check-fit my-bike 32)
```

We can subclass `bike-class`.

```
(define mtn-bike-class
  (create-class
    bike-class
    (suspension)
    (check-fit (lambda (me inseam)
                 (let ((bike-size (slot-value me 'size))
                       (ideal-size (- (* inseam 3/5) 2)))
                   (let ((diff (- bike-size ideal-size)))
                     (cond ((<= -2 diff 2) 'perfect-fit)
                           ((<= -4 diff 4) 'fits-well)
                           ((< diff -4) 'too-small)
                           ((> diff 4) 'too-big))))))))
```

`mtn-bike-class` adds a slot called `suspension` and uses a slightly different definition for the method `check-fit`.

12.2 Classes are instances too

It cannot have escaped the astute reader that classes themselves look like they could be the instances of some class (a *metaclass*, if you will). Note that all classes have some common behavior: each of them has slots, a superclass, a list of method names, and a method vector. `make-instance` looks like it could be their shared method. This suggests that we could specify this common behavior by another class (which itself should, of course, be a class instance too).

In concrete terms, we could rewrite our class implementation to itself make use of the object-oriented approach, provided we make sure we don't run into chicken-and-egg problems. In effect, we will be getting rid of the `class` struct and its attendant procedures and rely on the rest of the machinery to define classes as objects.

Let us identify `standard-class` as the class of which other classes are instances of. In particular, `standard-class` must be an instance of itself. What should `standard-class` look like?

We know `standard-class` is an instance, and we are representing instances by vectors. So it is a vector whose first element holds its class, ie, itself, and whose remaining elements are slot values. We have identified four slots that all classes must have, so `standard-class` is a 5-element vector.

```
(define standard-class
  (vector 'value-of-standard-class-goes-here
          (list 'slots
                'superclass
                'method-names
                'method-vector)
```

```
              #t
              '(make-instance)
              (vector make-instance)))
```

Note that the `standard-class` vector is incompletely filled in: the symbol `value-of-standard-class-goes-here` functions as a placeholder. Now that we have defined a `standard-class` value, we can use it to identify its own class, which is itself:

```
(vector-set! standard-class 0 standard-class)
```

Note that we cannot rely on procedures based on the `class` struct anymore. We should replace all calls of the form

```
(standard-class? x)
(standard-class.slots c)
(standard-class.superclass c)
(standard-class.method-names c)
(standard-class.method-vector c)
(make-standard-class ...)
```

by

```
(and (vector? x) (eqv? (vector-ref x 0) standard-class))
(vector-ref c 1)
(vector-ref c 2)
(vector-ref c 3)
(vector-ref c 4)
(send 'make-instance standard-class ...)
```

12.3 Multiple inheritance

It is easy to modify the object system to allow classes to have more than one superclass. We redefine the `standard-class` to have a slot called `class-precedence-list` instead of `superclass`. The `class-precedence-list` of a class is the list of *all* its superclasses, not just the *direct* superclasses specified during the creation of the class with `create-class`. The name implies that the superclasses are listed in a particular order, where superclasses occurring toward the front of the list have precedence over the ones in the back of the list.

```
(define standard-class
  (vector 'value-of-standard-class-goes-here
          (list 'slots 'class-precedence-list 'method-names 'method-
vector)
          '()
          '(make-instance)
          (vector make-instance)))
```

Not only has the list of slots changed to include the new slot, but the erstwhile `superclass` slot is now `()` instead of `#t`. This is because the `class-precedence-list` of `standard-class` must be a list. We could have had its value be `(#t)`, but we will not mention the zero class since it is in every class's `class-precedence-list`.

The `create-class` macro has to modified to accept a list of direct superclasses instead of a solitary superclass:

```
(define-macro create-class
  (lambda (direct-superclasses slots . methods)
    `(create-class-proc
      (list ,@(map (lambda (su) `,su) direct-superclasses))
      (list ,@(map (lambda (slot) `',slot) slots))
      (list ,@(map (lambda (method) `',(car method)) methods))
      (vector ,@(map (lambda (method) `,(cadr method)) methods))
      )))
```

The create-class-proc must calculate the class precedence list from the supplied direct superclasses, and the slot list from the class precedence list:

```
(define create-class-proc
  (lambda (direct-superclasses slots method-names method-vector)
    (let ((class-precedence-list
           (delete-duplicates
            (append-map
             (lambda (c) (vector-ref c 2))
             direct-superclasses))))
      (send 'make-instance standard-class
            'class-precedence-list class-precedence-list
            'slots
            (delete-duplicates
             (append slots (append-map
                            (lambda (c) (vector-ref c 1))
                            class-precedence-list)))
            'method-names method-names
            'method-vector method-vector))))
```

The procedure append-map is a composition of append and map:

```
(define append-map
  (lambda (f s)
    (let loop ((s s))
      (if (null? s) '()
          (append (f (car s))
                  (loop (cdr s)))))))
```

The procedure send has to search through the class precedence list left to right when it hunts for a method.

```
(define send
  (lambda (method-name instance . args)
    (let ((proc
           (let ((class (class-of instance)))
             (if (eqv? class #t) (error 'send)
                 (let loop ((class class)
                            (superclasses (vector-ref class 2)))
                   (let ((k (list-position
                             method-name
                             (vector-ref class 3))))
                     (cond (k (vector-ref
                               (vector-ref class 4) k))
                           ((null? superclasses) (error 'send))
                           (else (loop (car superclasses)
                                       (cdr superclasses))))
```

```
                  ))))))
        (apply proc instance args))))
```

Chapter 13

Jumps

One of the signal features of Scheme is its support for jumps or *nonlocal control*. Specifically, Scheme allows program control to jump to *arbitrary* locations in the program, in contrast to the more restrained forms of program control flow allowed by conditionals and procedure calls. Scheme's nonlocal control operator is a procedure named `call-with-current-continuation`. We will see how this operator can be used to create a breathtaking variety of control idioms.

13.1 call-with-current-continuation

The operator `call-with-current-continuation` *call*s its argument, which must be a unary procedure, *with* a value called the "*current continuation*". If nothing else, this explains the name of the operator. But it is a long name, and is often abbreviated `call/cc`.[4]

The current continuation at any point in the execution of a program is an abstraction of the *rest of the program*. Thus in the program

```
(+ 1 (call/cc
       (lambda (k)
         (+ 2 (k 3)))))
```

the rest of the program, from the point of view of the `call/cc`-application, is the following program-with-a-hole (with `[]` representing the hole):

```
(+ 1 [])
```

In other words, this continuation is a program that will add `1` to whatever is used to fill its hole.

This is what the argument of `call/cc` is *called with*. Remember that the argument of `call/cc` is the procedure

```
(lambda (k)
  (+ 2 (k 3)))
```

This procedure's body applies the continuation (bound now to the parameter `k`) to the argument `3`. This is when the unusual aspect of the continuation springs to the fore. The continuation call abruptly abandons its own computation and replaces it with the rest of the program saved in `k`! In other words, the part of the procedure involving the addition of `2` is jettisoned, and `k`'s argument `3` is sent directly to the program-with-the-hole:

```
(+ 1 [])
```

The program now running is simply

```
(+ 1 3)
```

[4] If your Scheme does not already have this abbreviation, include `(define call/cc call-with-current-continuation)` in your initialization code and protect yourself from RSI.

which returns 4. In sum,

```
(+ 1 (call/cc
       (lambda (k)
         (+ 2 (k 3)))))
⇒ 4
```

The above illustrates what is called an *escaping* continuation, one used to exit out of a computation (here: the `(+ 2 [])` computation). This is a useful property, but Scheme's continuations can also be used to return to previously abandoned contexts, and indeed to invoke them many times. The "rest of the program" enshrined in a continuation is available whenever and how many ever times we choose to recall it, and this is what contributes to the great and sometimes confusing versatility of `call/cc`. As a quick example, type the following at the listener:

```
(define r #f)

(+ 1 (call/cc
       (lambda (k)
         (set! r k)
         (+ 2 (k 3)))))
⇒ 4
```

The latter expression returns 4 as before. The difference between this use of `call/cc` and the previous example is that here we also store the continuation `k` in a global variable `r`.

Now we have a permanent record of the continuation in `r`. If we call it on a number, it will return that number incremented by `1`:

```
(r 5)
⇒ 6
```

Note that `r` will abandon its own continuation, which is better illustrated by embedding the call to `r` inside some context:

```
(+ 3 (r 5))
⇒ 6
```

The continuations provided by `call/cc` are thus *abortive* continuations.

13.2 Escaping continuations

Escaping continuations are the simplest use of `call/cc` and are very useful for programming procedure or loop exits. Consider a procedure `list-product` that takes a list of numbers and multiplies them. A straightforward recursive definition for `list-product` is:

```
(define list-product
  (lambda (s)
    (let recur ((s s))
      (if (null? s) 1
          (* (car s) (recur (cdr s)))))))
```

There is a problem with this solution. If one of the elements in the list is `0`, and if there are many elements after `0` in the list, then the answer is a foregone conclusion. Yet, the code will have us go through many fruitless recursive calls to `recur` before producing the answer. This is where an escape continuation comes in handy. Using `call/cc`, we can rewrite the procedure as:

```
(define list-product
  (lambda (s)
    (call/cc
      (lambda (exit)
        (let recur ((s s))
          (if (null? s) 1
              (if (= (car s) 0) (exit 0)
                  (* (car s) (recur (cdr s))))))))))
```

If a `0` element is encountered, the continuation `exit` is called with `0`, thereby avoiding further calls to `recur`.

13.3 Tree matching

A more involved example of continuation usage is the problem of determining if two trees (arbitrarily nested dotted pairs) have the same *fringe*, ie, the same elements (or *leaves*) in the same sequence. Eg,

```
(same-fringe? '(1 (2 3)) '((1 2) 3))
⇒ #t

(same-fringe? '(1 2 3) '(1 (3 2)))
⇒ #f
```

The purely functional approach is to flatten both trees and check if the results match.

```
(define same-fringe?
  (lambda (tree1 tree2)
    (let loop ((ftree1 (flatten tree1))
               (ftree2 (flatten tree2)))
      (cond ((and (null? ftree1) (null? ftree2)) #t)
            ((or (null? ftree1) (null? ftree2)) #f)
            ((eqv? (car ftree1) (car ftree2))
             (loop (cdr ftree1) (cdr ftree2)))
            (else #f)))))

(define flatten
  (lambda (tree)
    (cond ((null? tree) '())
          ((pair? (car tree))
           (append (flatten (car tree))
                   (flatten (cdr tree))))
          (else
           (cons (car tree)
                 (flatten (cdr tree)))))))
```

However, this traverses the trees completely to flatten them, and then again till it finds non-matching elements. Furthermore, even the best flattening algorithms will require `cons`es equal to the total number of leaves. (Destructively modifying the input trees is not an option.)

We can use `call/cc` to solve the problem without needless traversal and without any `cons`ing. Each tree is mapped to a *generator*, a procedure with internal state that successively produces the leaves of the tree in the left-to-right order that they occur in the tree.

```
(define tree->generator
  (lambda (tree)
    (let ((caller '*))
      (letrec
          ((generate-leaves
            (lambda ()
              (let loop ((tree tree))
                (cond ((null? tree) 'skip)
                      ((pair? tree)
                       (loop (car tree))
                       (loop (cdr tree)))
                      (else
                       (call/cc
                        (lambda (rest-of-tree)
                          (set! generate-leaves
                            (lambda ()
                              (rest-of-tree 'resume)))
                          (caller tree))))))
              (caller '()))))
        (lambda ()
          (call/cc
           (lambda (k)
             (set! caller k)
             (generate-leaves))))))))
```

When a generator created by `tree->generator` is called, it will store the continuation of its call in `caller`, so that it can know who to send the leaf to when it finds it. It then calls an internal procedure called `generate-leaves` which runs a loop traversing the tree from left to right. When the loop encounters a leaf, it will use `caller` to return the leaf as the generator's result, but it will remember to store the rest of the loop (captured as a `call/cc` continuation) in the `generate-leaves` variable. The next time the generator is called, the loop is resumed where it left off so it can hunt for the next leaf.

Note that the last thing `generate-leaves` does, after the loop is done, is to return the empty list to the `caller`. Since the empty list is not a valid leaf value, we can use it to tell that the generator has no more leaves to generate.

The procedure `same-fringe?` maps each of its tree arguments to a generator, and then calls these two generators alternately. It announces failure as soon as two non-matching leaves are found:

```
(define same-fringe?
  (lambda (tree1 tree2)
    (let ((gen1 (tree->generator tree1))
          (gen2 (tree->generator tree2)))
      (let loop ()
        (let ((leaf1 (gen1))
              (leaf2 (gen2)))
          (if (eqv? leaf1 leaf2)
              (if (null? leaf1) #t (loop))
              #f))))))
```

It is easy to see that the trees are traversed at most once, and in case of mismatch, the traversals extend only upto the leftmost mismatch. `cons` is not used.

13.4 Coroutines

The generators used above are interesting generalizations of the procedure concept. Each time the generator is called, it resumes its computation, and when it has a result for its caller returns it, but only after storing its continuation in an internal variable so the generator can resumed again. We can generalize generators further, so that they can mutually resume each other, sending results back and forth amongst themselves. Such procedures are called *coroutines* [HFW84].

We will view a coroutine as a unary procedure, whose body can contain `resume` calls. `resume` is a two-argument procedure used by a coroutine to resume another coroutine with a transfer value.

A coroutine (let's call it `A`) has an internal variable called `local-control-state` that stores, at any point, the remaining computation of the coroutine. Initially this is the entire coroutine computation. When `resume` is called — ie, invoking another coroutine `B` — the current coroutine will update its `local-control-state` value to the rest of itself, stop itself, and then jump to the `resume`d coroutine `B`. When coroutine `A` is itself `resume`d at some later point, its computation will proceed from the continuation stored in its `local-control-state`.

```
(define-macro coroutine
  (lambda (x . body)
    `(letrec ((local-control-state
               (lambda (,x) ,@body))
              (resume
               (lambda (c v)
                 (call/cc
                  (lambda (k)
                    (set! local-control-state k)
                    (c v))))))
       (lambda (v)
         (local-control-state v)))))
```

13.4.1 Tree-matching with coroutines

Tree-matching is further simplified using coroutines. The matching process is coded as a coroutine that depends on two other coroutines to supply the leaves of the respective trees:

```
(define make-matcher-coroutine
  (lambda (tree-cor-1 tree-cor-2)
    (coroutine dont-need-an-init-arg
      (let loop ()
        (let ((leaf1 (resume tree-cor-1 'get-a-leaf))
              (leaf2 (resume tree-cor-2 'get-a-leaf)))
          (if (eqv? leaf1 leaf2)
              (if (null? leaf1) #t (loop))
              #f))))))
```

The leaf-generator coroutines remember who to send their leaves to:

```
(define make-leaf-gen-coroutine
  (lambda (tree matcher-cor)
    (coroutine dont-need-an-init-arg
      (let loop ((tree tree))
        (cond ((null? tree) 'skip)
              ((pair? tree)
```

```
                (loop (car tree))
                (loop (cdr tree)))
               (else
                (resume matcher-cor tree))))
        (resume matcher-cor '())))))
```

The `same-fringe?` procedure can now *almost* be written as

```
(define same-fringe?
  (lambda (tree1 tree2)
    (letrec ((tree-cor-1
              (make-leaf-gen-coroutine
               tree1
               matcher-cor))
             (tree-cor-2
              (make-leaf-gen-coroutine
               tree2
               matcher-cor))
             (matcher-cor
              (make-matcher-coroutine
               tree-cor-1
               tree-cor-2)))
      (matcher-cor 'start-ball-rolling))))
```

Unfortunately, Scheme's `letrec` can resolve mutually recursive references amongst the lexical variables it introduces *only* if such variable references are wrapped inside a `lambda`. And so we write:

```
(define same-fringe?
  (lambda (tree1 tree2)
    (letrec ((tree-cor-1
              (make-leaf-gen-coroutine
               tree1
               (lambda (v) (matcher-cor v))))
             (tree-cor-2
              (make-leaf-gen-coroutine
               tree2
               (lambda (v) (matcher-cor v))))
             (matcher-cor
              (make-matcher-coroutine
               (lambda (v) (tree-cor-1 v))
               (lambda (v) (tree-cor-2 v)))))
      (matcher-cor 'start-ball-rolling))))
```

Note that `call/cc` is not called directly at all in this rewrite of `same-fringe?`. All the continuation manipulation is handled for us by the `coroutine` macro.

Chapter 14

Nondeterminism

McCarthy's nondeterministic operator `amb` [McC67, Cli82, ZMC87] is as old as Lisp itself, although it is present in no Lisp. `amb` takes zero or more expressions, and makes a nondeterministic (or "ambiguous") choice among them, preferring those choices that cause the program to converge meaningfully. Here we will explore an embedding of `amb` in Scheme that makes a depth-first selection of the ambiguous choices, and uses Scheme's control operator `call/cc` to backtrack for alternate choices. The result is an elegant backtracking strategy that can be used for searching problem spaces directly in Scheme without recourse to an extended language. The embedding recalls the continuation strategies used to implement Prolog-style logic programming [Hay87, Fel85], but is sparer because the operator provided is much like a Scheme boolean operator, does not require special contexts for its use, and does not rely on linguistic infrastructure such as logic variables and unification.

14.1 Description of 'amb'

An accessible description of `amb` and many example uses are found in the premier Scheme textbook SICP [AS96]. Informally, `amb` takes zero or more expressions and *nondeterministically* returns the value of *one* of them. Thus,

```
(amb 1 2)
```

may evaluate to 1 *or* 2.

`amb` called with *no* expressions has no value to return, and is considered to *fail*. Thus,

```
(amb)
```
$\rightarrow$*ERROR!!!* `amb tree exhausted`

(We will examine the wording of the error message later.)

In particular, `amb` is required to return a value if at least one its subexpressions converges, ie, doesn't fail. Thus,

```
(amb 1 (amb))
```

and

```
(amb (amb) 1)
```

both return 1.

Clearly, `amb` cannot simply be equated to its first subexpression, since it has to return a *non-failing* value, if this is at all possible. However, this is not all: The bias for convergence is more stringent than a merely local choice of `amb`'s subexpressions. `amb` should furthermore return *that* convergent value that makes the *entire program* converge. In denotational parlance, `amb` is an *angelic* operator.

For example,

```
(amb #t #f)
```

may return either `#t` or `#f`, but in the program

```
(if (amb #t #f)
    1
    (amb))
```

the first `amb`-expression *must* return `#t`. If it returned `#f`, the `if`'s "else" branch would be chosen, which causes the entire program to fail.

14.2 Implementing 'amb' in Scheme

In our implementation of `amb`, we will favour `amb`'s subexpressions from left to right. Ie, the first subexpression is chosen, and if it leads to overall failure, the second is picked, and so on. `amb`s occurring later in the control flow of the program are searched for alternates before backtracking to previous `amb`s. In other words, we perform a *depth-first* search of the `amb` *choice tree*, and whenever we brush against failure, we backtrack to the most recent node of the tree that offers a further choice. (This is called *chronological backtracking*.)

We first define a mechanism for setting the base failure continuation:

```
(define amb-fail '*)

(define initialize-amb-fail
  (lambda ()
    (set! amb-fail
      (lambda ()
        (error "amb tree exhausted")))))

(initialize-amb-fail)
```

When `amb` fails, it invokes the continuation bound at the time to `amb-fail`. This is the continuation invoked when all the alternates in the `amb` choice tree have been tried and were found to fail.

We define `amb` as a macro that accepts an indefinite number of subexpressions.

```
(define-macro amb
  (lambda alts...
    `(let ((+prev-amb-fail amb-fail))
       (call/cc
        (lambda (+sk)

          ,@(map (lambda (alt)
                   `(call/cc
                     (lambda (+fk)
                       (set! amb-fail
                         (lambda ()
                           (set! amb-fail +prev-amb-fail)
                           (+fk 'fail)))
                       (+sk ,alt))))
                 alts...)

          (+prev-amb-fail))))))
```

A call to `amb` first stores away, in `+prev-amb-fail`, the `amb-fail` value that was current at the time of entry. This is because the `amb-fail` variable will be set to different failure continuations as the various alternates are tried.

We then capture the `amb`'s *entry* continuation `+sk`, so that when one of the alternates evaluates to a non-failing value, it can immediately exit the `amb`.

Each alternate `alt` is tried in sequence (the implicit-`begin` sequence of Scheme).

First, we capture the current continuation `+fk`, wrap it in a procedure and set `amb-fail` to that procedure. The alternate is then evaluated as `(+sk alt)`. If `alt` evaluates without failure, its return value is fed to the continuation `+sk`, which immediately exits the `amb` call. If `alt` fails, it calls `amb-fail`. The first duty of `amb-fail` is to reset `amb-fail` to the value it had at the time of entry. It then invokes the failure continuation `+fk`, which causes the next alternate, if any, to be tried.

If all alternates fail, the `amb-fail` at `amb` entry, which we had stored in `+prev-amb-fail`, is called.

14.3 Using 'amb' in Scheme

To choose a number between 1 and 10, one could say

```
(amb 1 2 3 4 5 6 7 8 9 10)
```

To be sure, as a program, this will give 1, but depending on the context, it could return any of the mentioned numbers.

The procedure `number-between` is a more abstract way to generate numbers from a given `lo` to a given `hi` (inclusive):

```
(define number-between
  (lambda (lo hi)
    (let loop ((i lo))
      (if (> i hi) (amb)
          (amb i (loop (+ i 1)))))))
```

Thus `(number-between 1 6)` will first generate 1. Should that fail, the `loop` iterates, producing 2. Should *that* fail, we get 3, and so on, until 6. After 6, `loop` is called with the number 7, which being more than 6, invokes `(amb)`, which causes final failure. (Recall that `(amb)` by itself guarantees failure.) At this point, the program containing the call to `(number-between 1 6)` will backtrack to the chronologically previous `amb`-call, and try to satisfy *that* call in another fashion.

The guaranteed failure of `(amb)` can be used to program *assertions*.

```
(define assert
  (lambda (pred)
    (if (not pred) (amb))))
```

The call `(assert pred)` insists that `pred` be true. Otherwise it will cause the current `amb` choice point to fail.[5]

Here is a procedure using `assert` that generates a prime less than or equal to its argument `hi`:

```
(define gen-prime
  (lambda (hi)
    (let ((i (number-between 2 hi)))
      (assert (prime? i))
      i)))
```

[5] SICP names this procedure `require`. We use the identifier `assert` in order to avoid confusion with the popular if informal use of the identifier `require` for something else, viz, an operator that loads code modules on a per-need basis.

This seems devilishly simple, except that when called as a program with any number (say 20), it will produce the uninteresting first solution, ie, 2.

We would certainly like to get *all* the solutions, not just the first. In this case, we may want *all* the primes below 20. One way is to explicitly call the failure continuation left after the program has produced its first solution. Thus,

```
(amb)
=> 3
```

This leaves yet another failure continuation, which can be called again for yet another solution:

```
(amb)
=> 5
```

The problem with this method is that the program is initially called at the Scheme prompt, and successive solutions are also obtained by calling `amb` at the Scheme prompt. In effect, we are using different programs (we cannot predict how many!), carrying over information from a previous program to the next. Instead, we would like to be able to get these solutions as the return value of a form that we can call in any context. To this end, we define the `bag-of` macro, which returns all the successful instantiations of its argument. (If the argument never succeeds, `bag-of` returns the empty list.) Thus, we could say,

```
(bag-of
  (gen-prime 20))
```

and it would return

```
(2 3 5 7 11 13 17 19)
```

The `bag-of` macro is defined as follows:

```
(define-macro bag-of
  (lambda (e)
    `(let ((+prev-amb-fail amb-fail)
           (+results '()))
       (if (call/cc
             (lambda (+k)
               (set! amb-fail (lambda () (+k #f)))
               (let ((+v ,e))
                 (set! +results (cons +v +results))
                 (+k #t))))
           (amb-fail))
       (set! amb-fail +prev-amb-fail)
       (reverse! +results))))
```

`bag-of` first saves away its entry `amb-fail`. It redefines `amb-fail` to a local continuation `+k` created within an `if`-test. Inside the test, the `bag-of` argument `e` is evaluated. If `e` succeeds, its result is collected into a list called `+results`, and the local continuation is called with the value `#t`. This causes the `if`-test to succeed, causing `e` to be *retried* at its next backtrack point. More results for `e` are obtained this way, and they are all collected into `+results`.

Finally, when `e` fails, it will call the base `amb-fail`, which is simply a call to the local continuation with the value `#f`. This pushes control past the `if`. We restore `amb-fail` to its pre-entry value, and return the `+results`. (The `reverse!` is simply to produce the results in the order in which they were generated.)

14.4 Logic puzzles

The power of depth-first search coupled with backtracking becomes obvious when applied to solving logic puzzles. These problems are extraordinarily difficult to solve procedurally, but can be solved concisely and declaratively with `amb`, without taking anything away from the charm of solving the puzzle.

14.4.1 The Kalotan puzzle

The Kalotans are a tribe with a peculiar quirk.[6] Their males always tell the truth. Their females never make two consecutive true statements, or two consecutive untrue statements.

An anthropologist (let's call him Worf) has begun to study them. Worf does not yet know the Kalotan language. One day, he meets a Kalotan (heterosexual) couple and their child Kibi. Worf asks Kibi: "Are you a boy?" Kibi answers in Kalotan, which of course Worf doesn't understand.

Worf turns to the parents (who know English) for explanation. One of them says: "Kibi said: 'I am a boy.' " The other adds: "Kibi is a girl. Kibi lied."

Solve for the sex of the parents and Kibi.

—

The solution consists in introducing a bunch of variables, allowing them to take a choice of values, and enumerating the conditions on them as a sequence of `assert` expressions.

The variables: `parent1`, `parent2`, and `kibi` are the sexes of the parents (in order of appearance) and Kibi; `kibi-self-desc` is the sex Kibi claimed to be (in Kalotan); `kibi-lied?` is the boolean on whether Kibi's claim was a lie.

```
(define solve-kalotan-puzzle
  (lambda ()
    (let ((parent1 (amb 'm 'f))
          (parent2 (amb 'm 'f))
          (kibi (amb 'm 'f))
          (kibi-self-desc (amb 'm 'f))
          (kibi-lied? (amb #t #f)))
      (assert
       (distinct? (list parent1 parent2)))
      (assert
       (if (eqv? kibi 'm)
           (not kibi-lied?)))
      (assert
       (if kibi-lied?
           (xor
            (and (eqv? kibi-self-desc 'm)
                 (eqv? kibi 'f))
            (and (eqv? kibi-self-desc 'f)
                 (eqv? kibi 'm)))))
      (assert
       (if (not kibi-lied?)
           (xor
            (and (eqv? kibi-self-desc 'm)
                 (eqv? kibi 'm))
            (and (eqv? kibi-self-desc 'f)
```

[6] This puzzle is due to Hunter [Hun76].

```
               (eqv? kibi 'f)))))
        (assert
         (if (eqv? parent1 'm)
             (and
              (eqv? kibi-self-desc 'm)
              (xor
               (and (eqv? kibi 'f)
                    (eqv? kibi-lied? #f))
               (and (eqv? kibi 'm)
                    (eqv? kibi-lied? #t))))))
        (assert
         (if (eqv? parent1 'f)
             (and
              (eqv? kibi 'f)
              (eqv? kibi-lied? #t))))
        (list parent1 parent2 kibi))))
```

A note on the helper procedures: The procedure `distinct?` returns true if all the elements in its argument list are distinct, and false otherwise. The procedure `xor` returns true if only one of its two arguments is true, and false otherwise.

Typing `(solve-kalotan-puzzle)` will solve the puzzle.

14.4.2 Map coloring

It has been known for some time (but only lately proven) that four colors suffice to color a terrestrial map — ie, to color the countries so that neighbors are distinguished. To actually assign the colors is still an undertaking, and the following program shows how nondeterministic programming can help.

The following program solves the problem of coloring a map of Western Europe. The problem and a Prolog solution are given in *The Art of Prolog* [SS94]. (It is instructive to compare our solution with the book's.)

The procedure `choose-color` nondeterministically returns one of four colors:

```
(define choose-color
  (lambda ()
    (amb 'red 'yellow 'blue 'white)))
```

In our solution, we create for each country a data structure. The data structure is a 3-element list: The first element of the list is the country's name; the second element is its assigned color; and the third element is the colors of its neighbors. Note we use the initial of the country for its color variable.[7] Eg, the list for Belgium is `(list 'belgium b (list f h l g))`, because — per the problem statement — the neighbors of Belgium are France, Holland, Luxembourg, and Germany.

Once we create the lists for each country, we state the (single!) condition they should satisfy, viz, no country should have the color of its neighbors. In other words, for every country list, the second element should not be a member of the third element.

```
(define color-europe
  (lambda ()

    ;choose colors for each country
    (let ((p (choose-color)) ;Portugal
          (e (choose-color)) ;Spain
```

[7] Spain (Espana) has `e` so as not to clash with Switzerland.

```
    (f (choose-color)) ;France
    (b (choose-color)) ;Belgium
    (h (choose-color)) ;Holland
    (g (choose-color)) ;Germany
    (l (choose-color)) ;Luxemb
    (i (choose-color)) ;Italy
    (s (choose-color)) ;Switz
    (a (choose-color)) ;Austria
    )

;construct the adjacency list for
;each country: the 1st element is
;the name of the country; the 2nd
;element is its color; the 3rd
;element is the list of its
;neighbors' colors
(let ((portugal
       (list 'portugal p
             (list e)))
      (spain
       (list 'spain e
             (list f p)))
      (france
       (list 'france f
             (list e i s b g l)))
      (belgium
       (list 'belgium b
             (list f h l g)))
      (holland
       (list 'holland h
             (list b g)))
      (germany
       (list 'germany g
             (list f a s h b l)))
      (luxembourg
       (list 'luxembourg l
             (list f b g)))
      (italy
       (list 'italy i
             (list f a s)))
      (switzerland
       (list 'switzerland s
             (list f i a g)))
      (austria
       (list 'austria a
             (list i s g))))
  (let ((countries
         (list portugal spain
               france belgium
               holland germany
               luxembourg
               italy switzerland
               austria)))
```

```
;the color of a country
;should not be the color of
;any of its neighbors
(for-each
 (lambda (c)
   (assert
    (not (memq (cadr c)
               (caddr c)))))
 countries)

;output the color
;assignment
(for-each
 (lambda (c)
   (display (car c))
   (display " ")
   (display (cadr c))
   (newline))
 countries))))))
```

Type `(color-europe)` to get a color assignment.

Chapter 15

Engines

An engine [HF84] represents computation that is subject to timed preemption. In other words, an engine's underlying computation is an ordinary thunk that runs as a timer-preemptable process.

An engine is called with three arguments: (1) a number of time units or *ticks*, (2) a *success* procedure, and (3) a *failure* procedure. If the engine computation finishes within the allotted *ticks*, the *success* procedure is applied to the computation result and the remaining ticks. If the engine computation could not finish within the allotted *ticks*, the *failure* procedure is applied to a new engine representing the unfinished portion of the engine computation.

For example, consider an engine whose underlying computation is a loop that printed the nonnegative integers in sequence. It is created as follows, with the soon-to-be-defined `make-engine` procedure. `make-engine` is called on an argument thunk representing the underlying computation, and it returns the corresponding engine:

```
(define printn-engine
  (make-engine
    (lambda ()
      (let loop ((i 0))
        (display i)
        (display " ")
        (loop (+ i 1))))))
```

Here is a call to `printn-engine`:

```
(define *more* #f)
(printn-engine 50 list (lambda (ne) (set! *more* ne)))
⇒ 0 1 2 3 4 5 6 7 8 9
```

Ie, the loop gets to print upto a certain number (here `9`) and then fails because of the clock interrupt. However, our *failure* procedure sets a global variable called `*more*` to the failed engine, which we can use to resume the loop where the previous engine left off:

```
(*more* 50 list (lambda (ne) (set! *more* ne)))
⇒ 10 11 12 13 14 15 16 17 18 19
```

We will now construct engines using `call/cc` to capture the unfinished computation of a failing engine. First we will construct *flat* engines, or engines whose computation cannot include the running of other engines. We will later generalize the code to the more general *nestable* engines or *nesters*, which can call other engines. But in both cases, we need a timer mechanism, or a *clock*.

15.1 The clock

Our engines assume the presence of a global clock or interruptable timer that marks the passage of ticks as a program executes. We will assume the following clock interface — if your Scheme provides any kind of alarm mechanism, it should be an easy matter to rig up a clock of the following type. (Appendix Cdefines a clock for the Guile [FSF] dialect of Scheme.)

The internal state of our `clock` procedure consists of two items:

(1) the number of remaining ticks; and

(2) an interrupt handler to be invoked when the clock runs out of ticks.

`clock` allows the following operations:

(1) `(clock 'set-handler h)` sets the interrupt handler to `h`.

(2) `(clock 'set n)` resets the clock's remaining ticks to `n`, returning the previous value.

The number of ticks ranges over the non-negative integers and an atom called `*infinity*`. A clock with `*infinity*` ticks cannot run out of time and so will not set off the interrupt handler. Such a clock is *quiescent* or "already stopped". To stop a clock, set its ticks to `*infinity*`.

The clock handler is set to a thunk. For example,

```
(clock 'set-handler
  (lambda ()
    (error "Say goodnight, cat!")))

(clock 'set 9)
```

This will cause an error to be signaled after 9 ticks have passed, and the message displayed by the signal will be "Say goodnight, cat!"

15.2 Flat engines

We will first set the clock interrupt handler. Note that the handler is invoked only if a non-quiescent clock runs out of ticks. This happens only during engine computations that are on the brink of failure, for only engines set the clock.

The handler captures the current continuation, which is the rest of the computation of the currently failing engine. This continuation is sent to another continuation stored in the global `*engine-escape*`. The `*engine-escape*` variable stores the exit continuation of the current engine. Thus the clock handler captures the rest of the failing engine and sends it to an exit point in the engine code, so the requisite failure action can be taken.

```
(define *engine-escape* #f)
(define *engine-entrance* #f)

(clock 'set-handler
  (lambda ()
    (call/cc *engine-escape*)))
```

Let us now look into the innards of the engine code itself. As said, `make-engine` takes a thunk and fashions an engine out of it:

```
(define make-engine
  (lambda (th)
    (lambda (ticks success failure)
      (let* ((ticks-left 0)
             (engine-succeeded? #f)
             (result
              (call/cc
               (lambda (k)
                 (set! *engine-escape* k)
                 (let ((result
```

```
                      (call/cc
                        (lambda (k)
                          (set! *engine-entrance* k)
                          (clock 'set ticks)
                          (let ((v (th)))
                            (*engine-entrance* v))))))
                 (set! ticks-left (clock 'set *infinity*))
                 (set! engine-succeeded? #t)
                 result)))))
      (if engine-succeeded?
          (success result ticks-left)
          (failure
           (make-engine
            (lambda ()
              (result 'resume)))))))))
```

First we introduce the variables `ticks-left` and `engine-succeeded?`. The first will hold the ticks left over should the engine thunk finish in time. The second is a flag that will be used in the engine code to signal if the engine suceeded.

We then run the engine thunk within two nested calls to `call/cc`. The first `call/cc` captures the continuation to be used by a failing engine to abort out of its engine computation. This continuation is stored in the global `*engine-escape*`. The second `call/cc` captures an inner continuation that will be used by the return value of the thunk `th` if it runs to completion. This continuation is stored in the global `*engine-entrance*`.

Running through the code, we find that after capturing the continuations `*engine-escape*` and `*engine-entrance*`, we set the clock's ticks to the time allotted this engine and run the thunk `th`. If `th` succeeds, its value `v` is sent to the continuation `*engine-entrance*`, after which the clock is stopped, the remaining ticks ascertained, and the flag `engine-succeeded?` is set to true. We now go past the `*engine-escape*` continuation, and run the final dispatcher in the code: Since we know the engine succeeded, we apply the `success` procedure to the result and the ticks left.

If the thunk `th` *didn't* finish in time though, it will suffer an interrupt. This invokes the clock interrupt handler, which captures the current continuation of the running and now failing thunk and sends it to the continuation `*engine-escape*`. This puts the failed-thunk continuation in the outer `result` variable, and we are now in the final dispatcher in the code: Since `engine-succeeded?` is still false, we apply the `failure` procedure to new engine fashioned out of `result`.

Notice that when a failed engine is removed, it will traverse the control path charted by the first run of the original engine. Nevertheless, because we have explicitly use the continuations stored in the global variables `*engine-entrance*` and `*engine-escape*`, and we always set them anew before executing an engine computation, we are assured that the jumps will always come back to the currently executing engine code.

15.3 Nestable engines

In order to generalize the code above to accommodate the nestable type of engine, we need to incorporate into it some *tick management* that will take care of the apportioning of the right amounts of ticks all the engines in a nested run.

To run a new engine (the *child*), we need to stop the currently engine (the *parent*). We then need to assign an appropriate number of ticks to the child. This may not be the same as the ticks assigned by the program text, because it would be *unfair* for a child to consume more ticks than its parent has left. After the child completes, we need to update the parent's ticks. If the child finished in time, any leftover ticks it has revert to the parent. If ticks were denied from the child because the parent couldn't afford it, then if the child fails, the parent will fail too, but must remember to restart the child with its promised ticks when it (the parent) restarts.

We also need to `fluid-let` the globals `*engine-escape*` and `*engine-entrance*`, because each nested engine must have its own pair of these sentinel continuations. As an engine exits (whether through success or failure), the `fluid-let` will ensure that the next enclosing engine's sentinels take over.

Combining all this, the code for nestable engines looks as follows:

```
(define make-engine
  (lambda (th)
    (lambda (ticks s f)
      (let* ((parent-ticks
               (clock 'set *infinity*))

              ;A child can't have more ticks than its parent's
              ;remaining ticks
              (child-available-ticks
               (clock-min parent-ticks ticks))

              ;A child's ticks must be counted against the parent
              ;too
              (parent-ticks-left
               (clock-minus parent-ticks child-available-ticks))

              ;If child was promised more ticks than parent could
              ;afford, remember how much it was short-changed by
              (child-ticks-left
               (clock-minus ticks child-available-ticks))

              ;Used below to store ticks left in clock
              ;if child completes in time
              (ticks-left 0)

              (engine-succeeded? #f)

              (result
               (fluid-let ((*engine-escape* #f)
                           (*engine-entrance* #f))
                 (call/cc
                  (lambda (k)
                    (set! *engine-escape* k)
                    (let ((result
                           (call/cc
                            (lambda (k)
                              (set! *engine-entrance* k)
                              (clock 'set child-available-ticks)

                              (let ((v (th)))

                                (*engine-entrance* v))))))
                      (set! ticks-left
```

```
                (let ((n (clock 'set *infinity*)))
                  (if (eqv? n *infinity*) 0 n)))
              (set! engine-succeeded? #t)
              result))))))

;Parent can reclaim ticks that child didn't need
(set! parent-ticks-left
  (clock-plus parent-ticks-left ticks-left))

;This is the true ticks that child has left --
;we include the ticks it was short-changed by
(set! ticks-left
  (clock-plus child-ticks-left ticks-left))

;Restart parent with its remaining ticks
(clock 'set parent-ticks-left)
;The rest is now parent computation

(cond
 ;Child finished in time -- celebrate its success
 (engine-succeeded? (s result ticks-left))

 ;Child failed because it ran out of promised time --
 ;call failure procedure
 ((= ticks-left 0)
  (f (make-engine (lambda () (result 'resume)))))

 ;Child failed because parent didn't have enough time,
 ;ie, parent failed too.  If so, when parent is
 ;resumed, its first order of duty is to resume the
 ;child with its fair amount of ticks
 (else
  ((make-engine (lambda () (result 'resume)))
   ticks-left s f)))))))
```

Note that we have used the arithmetic operators `clock-min`, `clock-minus`, and `clock-plus` instead of `min`, `-`, and `+`. This is because the values used by the clock arithmetic includes `*infinity*` in addition to the integers. Some Scheme dialects provide an `*infinity*` value in their arithmetic[8] — if so, you can use the regular arithmetic operators. If not, it is an easy exercise to define the enhanced operators.

[8] Eg, in Guile, you can `(define *infinity* (/ 1 0))`.

Chapter 16

Shell scripts

It is often convenient to simply write what one wants done into a file or *script*, and execute the script as though it were any other operating-system shell command. The interface to more weighty programs is often provided in the form of a script, and users frequently build their own scripts or customize existing ones to suit particular needs. Scripting is arguably the most frequent programming task performed. For many users, it is the only programming they will ever do.

Operating systems such as Unix and DOS (the command-line interface provided in Windows) provide such a scripting mechanism, but the scripting language in both cases is very rudimentary. Often a script is just a sequence or *batch* of commands that one would type to the shell prompt. It saves the user from having to type every one of the shell commands individually each time they require the same or similar sequence to be performed. Some scripting languages throw in a small amount of programmability in the form of a conditional and a loop, but that is about all. This is enough for smallish tasks, but as one's scripts become bigger and more demanding, as scripts invariably seem to do, one often feels the need for a fuller fledged programming language. A Scheme with an adequate operating-system interface makes scripting easy and maintainable.

This section will describe how to write scripts in Scheme. Since there is wide variation in the various Scheme dialects on how to accomplish this, we will concentrate on the MzScheme dialect, and document in appendix Athe modifications needed for other dialects. We will also concentrate on the Unix operating system for the moment; appendix Bwill deal with the DOS counterpart.

16.1 Hello, World!, again

We will now create a Scheme script that says hello to the world. Saying hello is of course not a demanding scripting problem for traditional scripting languages. However, understanding how to transcribe it into Scheme will launch us on the path to more ambitious scripts. First, a conventional Unix hello script is a file, with contents that look like:

```
echo Hello, World!
```

It uses the shell command `echo`. The script can be named `hello`, made into an executable by doing

```
chmod +x hello
```

and placed in one of the directories named in the `PATH` environment variable. Thereafter, anytime one types

```
hello
```

at the shell prompt, one promptly gets the insufferable greeting.

A Scheme hello script will perform the same output using Scheme (using the program in sec 1), but we need something in the file to inform the operating system that it needs to construe the commands in the file as Scheme, and not as its default script language. The Scheme script file, also called `hello`, looks like:

```
":"; exec mzscheme -r $0 "$@"

(display "Hello, World!")
(newline))
```

Everything following the first line is straight Scheme. However, the first line is the magic that makes this into a script. When the user types `hello` at the Unix prompt, Unix will read the file as a regular script. The first thing it sees is the `":"`, which is a shell no-op. The `;` is the shell command separator. The next shell command is the `exec`. `exec` tells Unix to abandon the current script and run `mzscheme -r $0 "$@"` instead, where the parameter `$0` will be replaced by the name of the script, and the parameter `"$@"` will be replaced by the list of arguments given by the user to the script. (In this case, there are no such arguments.)

We have now, in effect, transformed the `hello` shell command into a different shell command, viz,

```
mzscheme -r /whereveritis/hello
```

where `/whereveritis/hello` is the pathname of `hello`.

`mzscheme` calls the MzScheme executable. The `-r` option tells it to load the immediately following argument as a Scheme file after collecting any succeeding arguments into a vector called `argv`. (In this example, `argv` will be the null vector.)

Thus, the Scheme script will be run as a Scheme file, and the Scheme forms in the file will have access to the script's original arguments via the vector `argv`.

Now, Scheme has to tackle the first line in the script, which as we've already seen, was really a well-formed, *traditional* shell script. The `":"` is a self-evaluating string in Scheme and thus harmless. The '`;`' marks a Scheme comment, and so the `exec ...` is safely ignored. The rest of the file is of course straight Scheme, and the expressions therein are evaluated in sequence. After all of them have been evaluated, Scheme will exit.

In sum, typing `hello` at the shell prompt will produce

```
Hello, World!
```

and return you to the shell prompt.

16.2 Scripts with arguments

A Scheme script uses the variable `argv` to refer to its arguments. For example, the following script echoes all its arguments, each on a line:

```
":"; exec mzscheme -r $0 "$@"

;Put in argv-count the number of arguments supplied

(define argv-count (vector-length argv))

(let loop ((i 0))
  (unless (>= i argv-count)
    (display (vector-ref argv i))
    (newline)
    (loop (+ i 1))))
```

Let's call this script `echoall`. Calling `echoall 1 2 3` will display

```
1
2
3
```

Note that the script name ("echoall") is *not* included in the argument vector.

16.3 Example

Let's now tackle a more substantial problem. We need to transfer files from one computer to another and the only method we have is to use a 3.5" floppy as a ferry. We need a script split4floppy that will split files larger than 1.44 million bytes into floppy-sized chunks. The script file split4floppy is as follows:

```
":";exec mzscheme -r $0 "$@"

;floppy-size = number of bytes that will comfortably fit on a
;              3.5" floppy

(define floppy-size 1440000)

;split splits the bigfile f into the smaller, floppy-sized
;subfiles, viz, subfile-prefix.1, subfile-prefix.2, etc.

(define split
  (lambda (f subfile-prefix)
    (call-with-input-file f
      (lambda (i)
        (let loop ((n 1))
          (if (copy-floppy-size-chunk i subfile-prefix n)
              (loop (+ n 1))))))))

;copy-to-floppy-sized-subfile copies the next 1.44 million
;bytes (if there are less than that many bytes left, it
;copies all of them) from the big file to the nth
;subfile.  Returns true if there are bytes left over,
;otherwise returns false.

(define copy-to-floppy-sized-subfile
  (lambda (i subfile-prefix n)
    (let ((nth-subfile (string-append subfile-prefix "."
                                      (number->string n))))
      (if (file-exists? nth-subfile) (delete-file nth-subfile))
      (call-with-output-file nth-subfile
        (lambda (o)
          (let loop ((k 1))
            (let ((c (read-char i)))
              (cond ((eof-object? c) #f)
                    (else
                     (write-char c o)
                     (if (< k floppy-size)
                         (loop (+ k 1))
                         #t))))))))))

;bigfile = script's first arg
;        = the file that needs splitting

(define bigfile (vector-ref argv 0))
```

```
;subfile-prefix = script's second arg
;               = the basename of the subfiles

(define subfile-prefix (vector-ref argv 1))

;Call split, making subfile-prefix.{1,2,3,...} from
;bigfile

(split bigfile subfile-prefix)
```

Script `split4floppy` is called as follows:

```
split4floppy largefile chunk
```

This splits `largefile` into subfiles `chunk.1`, `chunk.2`, ..., such that each subfile fits on a floppy.

After the `chunk.i` have been ferried over to the target computer, the file `largefile` can be retrieved by stringing the `chunk.i` together. This can be done on Unix with:

```
cat chunk.1 chunk.2 ... > largefile
```

and on DOS with:

```
copy /b chunk.1+chunk.2+... largefile
```

Chapter 17

CGI scripts

WARNING: CGI scripts without appropriate safeguards can compromise your site's security. The scripts presented here are simple examples and are not assured to be secure for actual Web use.

CGI scripts [NCS] are scripts that reside on a web server and can be run by a client (browser). The client accesses a CGI script by its URL, just as they would a regular page. The server, recognizing that the URL requested is a CGI script, runs it. How the server recognizes certain URLs as scripts is up to the server administrator. For the purposes of this text, we will assume that they are stored in a distinguished directory called `cgi-bin`. Thus, the script `testcgi.scm` on the server `www.foo.org` would be accessed as `http://www.foo.org/cgi-bin/testcgi.scm`.

The server runs the CGI script as the user `nobody`, who cannot be expected to have any `PATH` knowledge (which is highly subjective anyway). Therefore the introductory magic line for a CGI script written in Scheme needs to be a bit more explicit than the one we used for ordinary Scheme scripts. Eg, the line

```
":";exec mzscheme -r $0 "$@"
```

implicitly assumes that there is a particular shell (`bash`, say), and that there is a `PATH`, and that `mzscheme` is in it. For CGI scripts, we will need to be more expansive:

```
#!/bin/sh
":";exec /usr/local/bin/mzscheme -r $0 "$@"
```

This gives fully qualified pathnames for the shell and the Scheme executable. The transfer of control from shell to Scheme proceeds as for regular scripts.

17.1 Example: Displaying environment variables

Here is an example Scheme CGI script, `testcgi.scm`, that outputs the settings of some commonly used CGI environment variables. This information is returned as a new, freshly created, page to the browser. The returned page is simply whatever the CGI script writes to its standard output. This is how CGI scripts talk back to whoever called them — by giving them a new page.

Note that the script first outputs the line

```
content-type: text/plain
```

followed by a blank line. This is standard ritual for a web server serving up a page. These two lines aren't part of what is actually displayed as the page. They are there to inform the browser that the page being sent is plain (ie, un-marked-up) text, so the browser can display it appropriately. If we were producing text marked up in HTML, the `content-type` would be `text/html`.

The script `testcgi.scm`:

```
#!/bin/sh
":";exec /usr/local/bin/mzscheme -r $0 "$@"
```

```
;Identify content-type as plain text.

(display "content-type: text/plain") (newline)
(newline)

;Generate a page with the requested info.  This is
;done by simply writing to standard output.

(for-each
 (lambda (env-var)
   (display env-var)
   (display " = ")
   (display (or (getenv env-var) ""))
   (newline))
 '("AUTH_TYPE"
   "CONTENT_LENGTH"
   "CONTENT_TYPE"
   "DOCUMENT_ROOT"
   "GATEWAY_INTERFACE"
   "HTTP_ACCEPT"
   "HTTP_REFERER" ; [sic]
   "HTTP_USER_AGENT"
   "PATH_INFO"
   "PATH_TRANSLATED"
   "QUERY_STRING"
   "REMOTE_ADDR"
   "REMOTE_HOST"
   "REMOTE_IDENT"
   "REMOTE_USER"
   "REQUEST_METHOD"
   "SCRIPT_NAME"
   "SERVER_NAME"
   "SERVER_PORT"
   "SERVER_PROTOCOL"
   "SERVER_SOFTWARE"))
```

`testcgi.scm` can be called directly by opening it on a browser. The URL is:

```
http://www.foo.org/cgi-bin/testcgi.scm
```

Alternately, `testcgi.scm` can occur as a link in an HTML file, which you can click. Eg,

```
... To view some common CGI environment variables, click
<a href="http://www.foo.org/cgi-bin/testcgi.scm">here</a>.
...
```

However `testcgi.scm` is launched, it will produce a plain text page containing the settings of the environment variables. An example output:

```
AUTH_TYPE =
CONTENT_LENGTH =
CONTENT_TYPE =
DOCUMENT_ROOT = /home/httpd/html
GATEWAY_INTERFACE = CGI/1.1
HTTP_ACCEPT = image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
*/*
```

```
HTTP_REFERER =
HTTP_USER_AGENT = Mozilla/3.01Gold (X11; I; Linux 2.0.32 i586)
PATH_INFO =
PATH_TRANSLATED =
QUERY_STRING =
REMOTE_HOST = 127.0.0.1
REMOTE_ADDR = 127.0.0.1
REMOTE_IDENT =
REMOTE_USER =
REQUEST_METHOD = GET
SCRIPT_NAME = /cgi-bin/testcgi.scm
SERVER_NAME = localhost.localdomain
SERVER_PORT = 80
SERVER_PROTOCOL = HTTP/1.0
SERVER_SOFTWARE = Apache/1.2.4
```

17.2 Example: Displaying selected environment variable

testcgi.scm does not take any input from the user. A more focused script would take an argument environment variable from the user, and output the setting of that variable and none else. For this, we need a mechanism for feeding arguments to CGI scripts. The form tag of HTML provides this capability. Here is a sample HTML page for this purpose:

```
<html>
<head>
<title>Form for checking environment variables</title>
</head>
<body>

<form method=get
      action="http://www.foo.org/cgi-bin/testcgi2.scm">
Enter environment variable: <input type=text name=envvar size=30>
<p>

<input type=submit>
</form>

</body>
</html>
```

The user enters the desired environment variable (eg, GATEWAY_INTERFACE) in the textbox and clicks the submit button. This causes all the information in the form — here, the setting of the parameter envvar to the value GATEWAY_INTERFACE — to be collected and sent to the CGI script identified by the form, viz, testcgi2.scm. The information can be sent in one of two ways: (1) if the form's method=get (the default), the information is sent via the environment variable called QUERY_STRING; (2) if the form's method=post, the information is available to the CGI script at the latter's standard input port (stdin). Our form uses QUERY_STRING.

It is testcgi2.scm's responsibility to extract the information from QUERY_STRING, and output the answer page accordingly.

The information to the CGI script, whether arriving via an environment variable or through stdin, is formatted as a sequence of parameter/argument pairs.

The pairs are separated from each other by the & character. Within a pair, the parameter occurs first and is separated from the argument by the = character. In this case, there is only one parameter/argument pair, viz, envvar=GATEWAY_INTERFACE.

The script testcgi2.scm:

```
#!/bin/sh
":";exec /usr/local/bin/mzscheme -r $0 "$@"

(display "content-type: text/plain") (newline)
(newline)

;string-index returns the leftmost index in string s
;that has character c

(define string-index
  (lambda (s c)
    (let ((n (string-length s)))
      (let loop ((i 0))
        (cond ((>= i n) #f)
              ((char=? (string-ref s i) c) i)
              (else (loop (+ i 1))))))))

;split breaks string s into substrings separated by character c

(define split
  (lambda (c s)
    (let loop ((s s))
      (if (string=? s "") '()
          (let ((i (string-index s c)))
            (if i (cons (substring s 0 i)
                        (loop (substring s (+ i 1)
                                         (string-length s))))
                (list s)))))))

(define args
  (map (lambda (par-arg)
         (split #\= par-arg))
       (split #\& (getenv "QUERY_STRING"))))

(define envvar (cadr (assoc "envvar" args)))

(display envvar)
(display " = ")
(display (getenv envvar))

(newline)
```

Note the use of a helper procedure split to split the QUERY_STRING into parameter/argument pairs along the & character, and then splitting parameter and argument along the = character. (If we had used the post method rather than get, we would have needed to extract the parameters and arguments from the standard input.)

The <input type=text> and <input type=submit> are but two of the many different input tags possible in an HTML form. Consult [NCS] for the full repertoire.

17.3 CGI script utilities

In the example above, the parameter's name or the argument it assumed did not themselves contain any '&' or '=' characters. In general, they may. To accommodate such characters, and not have them be mistaken for separators, the CGI argument-passing mechanism treats all characters other than letters, digits, and the underscore, as *special*, and transmits them in an encoded form. A space is encoded as a '+'. For other special characters, the encoding is a three-character sequence, and consists of '%' followed the special character's hexadecimal code. Thus, the character sequence '`20% + 30% = 50%, &c.`' will be encoded as

```
20%25+%2b+30%25+%3d+50%25%2c+%26c%2e
```

(Space become '+'; '%' becomes '`%25`'; '+' becomes '`%2b`'; '=' becomes '`%3d`'; '`,`' becomes '`%2c`'; '&' becomes '`%26`'; and '`.`' becomes '`%2e`'.)

Instead of dealing anew with the task of getting and decoding the form data in each CGI script, it is convenient to collect some helpful procedures into a library file `cgi.scm`. `testcgi2.scm` can then be written more compactly as

```
#!/bin/sh
":";exec /usr/local/bin/mzscheme -r $0 "$@"

;Load the cgi utilities

(load-relatve "cgi.scm")

(display "content-type: text/plain") (newline)
(newline)

;Read the data input via the form

(parse-form-data)

;Get the envvar parameter

(define envvar (form-data-get/1 "envvar"))

;Display the value of the envvar

(display envvar)
(display " = ")
(display (getenv envvar))
(newline)
```

This shorter CGI script uses two utility procedures defined in `cgi.scm`. `parse-form-data` to read the data supplied by the user via the form. The data consists of parameters and their associated values. `form-data-get/1` finds the value associated with a particular parameter.

`cgi.scm` defines a global table called `*form-data-table*` to store form data.

```
;Load our table definitions

(load-relative "table.scm")

;Define the *form-data-table*

(define *form-data-table* (make-table 'equ string=?))
```

An advantage of using a general mechanism such as the `parse-form-data` procedure is that we can hide the details of what `method` (`get` or `put`) was used.

```
(define parse-form-data
  (lambda ()
    ((if (string-ci=? (or (getenv "REQUEST_METHOD") "GET") "GET")
         parse-form-data-using-query-string
         parse-form-data-using-stdin))))
```

The environment variable `REQUEST_METHOD` tells which method was used to transmit the form data. If the method is `GET`, then the form data was sent as the string available via another environment variable, `QUERY_STRING`. The auxiliary procedure `parse-form-data-using-query-string` is used to pick apart `QUERY_STRING`:

```
(define parse-form-data-using-query-string
  (lambda ()
    (let ((query-string (or (getenv "QUERY_STRING") "")))
      (for-each
       (lambda (par=arg)
         (let ((par/arg (split #\= par=arg)))
           (let ((par (url-decode (car par/arg)))
                 (arg (url-decode (cadr par/arg))))
             (table-put!
              *form-data-table* par
              (cons arg
                    (table-get *form-data-table* par '()))))))
       (split #\& query-string)))))
```

The helper procedure `split`, and *its* helper `string-index`, are defined as in sec 17.2. As noted, the incoming form data is a sequence of name-value pairs separated by `&`s. Within each pair, the name comes first, followed by an `=` character, followed by the value. Each name-value combination is collected into a global table, the `*form-data-table*`.
Both name and value are encoded, so we need to decode them using the `url-decode` procedure to get their actual representation.

```
(define url-decode
  (lambda (s)
    (let ((s (string->list s)))
      (list->string
       (let loop ((s s))
         (if (null? s) '()
             (let ((a (car s)) (d (cdr s)))
               (case a
                 ((#\+) (cons #\space (loop d)))
                 ((#\%) (cons (hex->char (car d) (cadr d))
                              (loop (cddr d))))
                 (else (cons a (loop d)))))))))))
```

'`+`' is converted into space. A triliteral of the form '`%xy`' is converted, using the procedure `hex->char` into the character whose ascii encoding is the hex number '`xy`'.

```
(define hex->char
  (lambda (x y)
    (integer->char
     (string->number (string x y) 16))))
```

We still need a form-data parser for the case where the request method is `POST`. The auxiliary procedure `parse-form-data-using-stdin` does this.

```
(define parse-form-data-using-stdin
  (lambda ()
    (let* ((content-length (getenv "CONTENT_LENGTH"))
           (content-length (if content-length
                               (string->number content-length) 0))
           (i 0))
      (let par-loop ((par '()))
        (let ((c (read-char)))
          (set! i (+ i 1))
          (if (or (> i content-length) (eof-object? c) (char=? c #\=))
              (let arg-loop ((arg '()))
                (let ((c (read-char)))
                  (set! i (+ i 1))
                  (if (or (> i content-length) (eof-object? c) (char=?
c #\&))
                      (let ((par (url-decode (list->string (reverse!
par))))
                            (arg (url-decode (list->string (reverse!
arg)))))
                        (table-put! *form-data-table* par
                                    (cons arg (table-get *form-data-
table*
                                                         par '()))))
                        (unless (or (> i content-length)
                                    (eof-object? c))
                          (par-loop '())))
                      (arg-loop (cons c arg)))))
              (par-loop (cons c par))))))))
```

The `POST` method sends form data via the script's `stdin`. The number of characters sent is placed in the environment variable `CONTENT_LENGTH`. `parse-form-data-using-stdin` reads the required number of characters from `stdin`, and populates the `*form-data-table*` as before, making sure to decode the parameters' names and values.

It remains to retrieve the values for specific parameters from the `*form-data-table*`. Note that the table associates a list with each parameter, in order to accommodate the possibility of multiple values for a parameter. `form-data-get` retrieves all the values assigned to a parameter. If there is only one value, it returns a singleton containing that value.

```
(define form-data-get
  (lambda (k)
    (table-get *form-data-table* k '())))
```

`form-data-get/1` returns the first (or most significant) value associated with a parameter.

```
(define form-data-get/1
  (lambda (k . default)
    (let ((vv (form-data-get k)))
      (cond ((pair? vv) (car vv))
            ((pair? default) (car default))
            (else "")))))
```

In our examples so far, the CGI script has generated plain text. Generally, though, we will want to generate an HTML page. It is not uncommon for a combination of HTML form and CGI script to trigger a series of HTML pages with forms. It is also common to code all the action corresponding to these various forms in a single CGI script. In any case, it is helpful to have a utility procedure that writes out strings in HTML format, ie, with the HTML special characters encoded appropriately:

```
(define display-html
  (lambda (s . o)
    (let ((o (if (null? o) (current-output-port)
                 (car o))))
      (let ((n (string-length s)))
        (let loop ((i 0))
          (unless (>= i n)
            (let ((c (string-ref s i)))
              (display
               (case c
                 ((#\<) "&lt;")
                 ((#\>) "&gt;")
                 ((#\") "&quot;")
                 ((#\&) "&amp;")
                 (else c)) o)
              (loop (+ i 1)))))))))
```

17.4 A calculator via CGI

Here is an CGI calculator script, `cgicalc.scm`, that exploits Scheme's arbitrary-precision arithmetic.

```
#!/bin/sh
":";exec /usr/local/bin/mzscheme -r $0

;Load the CGI utilities
(load-relative "cgi.scm")

(define uhoh #f)

(define calc-eval
  (lambda (e)
    (if (pair? e)
        (apply (ensure-operator (car e))
               (map calc-eval (cdr e)))
        (ensure-number e))))

(define ensure-operator
  (lambda (e)
    (case e
      ((+) +)
      ((-) -)
      ((*) *)
      ((/) /)
      ((**) expt)
      (else (uhoh "unpermitted operator")))))
```

```
(define ensure-number
  (lambda (e)
    (if (number? e) e
        (uhoh "non-number"))))

(define print-form
  (lambda ()
    (display "<form action=\"")
    (display (getenv "SCRIPT_NAME"))
    (display "\">
  Enter arithmetic expression:<br>
  <input type=textarea name=arithexp><p>
  <input type=submit value=\"Evaluate\">
  <input type=reset value=\"Clear\">
</form>")))

(define print-page-begin
  (lambda ()
    (display "content-type: text/html

<html>
  <head>
    <title>A Scheme Calculator</title>
  </head>
  <body>")))

(define print-page-end
  (lambda ()
    (display "</body>
</html>")))

(parse-form-data)

(print-page-begin)

(let ((e (form-data-get "arithexp")))
  (unless (null? e)
    (let ((e1 (car e)))
      (display-html e1)
      (display "<p>
  =&gt;  ")
      (display-html
       (call/cc
        (lambda (k)
          (set! uhoh
                (lambda (s)
                  (k (string-append "Error: " s))))
          (number->string
           (calc-eval (read (open-input-string (car e))))))))
      (display "<p>"))))

(print-form)
(print-page-end)
```

Appendix A

Scheme dialects

All major Scheme dialects implement the R5RS specification [KCR98]. By using only the features documented in the R5RS, one can write Scheme code that is portable across the dialects. However, the R5RS, either for want of consensus or because of inevitable system dependencies, remains silent on several matters that non-trivial programming cannot ignore. The various dialects have therefore had to solve these matters in a non-standard and idiosyncratic manner.

This book uses the MzScheme [Fla] dialect of Scheme, and thereby uses several features that are nonstandard. The complete list of the dialect-dependent features used in this book is: the command-line (both for opening a listener session and for shell scripts), `define-macro`, `delete-file`, `file-exists?`, `file-or-directory-modify-seconds`, `fluid-let`, `gensym`, `getenv`, `get-output-string`, `load-relative`, `open-input-string`, `open-output-string`, `read-line`, `reverse!`, `system`, `when`, and `unless`.

Some of these features (eg, `file-exists?`, `delete-file`) are de facto standards and are present in many Schemes. Some other features (eg, `when`, `unless`) have more or less "plug-in" definitions (given in this book) that can be loaded into any Scheme dialect that doesn't have them primitively. The rest require a dialect-specific definition (eg, `load-relative`).

This chapter describes how to incorporate into your Scheme dialect the nonstandard features used in this book. For further detail about your Scheme dialect, consult the documentation provided by its implementor (sec D).

A.1 Invocation

To invoke the Guile [FSF] listener, type `guile`. To invoke SCM [Jaf], type `scm`.

To invoke STk [Gal], type `snow`. This is the "no window" executable, which is enough for the purposes of this book.

A.2 Init files

Many Scheme dialects load, if available, an *initialization file* (or *init file*), usually supplied in the user's home directory. The init file is a convenient location in which to place definitions for nonstandard features. Eg, the nonstandard procedure `file-or-directory-modify-seconds` can be added to the Guile dialect of Scheme by putting the following code in Guile's init file, which is `~/.guile`:

```
(define file-or-directory-modify-seconds
  (lambda (f)
    (vector-ref (stat f) 9)))
```

The init files for SCM and STk are `~/ScmInit.scm` and `~/.stkrc` respectively.

MzScheme also has an init file, viz, `~/.mzschemerc`. Note, however, that for the purposes of this text, you don't need to insert code into `~/.mzschemerc`, since we don't use any nonstandard Scheme feature not already defined in MzScheme.

A.3 Shell scripts

The initial line for a shell script written in Guile is:

```
":";exec guile -s $0 "$@"
```

In the script, the procedure-call `(command-line)` returns the list of the script's name and arguments. To access just the arguments, take the `cdr` of this list.

A shell script written in SCM starts out as:

```
":";exec scm -l $0 "$@"
```

In the script, the variable `*argv*` contains the list of the Scheme executable name, the script's name, the option `-l`, and the script's arguments. To access just the arguments, take the `cdddr` of this list.

STk shell scripts start out as:

```
":";exec snow -f $0 "$@"
```

In the script, the variable `*argv*` contains the list of the script's arguments.

A.4 define-macro

The `define-macro` used in the text occurs in the Scheme dialects Bigloo [Ser], Gambit [Fee], Guile, MzScheme and Pocket Scheme [Goe]. There are minor variations in how macros are defined in the other Scheme dialects. The rest of this section will point out how these other dialects notate the following code fragment:

```
(define-macro MACRO-NAME
  (lambda MACRO-ARGS
    MACRO-BODY ...))
```

In MIT Scheme [MIT], this is written as:

```
(syntax-table-define system-global-syntax-table 'MACRO-NAME
  (macro MACRO-ARGS
    MACRO-BODY ...))
```

In SCM and Kawa [Bot]:

```
(defmacro MACRO-NAME MACRO-ARGS
  MACRO-BODY ...)
```

In STk:

```
(define-macro (MACRO-NAME . MACRO-ARGS)
  MACRO-BODY ...)
```

A.5 load-relative

The procedure `load-relative` may be defined for Guile as follows:

```
(define load-relative
  (lambda (f)
    (let* ((n (string-length f))
           (full-pathname?
            (and (> n 0)
                 (let ((c0 (string-ref f 0)))
                   (or (char=? c0 #\/)
                       (char=? c0 #\~))))))
      (basic-load
```

```
       (if full-pathname? f
           (let ((clp (current-load-port)))
             (if clp
                 (string-append (dirname (port-filename clp)) "/" f)
                 f)))))))
```

For SCM:

```
(define load-relative
  (lambda (f)
    (let* ((n (string-length f))
           (full-pathname?
            (and (> n 0)
                 (let ((c0 (string-ref f 0)))
                   (or (char=? c0 #\/)
                       (char=? c0 #\~))))))
    (load (if (and *load-pathname* full-pathname?)
              (in-vicinity (program-vicinity) f)
              f)))))
```

For STk, the following definition for `load-relative` works only if you discipline yourself to not use `load`:

```
(define *load-pathname* #f)

(define stk%load load)

(define load-relative
  (lambda (f)
    (fluid-let ((*load-pathname*
                 (if (not *load-pathname*) f
                     (let* ((n (string-length f))
                            (full-pathname?
                             (and (> n 0)
                                  (let ((c0 (string-ref f 0)))
                                    (or (char=? c0 #\/)
                                        (char=? c0 #\~))))))
                       (if full-pathname? f
                           (string-append (dirname *load-pathname*)
                                          "/" f))))))
      (stk%load *load-pathname*))))

(define load
  (lambda (f)
    (error "Don't use load.  Use load-relative instead.")))
```

Appendix B

DOS batch files in Scheme

DOS shell scripts are known as *batch files*. A conventional DOS batch file that outputs "Hello, World!" has the following contents:

```
echo Hello, World!
```

It uses the DOS command `echo`. The batch file is named `hello.bat`, which identifies it to the operating system as an executable. It may then be placed in one of the directories on the `PATH` environment variable. Thereafter, anytime one types

```
hello.bat
```

or simply

```
hello
```

at the DOS prompt, one promptly gets the insufferable greeting.

A Scheme version of the hello batch file will perform the same output using Scheme, but we need something in the file to inform DOS that it needs to construe the commands in the file as Scheme, and not as its default batch language. The Scheme batch file, also called `hello.bat`, looks like:

```
;@echo off
;goto :start
#|
:start
echo. > c:\_temp.scm
echo (load (find-executable-path "hello.bat" >> c:\_temp.scm
echo "hello.bat")) >> c:\_temp.scm
mzscheme -r c:\_temp.scm %1 %2 %3 %4 %5 %6 %7 %8 %9
goto :eof
|#

(display "Hello, World!")
(newline))

;:eof
```

The lines upto `|#` are standard DOS batch. Then follows the Scheme code for the greeting. Finally, there is one more standard DOS batch line, viz, `;:eof`.

When the user types `hello` at the DOS prompt, DOS reads and runs the file `hello.bat` as a regular batch file. The first line, `;@echo off`, turns off the echoing of the commands run — as we don't want excessive verbiage clouding the effect of our script. The second line, `;goto :start`, causes execution to jump forward to the line labeled `:start`, ie, the fourth line. The three ensuing `echo` lines create a temporary Scheme file called `c:\_temp.tmp` with the following contents:

```
(load (find-executable-path "hello.bat" "hello.bat"))
```

The next batch command is a call to MzScheme. The `-r` option loads the Scheme file `c:\_temp.scm`. All the arguments (in this example, none) will be available to Scheme in the vector `argv`. This call to Scheme will evaluate our Scheme script, as we will see below. After Scheme returns, we still need to ensure that the batch file winds up cleanly. The next batch command is `goto :eof`, which causes control to skirt all the Scheme code and go to the very end of the file, which contains the label `;:eof`. The script thus ends.

Now we can see how the call to Scheme does its part, viz, to run the Scheme expressions embedded in the batch file. Loading `c:\_temp.scm` will cause Scheme to deduce the full pathname of the file `hello.bat` (using `find-executable-path`), and to then *load* `hello.bat`.

Thus, the Scheme script file will now be run as a Scheme file, and the Scheme forms in the file will have access to the script's original arguments via the vector `argv`.

Now, Scheme has to skirt the batch commands in the script. This is easily done because these batch commands are either prefixed with a semicolon or are enclosed in `#| ... |#`, making them Scheme comments.

The rest of the file is of course straight Scheme, and the expressions therein are evaluated in sequence. (The final expression, `;:eof`, is a Scheme comment, and causes no harm.) After all the expressions have been evaluated, Scheme will exit.

In sum, typing `hello` at the DOS prompt will produce

```
Hello, World!
```

and return you to the DOS prompt.

Appendix C

A clock for infinity

The Guile [FSF] procedure `alarm` provides an interruptable timer mechanism. The user can set or reset the alarm for some time units, or stop it. When the alarm's timer runs out of this time, it will set off an alarm, whose consequences are user-settable. Guile's `alarm` is not quite the clock of sec 15.1, but we can modify it easily enough.

The alarm's timer is initially *stopped* or *quiescent*, ie, it will not set off an alarm even as time goes by. To set the alarm's time-to-alarm to be `n` seconds, where `n` is not `0`, run `(alarm n)`. If the timer was already set (but has not yet set off an alarm), the `(alarm n)` procedure call will return the number of seconds remaining from the previous alarm setting. If there is no previous alarm setting, `(alarm n)` returns `0`.

The procedure call `(alarm 0)` *stops* the alarm's timer, ie, the countdown of time is stopped, the timer becomes quiescent and no alarm will go off. `(alarm 0)` also returns the seconds remaining from a previous alarm setting, if any.

By default, when the alarm's countdown reaches 0, Guile will display a message on the console and exit. More useful behavior can be obtained by using the procedure `sigaction`, as follows:

```
(sigaction SIGALRM
  (lambda (sig)
    (display "Signal ")
    (display sig)
    (display " raised.  Continuing...")
    (newline)))
```

The first argument `SIGALRM` (which happens to be `14`) identifies to `sigaction` that it is the alarm handler that needs setting.[9] The second argument is a unary alarm-handling procedure of the user's choice. In this example, when the alarm goes off, the handler displays `"Signal 14 raised. Continuing..."` on the console without exiting Scheme. (The `14` is the `SIGALRM` value that the alarm will pass to its handler. Don't worry about it now.)

From our point of view, this simple timer mechanism poses one problem. A return value of `0` from a call to the procedure `alarm` is ambiguous: It could either mean that the alarm was quiescent, or that it was just about to run out of time. We could resolve this ambiguity if we could include "`*infinity*`" in the alarm arithmetic. In other words, we would like a *clock* that works almost like `alarm`, except that a quiescent clock is one with `*infinity*` seconds. This will make many things natural, viz,

(1) `(clock n)` on a quiescent clock returns `*infinity*`, not `0`.

(2) To stop the clock, call `(clock *infinity*)`, *not* `(clock 0)`.

(3) `(clock 0)` is equivalent to setting the clock to an infinitesimally small amount of time, viz, to cause it to raise an alarm instantaneously.

In Guile, we can define `*infinity*` as the following "number":

[9] There are other signals with their corresponding handlers, and `sigaction` can be used to set these as well.

```
(define *infinity* (/ 1 0))
```

We can define clock in terms of alarm.

```
(define clock
  (let ((stopped? #t)
        (clock-interrupt-handler
         (lambda () (error "Clock interrupt!"))))
    (let ((generate-clock-interrupt
           (lambda ()
             (set! stopped? #t)
             (clock-interrupt-handler))))
      (sigaction SIGALRM
                 (lambda (sig) (generate-clock-interrupt)))
      (lambda (msg val)
        (case msg
          ((set-handler)
           (set! clock-interrupt-handler val))
          ((set)
           (cond ((= val *infinity*)
                  ;This is equivalent to stopping the clock.
                  ;This is almost equivalent to (alarm 0), except
                  ;that if the clock is already stopped,
                  ;return *infinity*.

                  (let ((time-remaining (alarm 0)))
                    (if stopped? *infinity*
                        (begin (set! stopped? #t) time-remaining))))

                 ((= val 0)
                  ;This is equivalent to setting the alarm to
                  ;go off immediately.  This is almost equivalent
                  ;to (alarm 0), except you force the alarm
                  ;handler to run.

                  (let ((time-remaining (alarm 0)))
                    (if stopped?
                        (begin (generate-clock-interrupt) *infinity*)
                        (begin (generate-clock-interrupt) time-remaining))))

                 (else
                  ;This is equivalent to (alarm n) for n != 0.
                  ;Just remember to return *infinity* if the
                  ;clock was previously quiescent.

                  (let ((time-remaining (alarm val)))
                    (if stopped?
                        (begin (set! stopped? #f) *infinity*)
                        time-remaining))))))))))
```

The clock procedure uses three internal state variables:

(1) stopped?, to describe if the clock is stopped;

(2) clock-interrupt-handler, which is a thunk describing the user-specified part of the alarm-handling action; and

(3) `generate-clock-interrupt`, another thunk which will set `stopped?` to false before running the user-specified alarm handler.

The `clock` procedure takes two arguments. If the first argument is `set-handler`, it uses the second argument as the alarm handler.

If the first argument is `set`, it sets the time-to-alarm to the second argument, returning the time remaining from a previous setting. The code treats `0`, `*infinity*` and other values for time differently so that the user gets a mathematically transparent interface to `alarm`.

Appendix D

References

[AS96] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 2nd edition, 1996. "SICP".

[Bot] Per Bothner. Kawa, the Java-based Scheme system (`http://www.gnu.org/software/kawa`).

[Cli82] William Clinger. Nondeterministic call by need is neither lazy nor by name. In *Proc ACM Symp Lisp and Functional Programming*, pages 226–234, 1982.

[Fee] Marc Feeley. Gambit Scheme System (`http://www.iro.umontreal.ca/~gambit`).

[Fel85] Matthias Felleisen. Transliterating Prolog into Scheme. Technical Report 182, Indiana U Comp Sci Dept, 1985.

[FF96a] Daniel P Friedman and Matthias Felleisen. *The Little Schemer.* MIT Press, 4th edition, 1996.

[FF96b] Daniel P Friedman and Matthias Felleisen. *The Seasoned Schemer.* MIT Press, 1996.

[FFFK] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. How to Design Programs: An Introduction to Programming and Computing (`http://www.cs.rice.edu/CS/PLT/Teaching/Lectures/Released/curriculum`).

[Fla] Matthew Flatt. MzScheme (`http://www.cs.rice.edu/CS/PLT/packages/mzscheme`).

[FSF] FSF. Guile: Duct Tape for Bits: Project GNU's Extension Language (`http://www.gnu.org/software/guile/guile.html`).

[FWH92] Daniel P Friedman, Mitchell Wand, and Christopher T Haynes. *Essentials of Programming Languages.* MIT Press, McGraw-Hill, 1992.

[Gal] Erick Gallesio. STk (`http://kaolin.unice.fr/STk/STk.html`).

[Goe] Ben Goetter. Pocket Scheme for the H/PC and P/PC (`http://www.angrygraycat.com/scheme/pscheme.htm`).

[Hay87] Christopher T Haynes. Logic continuations. In *J Logic Program*, pages 157–176, 1987. vol 4.

[HF84] Christopher T Haynes and Daniel P Friedman. Engines Build Process Abstractions. In *Conf ACM Symp Lisp and Functional Programming*, pages 18–24, 1984.

[HFW84] Christopher T Haynes, Daniel P Friedman, and Mitchell Wand. Continuations and Coroutines. In *Conf ACM Symp Lisp and Functional Programming*, pages 293–298, 1984.

[Hun76] J A H Hunter. *Mathematical Brain-Teasers*. Dover Publications, 1976.

[Jaf] Aubrey Jaffer. SCM (`http://swissnet.ai.mit.edu/~jaffer/SCM.html`).

[KCR98] Richard Kelsey, William Clinger, and Jonathan Rees (eds). Revised^5 Report on the Algorithmic Language Scheme ("R5RS") (`http://www-swiss.ai.mit.edu/ftpdir/scheme-reports/r5rs-html/r5rs_toc.html`), 1998.

[KdRB91] Gregor Kiczales, Jim des Rivieres, and Daniel G Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[Kee89] Sonya E Keene. *Object-oriented Programming in Common Lisp: A Programmer's Guide to CLOS*. Addison-Wesley, 1989.

[McC67] John McCarthy. A Basis for a Mathematical Theory of Computation. In P Braffort and D Hirschberg, editors, *Computer Programming and Formal Systems*. North-Holland, 1967.

[MIT] MIT Scheme Team. MIT Scheme (`http://www.swiss.ai.mit.edu/projects/scheme`).

[NCS] NCSA. The Common Gateway Interface (`http://hoohoo.ncsa.uiuc.edu/cgi`).

[Que96] Christian Queinnec. *Lisp in Small Pieces*. Cambridge University Press, 1996.

[Ser] Manuel Serrano. Bigloo (`http://kaolin.unice.fr/bigloo/bigloo.html`).

[SS94] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. MIT Press, 2nd edition, 1994.

[ZMC87] Ramin Zabih, David McAllester, and David Chapman. Non-deterministic Lisp with dependency-directed backtracking. In *AAAI-87*, pages 59–64, 1987.