The Scam Reference Manual

by John C. Lusth

Revision date: September 5, 2016

Contents

Chapter 1

Notes on Terminology

This document assumes familiarity with basic programming, especially the idea of expressions and built-in functions and placing source code in files.

`Code` is displayed in a `typewriter font`, while *variables* and *filenames* are shown in an *italic font*.

Sometimes an expression will be given as an example, as in:

```
(+ 2 3 4)
```

If the value of the expression is of interest, it is shown on the next line, introduced by the symbol `->`, as in:

```
(+ 2 3 4)
-> 9
```

For the *display*, *print*, *println*, and *inspect* functions, the response is the print value, not the return value.

```
(inspect (+ 2 3 4))
-> (+ 2 3 4) is 9
```

Finally, this document assumse a familiarity with the command-line in Linux or the Mac OS.

Chapter 2

Starting Out

A word of warning: if you require fancy graphically oriented development environments, you will be sorely disappointed in the Scam Programming Language. Scam is programming at its simplest: you place code in a file and ask scam to evaluate it. You create Scam programs using your favorite text editor (mine is *vim*).

Running Scam

In a terminal window, simply type the command:

```
scam <fileName>
```

where `<fileName>` is replaced by the name of the file contining your Scam program (*i.e.* Scam source code). You should be rewarded with the output from your Scam program.

For example, create a file named *hello.scm*. In it, place the line:

```
(println "hello, world!")
```

Save the file and exit. Now run your program:

```
$ scam hello.scm
hello, world!
$
```

The `$` represents the system prompt.

Scam options

Scam takes a number of options:

-M display current memory size, then exit

-m NNN set the memory size to *NNN*

-s NNN set the stack size to *NNN*

-v display the Scam version number

-t display a full error trace for an uncaught exception

At this point, you are ready to proceed to next chapter.

Chapter 3

Comments

Comments should be used both sparingly and effectively in code. Scam allows for three kinds of comments for documentation:

file all text in a file after the two character combination `;$` is ignored.

block all text between the two character combinations `;{` and `;}` is ignored.

line all text on a line following the character `;` is ignored.

A fourth comment is provided to pass instructions to the interpreter while parsing the source code. Such comments start with the combination `;@`. Following the combination is an attribute-value pair. At the moment, only two directives are understood:

```
;@ file NEWFILENAME
```

This directive fools the parser into thinking it is parsing the file NEWFILENAME instead of the file that contains the directive. The other directive is:

```
;@ line NEWLINENUMBER
```

This directive fools the parser into thinking that the next line parsed in the source code file will have line number NEWLINENUMBER.

The file directive is useful for embedding Scam code in documents. If the document processor executes the embedded code (in the old file) by writing the code out to a new file and calling the Scam interpreter on that file, the processor can fool the interpreter into thinking it is parsing the old file. This is useful if the embedded code causes an error, as error messages will now reference the old file, not the new file. The document processor can also reset the line number to the line number at which the embedded code begins using the line directive.

Chapter 4

Literals

Scam works by figuring out the meaning or value of some code. This is true for the tiniest pieces of code to the largest programs. The process of finding out the meaning of code is known as *evaluation*.

The things whose values are the things themselves are known as *literals*. The literals of Scam can be categorized by the following types: *integers*, *real numbers*, *strings*, BOOLEANS, *symbols*, and *lists*.

Scam (or more correctly, the Scam interpreter) responds to literals by echoing back the literal itself. Here are examples of each of the types:

```
(inspect 3)
-> 3 is 3

(inspect -4.9)
-> -4.900000 is -4.900000

(inspect "hello")
-> hello is hello

(inspect #t)
-> #t is #t

(inspect (list 3 -4.9 "hello"))
-> (list 3 -4.9 "hello") is (3, -4.9, "hello")
```

Let's examine the five types in more detail.

Integers

Integers are numbers without any fractional parts. Examples of integers are:

```
(inspect 3)
-> 3 is 3

(inspect -5)
-> -5 is -5

(inspect 0)
-> 0 is 0
```

Integers must begin with a digit or a minus sign. The initial minus sign must immediately be followed by a digit.

Real Numbers

Reals are numbers that do have a fractional part (even if that fractional part is zero!). Examples of real numbers are:

```
(inspect 3.2)
-> 3.200000 is 3.200000

(inspect 4.0)
-> 4.000000 is 4.000000

(inspect 5.)
-> 5.000000 is 5.000000

(inspect 0.3)
-> 0.300000 is 0.300000

(inspect .3)
-> 0.300000 is 0.300000

(inspect 3.0e-4)
-> 0.000300 is 0.000300

(inspect 3e4)
-> 30000.000000 is 30000.000000

(inspect .000000987654321)
-> 0.000001 is 0.000001
```

Real numbers must start with a digit or a minus sign or a decimal point. An initial minus sign must immediately be followed by a digit or a decimal point. An initial decimal point must immediately be followed by a digit. Scam accepts real numbers in scientific notation. For example, $3.0 * 10^{-11}$ would be entered as 3.0e-11. The 'e' stands for exponent and the 10 is understood, so e-11 means multiply whatever precedes the e by 10^{-11}.

The Scam interpreter can hold huge numbers, limited by only the amount of memory available to the interpreter, but holds only 15 digits after the decimal point:

```
(inspect 1.2345678987654329)
-> 1.234568 is 1.234568
```

Note that Scam rounds up or rounds down, as necessary.

Numbers greater than 10^6 and less than 10^{-6} are displayed in scientific notation.

Strings

Strings are sequences of characters delineated by double quotation marks:

```
(println "hello, world!")
-> hello, world!

(println "x\nx")
-> x
   x

(println "\"z\"")
-> "z"
```

Characters in a string can be *escaped* (or quoted) with the backslash character, which changes the meaning of some characters. For example, the character n, in a string refers to the letter n while the character sequence $\backslash n$ refers to the *newline* character. A backslash also changes the meaning of the letter t, converting it into a tab character. You can also quote single and double quotes with backslashes. When other characters are escaped, it is assumed the backslash is a character of the string and it is escaped (with a backslash) in the result:

```
(println "\z")
-> z
```

Note that Scam, when asked the value of strings that contain newline and tab characters, displays them as escaped characters. When newline and tab characters in a string are printed in a program, however, they are displayed as actual newline and tab characters, respectively. As already noted, double and single quotes can be embedded in a string by quoting them with backslashes. A string with no characters between the double quotes is known as an empty string.

Unlike some languages, there is no character type in Scam. A single character `a`, for example, is entered as the string `"a"`.

Strings are treated like arrays in terms of accessing the individual 'characters' of a string. You can read more about arrays and strings in a subsequent chapter.

Symbols

A symbol is a set of characters, much like a string. Like strings, symbols evaluate to themselves. Unlike strings, symbols are not formed using a beginning quotation mark and an ending quotation mark. They are also limited in the characters that compose them. For example, a symbol cannot contain a space character while a string can. A symbol is introduced with a single quotation mark:

```
(print 'a)
-> a

(print 'hello)
-> hello
```

We we learn more about symbols and their relationship to entities called *variables* in a later chapter.

True, False, and nil

There are two special literals, `#t` and `#f`. These literals are known as the BOOLEAN values; `#t` is true and `#f` is false. Boolean values are used to guide the flow of a program. The term BOOLEAN is derived from

the last name of George Boole, who, in his 1854 paper *An Investigation of the Laws of Thought, on which are founded the Mathematical Theories of Logic and Probabilities*, laid one of the cornerstones of the modern digital computer. The so-called BOOLEAN logic or BOOLEAN algebra is concerned with the rules of combining truth values (i.e., true or false). As we will see, knowledge of such rules will be important for making Scam programs behave properly. In particular, BOOLEAN expressions will be used to control conditionals and loops.

Another special literal is `nil`. This literal is used to indicate an empty list or an empty string; it also is used to indicate something that has not yet been created. More on `nil` when we cover lists and objects.

Lists

Lists are just collections of entities. The simplest list is the empty list:

```
(inspect ())
-> nil is nil
```

Since the empty list looks kind of strange, Scam uses the symbol `nil` to represent an empty list.

One creates non-empty list by using the built-in *list function*. Here, we make a list containing the numbers 10, 100, and 1000:

```
(list 10 100 1000)
-> (10 100 1000)
```

Lists can contain values besides numbers:

```
(list 'a "help me" length)
-> (a "help me" <builtin length(item)>)
```

The first value is a symbol, the second a string, and the third item is a function. The built-in *length* function is used to tell us how many items are in a list:

```
(length (list 'a "help me" length))
-> 3
```

As expected, the *length* function tells us that the list `('a "help me" length)` has three items in it.

Lists can even contain lists!

```
(list 0 (list 3 2 1) 4)
-> (0 (3 2 1) 4)
```

A list is something known as a *data structure*; data structures are extremely important in writing sophisticated programs.

We will see more of lists in a later chapter.

Chapter 5

Combining Literals

Like the literals themselves, combinations of literals are also expressions. For example, suppose you have forgotten your times table and aren't quite sure whether 8 times 7 is 54 or 56. We can ask Scam, presenting the interpreter with the expression:

```
(* 8 7)
-> 56
```

The multiplication sign * is known as an *operator*, as it *operates* on the 8 and the 7, producing an equivalent literal value. As with all LISP/Scheme-like languages, operators like `*` are true functions and thus prefix notation is used in function calls.

The 8 and the 7 in the above expression are known as *operands*. It seems that the actual names of various operands are not being taught anymore, so for nostalgia's sake, here they are. The operand to the left of the multiplication sign (in this case the 8) is known as the *multiplicand*. The operand to the right (in this case the 7) is known as the *multiplier*. The result is known as the *product*.

The operands of the other basic operators have special names too. For addition of two operands, the left operand is known as the *augend* and the right operand is known as the *addend*. The result is known as the *sum*. For subtraction, the left operand is the *minuend*, the right the *subtrahend*, and the result as the *difference*. Finally for division (and I think this is still taught), the left operand is the *dividend*, the right operand is the *divisor*, and the result is the *quotient*.

We separate operators from their operands by spaces, tabs, or newlines, collectively known as *whitespace*.[1]

Scam always takes in an expression and returns an equivalent literal expression (*e.g.*, integer or real). All Scam operators are variadic, meaning they operate on exactly on any number of operands:

```
(+ 1 2 3 4 5)
-> 15
```

Numeric operators

If it makes sense to add two things together, you can probably do it in Scam using the + operator. For example:

[1] Computer Scientists, when they have to write their annual reports, often refer to the things they are reporting on as *darkspace*. It's always good to have a lot of darkspace in your annual report!

```
(+ 2 3)
-> 5

(+ 1.9 3.1)
-> 5.000000
```

One can see that if one adds two integers, the result is an integer. If one does the same with two reals, the result is a real. Things get more interesting when you add things having different types. Adding an integer and a real (in any order) always yields a real.

```
(+ 2 3.3)
-> 5.300000

(+ 3.3 2)
-> 5.300000
```

Adding an string to an integer (with an augend integer) yields an error; the types are not 'close' enough, like they are with integers and reals:

```
(+ 2 "hello")
-> EXCEPTION: generalException
   wrong types for '+': INTEGER and STRING
```

In general, when adding two things, the types must match or nearly match.

Of special note is the division operator with respect to integer operands. Consider evaluating the following expression:

```
15 / 2
```

If one asked the Scam interpreter to perform this task, the result will not be 7.5, as expected, but rather 7, as the division operator performs *integer division*:

```
(/ 15 2)
-> 7
```

However, we wish for a real result, we can convert one of the operands to a real, as in:

```
(/ (real 15) 2)
-> 7.500000
```

Note that Scheme would produce the rational number $\frac{15}{2}$ in this case. Scam does not have rationals, but they can be added to the language if one desires more Scheme compatibility.

The complement to integer division is the modulus operator %. While the result of integer division is the quotient, the result of the modulus operator is the remainder. Thus

```
(% 14 5)
-> 4
```

evaluates to 4 since 4 is left over when 5 is divided into 14. To check if this is true, one can ask the interpreter to evaluate:

```
(== (+ (* (/ 14  5) 5) (% 14 5)) 14)
-> #t
```

This complicated expression asks the question 'is it true that the quotient times the divisor plus the remainder is equal to the original dividend?'. The Scam interpreter will respond that, indeed, it is true.

Combining strings

To concatenate strings together, one uses the *string+* operator. Like `+` and `-`, *string+* is variadic; we can concatenate any number of strings at one time. Strings are concatenated from left to right. For examle, the expression:

```
(string+ "a" "b" "c")
```

produces the new string `"abc"`. Note that the strings passed to *string+* are unmodified.

Comparing things

Remember the Boolean literals, `#t` and `#f`? We can use the Boolean comparison operators to generate such values. For example, we can ask if 3 is less than 4:

```
(< 3 4)
-> #t
```

Evaluating this expression shows that, indeed, 3 is less than 4. If it were not, the result would be `#f`. Besides `<` (less than), there are other Boolean comparison operators: `<=` (less than or equal to), `>` (greater than), `>=` (greater than or equal to), `==` (equal to), and `!=` (not equal to).

The comparison operators are variadic:

```
(< 1 2 3)
-> #t

(< 1 2 2)
-> #f
```

Any Scam type can be compared with any other type with the `==` and `!=` comparison operators. If an integer is compared with a real with these operators, the integer is converted into a real before the comparison is made. In other cases, comparing different types with `==` will yield a value of `#f`. Conversely, comparing different types with `!=` will yield `#t` (the exception, as above, being integers compared with reals). If the types match, `==` will yield true only if the values match as well. The operator `!=` behaves accordingly.

Combining comparisons

We can combine comparisons with the Boolean logical connectives `and` and `or`:

```
(and (< 3 5) (< 4 5))
-> #t

(or (< 3 4) (< 4 5))
-> #t

(and (< 3 4) (< 5 4))
-> #f

(or (< 3 4) (< 5 4))
-> #t
```

The first interaction asks if both the expression `(< 3 4)` and the expression `(< 4 5)` are true. Since both are, the interpreter responds with true. The second interaction asks if at least one of the expressions is true. Again, the interpreter responds with true. The difference between `and` and `or` is illustrated in the last two interactions. Since only one expression is true (the latter expression being false) only the `or` operator yields a true value. If both expressions are false, both `and` and `or` returns false.

There is one more BOOLEAN logic operation, called *not*. It simply reverses the value of the given expression.

```
(not (and (< 3 4) (< 4 5)))
-> #f
```

Chapter 6

Lists, Strings, and Arrays

Recall that the built-in function *list* is used to construct a list. A similar function, called *array*, is used to construct an array populated with the given elements:

```
(array 1 "two" 'three)
-> [1 two three]
```

while a function named *allocate* is used to allocate an array of the given size:

```
(allocate 5)
-> [0 0 0 0 0]
```

Note that the elements of an allocated array are initialized to *zero*.

The functions for manipulating lists, arrays, and strings are quite similar. In the following sections, we will use the term *collection* to stand for lists, arrays, and strings.

Comparing collections

The *equal?* function is used to compare two collections of the same type:

```
(equal? (list 1 (array "23" 'hello)) (list 1 (array "23" 'hello)))
-> #t

(eq? (list 1 (array "23" 'hello)) (list 1 (array "23" 'hello)))
-> #f
```

Note that the *eq?* function tests for pointer equality and thus fails when comparing two separate lists, even though they look similar.

Strings can be compared with the *string-compare* function. Assuming a lexigraphical ordering based upon the ASCII code, then a string compare of two strings, *a* and *b*, returns a negative number if *a* appears lexigraphically before *b*, returns zero if a and b are lexigraphically equal, and returns a positive number otherwise.

```
(string-compare "abc" "bbc")
-> -1
```

Extracting elements

You can pull out an item from a collection by using the *getElement* function. With *getElement*, you specify exactly which element you wish to extract. This specification is called an *index*. The first element of a collection has index 0, the second index 1, and so on. This concept of having the first element having an index of zero is known as *zero-based counting*, a common concept in Computer Science. Here is some code that extracts the first element of a list:

```
(getElement (list "a" #t 7) 0)
-> a

(getElement (array "b" #f 11) 1)
-> #f

(getElement "howdy" 2)
-> w
```

What happens if the index is too large?

```
(getElement (list "a" #t 7) 3)
EXCEPTION: generalException
index (3) is too large
```

Not surprisingly, an error is generated. In Scam, as with many programming languages, an error is known as an *exception*.

As with Scheme, the built-in *car* and *cdr* functions returns the first item and the tail of a collection, respectively:

```
(car (list 3 5 7))
-> 3

(cdr "howdy")
-> owdy

(cdr (array 2 4 6))
-> [4 6]

(car "bon jour")
-> b
```

In addition to extracting elements of a collection, one can change the elements in a collection as well. For more information, please see the chapter on Assignment.

Chapter 7

Variables

One defines variables with the *define* function:

```
(define x 13)
```

The above expression creates a variable named x in the current scope and initializes it to the value 13. If the initializer is missing:

```
(define y)
```

an uninitialized variable error is generated:

```
EXCEPTION: uninitializedVariable
variable y is uninitialized
```

Functions

A function definition is another way to create a variable. There are two ways to define a function. The first is through a regular variable definition, where the initializer is a lambda expression:

```
(define square (lambda (x) (* x x)))

(square 3)
-> 9

(inspect square)
-> square is <function anonymous(x)>
```

The above expression defines a function that returns the square of its argument and emphasizes that the name of a function is simply a variable that is bound to a lambda (i.e. an anonymous function). The second method uses a special syntax:

```
(define (square x) (* x x))

(inspect square)
-> square is <function square(x)>
```

In either case, a variable is created and bound to some entity that knows how to compute values.

Scopes, Environments and Objects

When one defines a variable, the variable name and value are inserted into the current scope. In actuality, the name-value pair is stored in a table called an *environment*. For the predefined variable *this*, its value is the current scope or environment. For example, consider the following interaction:

```
(define n 10)

(ppTable this)
-> <object 8393>
             __label  : environment
           __context  : <environment 4495>
             __level  : 0
       __constructor  : nil
                this  : <environment 8393>
                   n  : 10
```

The *pp* function will print out the list of variables and their values for the given environment. Among other information stored in the current environment, we see an entry for n and its value is indeed 10.

The Scam object system is based upon environments. We will learn about objects in a later chapter.

Defining Variables Programatically

The function *addSymbol* is used to define variables on the fly. For example, to define a variable named x in the current scope and to initialized it to 13, one might use the following expression:

```
(addSymbol 'x 13 this)
```

You can also define functions this way:

```
(addSymbol 'square (lambda (x) (* x x)) this)
```

Since *addSymbol* evaluates all its arguments, the first argument can be any expression that resolves to a symbol, the second argument can be any expression that resolves to an appropriate value, and the third argument can be any expression that resolves to an environment or object.

Variable naming

Unlike many languages, Scam is quite liberal in regards to legal variable names. A variable can't begin with any of the these characters: `0123456789;,`'"()` nor whitespace and cannot contain any of these characters: `;,`'"()` nor whitespace. Typically, variable names start with a letter or underscore, but they do not have to. This flexibility allows Scam programmers to easily define new functions that have appropriate names. Here is a function that increments the value of its argument:

```
(define (+1 n) (+ n 1))
```

While Scam lets you name variables in wild ways:

```
(define $#1_2!3iiiiii@ 7)
```

you should temper your creativity if it gets out of hand. While the name `$#1_2!3iiiiii@` is a perfectly good variable name from Scam's point of view, it is a particularly poor name from the point of making your Scam programs readable by you and others. It is important that your variable names reflect their purpose.

Chapter 8

Assignment

Once a variable has been created, it is possible to change its value, or *binding*, Consider the following interaction with the Scam interpreter:

```
(define eyeColor 'black)    ; creation

(inspect eyeColor)          ; reference
-> eyeColor is black

(set! eyeColor 'green)      ; assignment

(eq? eyeColor 'black)       ; equality
-> #f

(eq? eyeColor 'green)       ; equality
-> #t

(assign eyeColor BROWN)     ; assignment (alternate)
```

The assignment function is not like the arithmetic operators. Recall that `+` evaluates all its arguments before performing the addition. For *set!* and *assign*, the leftmost operand is not evaluated: If it were, the assignment

```
(define x 1)
(set! x 3)
```

would be equivalent to:

```
(set! 1 3)
```

In general, an operator which does not evaluate all its arguments is known as a *special form*[1]. For *assign*, the evaluation of the first argument is suppressed.

Other functions for changing the value of a variable

Scam has another functions for changing the value of a varible:

[1] In Scam, there are no special forms. As such, *assign* is a true function and can be given a new meaning.

```
(set 'x 5)
```

which changes the current value of x to 5. It is equivalent to the *set!* function, except that it evaluates all its arguments. This is why the variable name was quoted in the example. The reason for this behavior is that, sometimes, it is useful to derive the variable name to be modified programmatically. The *set* function allows for this while the *set!* function does not.

Assignment and Environments/Objects

The assignment functions can take an environment as an optional third argument. Because the predefined variable *this* always points to the current environment, the following three expressions are equivalent:

```
(assign x 5)
(set! x 5 this)
(set (symbol "x") 5 this)
```

The *symbol* function is used to create a variable name from a string. Since environments form the basis for objects in Scam, *set!* and *set* can be used to update the instance variables of objects.

Setting elements of a collection

The *set-car!* function can be used to set the first element of a collection:

```
(define a (list 3 5 7))

(inspect a)
-> a is (3 5 7)

(set-car! a 11)

(inspect a)
-> a is (11 5 7)
```

More generally, the *setElement* function can be used to set a new value at any legal index. The first argument to *setElement* is the collection to be modified, the second is in index at which to place the new value, the third argument. Indices are numbered using zero-based counting:

```
(define a (list 3 5 7))
(setElement a 1 44)         ;index 1 refers to the 2nd element
(inspect a)
-> a is (3 44 7)
```

For strings, the new value must be a non-empty string. If the value is composed of multiple characters, the characters after the first character are ignored.

For lists, it is possible to set the tail of a list. The tail of a list is the list of all elements in the list except the first element. Here, we set the tail of list a to be list b, using *set-cdr!*:

```
(define a (list 1 2 3))
```

```
(define b (list "two" "three" "four" "five"))

(set-cdr! a b)

(inspect a)
-> a is (1 "two" "three" "four" "five")

(set-cdr! (cdr (cdr a)) (list 6))

(inspect a)
-> a is (1 "two" "three" 6)
```

The last two interactions show that any expression that resolves to a list can be passed as the first and second arguments. Note that you cannot set the tail of either an array or a string.

Chapter 9

Conditionals

Conditionals implement decision points in a computer program. Suppose you have a program that performs some task on an image. You may well have a point in the program where you do one thing if the image is a JPEG and quite another thing if the image is a GIF file. Likely, at this point, your program will include a conditional expression to make this decision.

Before learning about conditionals, it is important to learn about logical expressions. Such expressions are the core of conditionals and loops.[1]

Logical expressions

A logical expression evaluates to a truth value, in essence true or false. For example, the expression `(> x 0)` resolves to true if x is positive and false if x is negative or zero. In Scam, truth is represented by the symbol `#t` and falsehood by the symbol `#f`. Together, these two symbols are known as BOOLEAN values.

One can assign truth values to variables:

```
(define c -1)
(define z (> c 0))

(inspect z)
-> z is #f
```

Here, the variable z would be assigned true if c was positive; since c is negative, however, it is assigned false.

Logical operators

Scam has the following logical operators:

[1] We will learn about loops in the next chapter.

`=`	numeric equal to
`!=`	not equal to
`>`	greater than
`>=`	greater than or equal to
`<`	less than
`<=`	less than or equal to
`==`	pointer equality
`neq?`	pointer inequality
`eq?`	pointer equality
`equal?`	structural equality
`and`	and
`or`	or
`not`	not

The first ten operators are used for comparing two (or more) things, while the last three operators are the glue that joins up simpler logical expressions into more complex ones.

Short circuiting

When evaluating a logical expression, Scam evaluates the expression from left to right and stops evaluating as soon as it finds out that the expression is definitely true or definitely false. For example, when encountering the expression:

```
(and (!= x 0) (> (/ y x) 2))
```

if x has a value of 0, the subexpression on the left side of the *and* connective resolves to false. At this point, there is no way for the entire expression to be true (since both the left hand side and the right hand side must be true for an *and* expression to be true), so the right hand side of the expression is not evaluated. Note that this expression protects against a divide-by-zero error.

If expressions

Scam's *if* expressions are used to conditionally execute code, depending on the truth value of what is known as the *test* expression. One version of *if* has a single expression following the test expression:

Here is an example:

```
(if (equal? name "John")
    (println "What a great name you have!")
    )
```

In this version, when the test expression is true (*i.e.*, the string `"John"` is bound to the variable *name*), then the following expression is evaluated (i.e., the compliment is printed). If the test expression is false, however the expression following the test expression is not evaluated.

Here is another form of *if*:

```
(if (equal? major "Computer Science")
    (println "Smart choice!")
    (println "Ever think about changing your major?")
    )
```

In this version, *if* has two expressions following the test. As before, the first expression is evaluated if the test expression is true. If the test expression is false, however, the second expression is evaluated instead.

else-if chains and the *cond* function

You can chain `if` statements together, as in:

```
(if (== bases 4)
    (print "HOME RUN!!!")
    (if (== bases 3)
        (print "Triple!!")
        (if (== bases 2)
            (print "double!")
            (if (== bases 1)
                (print "single")
                (print "out")
                )
            )
        )
    )
```

The expression that is eventually evaluated is directly underneath the first test expression that is true, reading from top to bottom. If no test expression is true, the second expression associated with the most nested if is evaluated.

The *cond* function takes care of the awkward indentation of the above construct:

```
(cond
    ((== bases 4) (print "HOME RUN!!!"))
    ((== bases 3) (print "Triple!!"))
    ((== bases 2) (print "double!"))
    ((== bases 1) (print "single"))
    (else (print "out"))
    )
```

The general form of a *cond* function call is:

```
(cond (expr1 action1) (expr2 action2) ... (else actionN))
```

where *expr1*, *expr2*, and so on are Boolean expressions. In addition to its compactness, another advantage of a *cond* is each action portion of a clause is really an implied block. For example, suppose we wish to debug an *if* expression and print out a message if the test resolves to true. We are required to insert a *begin* block, so:

```
(if (alpha a b c)
    (beta y)
    (gamma z)
    )
```

becomes:

```
(if (alpha a b c)
    (begin
        (println "it's true!")
        (beta y)
        )
    (gamma z)
    )
```

On the other hand:

```
(cond
    ((alpha a b c)
        (beta y)
        )
    (else
        (gamma z)
        )
    )
```

becomes:

```
(cond
    ((alpha a b c)
        (println "it's true!")
        (beta y)
        )
    (else
        (gamma z)
        )
    )
```

Note the lack of a *begin* block for *cond*.

Chapter 10

Functions

Consider the equation to find the y-value of a point on the line:

```
y = 5x - 3
```

First, we assigned values to the slope, the x-value, and the y-intercept:

```
(define m 5)
(define x 9)
(define b -3)
```

Once those variables have been created, we can compute the value of y:

```
(define y (+ (* m x) b))

(inspect y)
-> y is 42
```

Now, suppose we wished to find the y-value corresponding to a different x-value or, worse yet, for a different x-value on a different line. All the work we did would have to be repeated. A *function* is a way to encapsulate all these operations so we can repeat them with a minimum of effort.

Encapsulating a series of operations

First, we will define a not-too-useful function that calculates y give a slope of 5, a y-intercept of -3, and an x-value of 9 (exactly as above). We do this by wrapping a function around the sequence of operations above. The return value of this function is the computed y value:

```
(define (y)
    (define m 5)
    (define x 9)
    (define b -3)
    (+ (* m x) b) ;the value of this expression is the return value
    )
```

There are a few things to note. The call to the *define* function indicates that a variable definition is occurring. The fact that the first argument to *define* looks like a list indicates that the variable being defined will be bound to a function and that the variable/function name is y, as it is the first member of that list. The formal parameters of the function follow the function name; since there is nothing after the y, we don't need to send any information to this function when we call it. Together, the first line is known as the *function signature*, which tells you the name of the function and how many values it expects to be sent when called.

The expressions after the function name and formal parameters are called the *function body*; the body is the code that will be evaluated (or executed) when the function is called. You must remember this: *the function body is not evaluated until the function is called.*

Finally, the return value of a function is the value of the last expression evaluated. In the above case, the expression is:

```
(+ (* m x) b)
```

Once the function is defined, we can find the value of y repeatedly.

```
(y)
-> 42

(y)
-> 42
```

Because we designed the function to take no values when called, we do not place any values between the parentheses.

Note that when we call the y function again, we get the exact same answer.

The y function, as written, is not too useful in that we cannot use it to compute similar things, such as the y-value for a different value of x. This is because we 'hard-wired' the values of b, x, and m, We can improve this function by passing in the value of x instead of hard-wiring the value to 9.

Passing arguments

A hallmark of a good function is that it lets you compute more than one thing. We can modify our y function to *take in* the value of x in which we are interested. In this way, we can compute more than one value of y. We do this by *passing* in an *argument*[1], in this case, the value of x.

```
(define (y x)
    (define m 5)
    (define b -3)
    (+ (* m x) b)
    )
```

Note that we have moved x from the body of the function to after the function name. We have also refrained from giving it a value since its value is to be sent to the function when the function is called. What we have done is to *parameterize* the function to make it more general and more useful. The variable x is now called a *formal parameter*.

Now we can compute y for an infinite number of x's:

[1] The information that is passed into a function is collectively known as *arguments*.

```
(y 9)
-> 42

(y 0)
-> -3

(y -2)
-> -13
```

What if we wish to compute a y-value for a given x for a different line? One approach would be to pass in the *slope* and *intercept* as well as x:

```
(define (y x m b)
       (+ (* m x) b)
       )
```

Now we need to pass even more information to y when we call it:

```
(y 9 5 -3)
-> 42

(y 0 5 -3)
-> -3
```

If we wish to calculate using a different line, we just pass in the new *slope* and *intercept* along with our value of x. This certainly works as intended, but is not the best way. One problem is that we keep on having to type in the slope and intercept even if we are computing y-values on the same line. Anytime you find yourself doing the same tedious thing over and over, be assured that someone has thought of a way to avoid that particular tedium. If so, how do we customize our function so that we only have to enter the slope and intercept once per particular line? We will explore one way for doing this. In reading further, it is not important if you understand all that is going on. What is important is that you know you can use functions to run similar code over and over again.

Creating functions on the fly

Since creating functions is hard work (lots of typing) and Computer Scientists avoid unnecessary work like the plague, somebody early on got the idea of writing a function that itself creates functions! Brilliant! We can do this for our line problem. We will tell our creative function to create a y function for a particular slope and intercept! While we are at it, let's change the variable names m and b to *slope* and *intercept*, respectively:

```
(define (createLine slope intercept)
    (define (y x)
        (+ (* slope x) intercept)
        )
    y    ; the value of y is returned, y is NOT CALLED!
    )
```

The *createLine* function creates a y function and then returns it. Note that this returned function y takes one value when called, the value of x.

So our creative *createLine* function simply defines a y function and then returns it. Now we can create a bunch of different lines:

```
(define a (createLine 5 -3))
(define b (createLine 6 2))

(a 9)
-> 42

(b 9)
-> 56

(a 9)
-> 42
```

Notice how lines a and b remember the slope and intercept supplied when they were created.[2] While this is decidedly cool, the problem is many languages (for example C, C++, and Java[3]) do not allow you to define functions that create other functions. Fortunately, Scam, Python, and most functional languages allow this.

While this might seem a little mind-boggling, don't worry. The things you should take away from this are:

- functions encapsulate calculations
- functions can be parameterized
- functions can be called
- functions return values

[2]The local function y does not really remember these values, but at this point in time, this is a good enough explanation.
[3]C++ and Java, as well as Scam, give you another approach, *objects*. We will discuss objects in a later chapter.

Chapter 11

Input and Output

Scam uses a *port* system for input and output. When Scam starts up, the current input port defaults to *stdin* (the keyboard) and the current output port defaults to *stdout* (the screen).

To change these ports, one first creates new port and then sets the port. For example, to read from a file (say "data") instead of the keyboard, first create a file port:

```
(define p (open "data" 'read))   ; p points to a port
(define oldInput (setPort p))
...                              ; read stuff from the file data
(setPort oldInput)               ; restore the old input port
```

Once the port is set, all input will come from the new port. The *setPort* function, in addition to setting the port, returns the old port so that it eventually can be restored, if needed.

To change the output port, the procedure is similar, except the symbol `'write` is used instead of `'read`.

```
(define p (open "data" 'write))   ; p points to a port
(define oldOutput (setPort p))
...                               ; write stuff to the file data
(setPort oldOutput)               ; restore the old output port
```

Opening a file in `'write` mode overwrites the file; to append content to an existing file, use the `'append` symbol instead.

Scam only allows a limited number of ports to be open at any given time. If you no longer need a port, close it with the built-in function *close*, which takes a port as its sole argument:

```
(close p)
```

You can retrieve the current input and output ports with the function calls:

```
(getInputPort)
(getOutputPort)
```

respectively.

Reading

Scam supplies built-in functions for reading characters, integers, reals, strings, and whitespace delimited tokens:

```
(assign s (readChar))
(assign i (readInt))
(assign r (readReal))
(assign s (readString))
(assign t (readToken))
(assign s (readRawChar))
(assign u (readUntil stopCharacterString))
(assign w (readWhile continueCharacterString))
```

The first five functions listed skip any whitespace preceeding the entity they are to read. The last three functions do not skip whitespace.

Both the *readChar* and the *readToken* functions return strings. Scam uses the same rules as the C programming language for what characters constitute an integer and a real. None of these functions take an argument; they use the current input port.

To read a symbol, use the *symbol* function in conjunction with the *readToken* function:

```
s = symbol(readToken());
```

To read a line of text, use the built-in *readLine* function:

```
(assign l (readLine))
```

The *readLine* function reads up to, and including, the next newline character, but the newline is not part of the returned string.

The *pause* function always reads from *stdin*, regardless of the current input port. It reads (and discards) a line of text (up to and including the newline). Its purpose is to pause execution of a program for debugging purposes.

Three other reading functions are useful for scanning text. The first is *readRawChar*, which returns a string containing the next character in the file, regardless of whether that character is whitespace or not. The second is *readUntil*, which is passed a string of characters that is used to control the read. For example,

```
(readUntil " \t\n")
```

will start reading at the current point in the file and return a string of all characters read up to point where a character in a given string is encountered. The character that caused the read to stop is pushed back into the input stream and will be the next character read.

The *readWhile* function is analogous, stopping when a character not in the given string is encountered.

Writing

Most output functions write to the current output port.

The simplest output function is *display*. It takes a single argument, which can be any Scam object:

```
(display "Hello, world!\n")
```

The character sequence `\` followed by `n` indicate that a newline is to be displayed.

More useful than *display* are the functions *print* and *println*, in that they take any number of arguments:

```
(print "(f x) is " (f x) "\n")
(println "(f x) is " (f x))
```

The *println* function is just like *print*, except it outputs a newline after the displaying the last argument. Thus, the two calls above produce the same output.

When a string is printed, the quote marks are not displayed. Likewise, when a symbol is printed, the quote mark is not displayed.

The *inspect* function prints out the unevaluated version of its argument followed by the arguments evaluation value:

```
(inspect (f x))
-> (f x) is 3
```

The *inspect* function always prints to *stdout*, regardless of the current output port. Like *pause*, it is useful for debugging.

Pretty printing

The function *pp* acts much like *display*, unless it is passed an environment/object. In such cases, it prints out a table listing the variables defined in that scope. Since functions, thunks, exceptions, and errors are all encoded as objects, *pp* can be used to inspect them in greater detail. For example, consider this definition of square:

```
(define (square x)
    (* x x)
    )
```

Printing the value of square using `(print square)` yields:

```
<function square(x)>
```

In contrast, using `(ppTable square)` yields:

```
<object 8573>
           label  : closure
         context  : <object 8424>
            name  : square
      parameters  : (x)
            code  : (begin (* x x))
```

Yet a third way to look at a functgion is with the *pretty* function:

```
(include "pretty.lib")

(pretty square)
```

outputs:

```
(define (square x)
    (* x x)
    )
```

To use the *pretty* function, you must include the *pretty.lib* library. You can use the *pretty* function to display any arbitrary piece of Scam code:

```
(pretty '(begin (println "testing square...") (square 3)))
```

produces:

```
(begin
    (println "testing square...")
    (square 3)
    )
```

To change the level of indent, set the variable *__pretty-indent* appropriately:

```
(assign __pretty-indent "  ")
(pretty square)
```

produces:

```
(define (square x)
  (* x x)
  this
  )
```

The initial indentation defaults to no indentation. To change this, call the *ppCode* function:

```
(ppCode square "    ")
```

Formatting

The *fmt* function can be used to format numbers and strings if the default formatting is not acceptable. It uses the C programming language formatting scheme, taking a formatting specification as a string, and the item to be formatted. The function returns a string.

For example,

```
(string+ "<" (fmt "%6d" 3) ">")
-> "<     3>"

(string+ "<" (fmt "%-6d" 3) ">")
-> "<3     >"
```

A format specification begins with a percent sign and is usually followed by a number representing the width (in number of characters) of the resulting string. If the width is positive, the item is right justified in the resulting string. If the width is negative, the item is left justified. After any width specification is a character specifying the type of the item to be formatted: `d` for an integer, `f` for a real number, and `s` for a string.

The format specification is quite a bit more sophisticated than shown here. You can read more on a Linux system by typing the command `man 3 printf` at the system prompt.

Testing for end of file

The *eof?* function can be used to test whether the last read was successful or not. The function is NOT used to test if the *next* read would be successful. Here is a typical use of *eof?* in tokenizing a file:

```
(define t (readToken))
(while (not(eof?))
    (store t)
    (assign t (readToken))
    )
```

Pushing back a character

Sometimes, it is necessary to read one character too many from the input. This happens in cases like advancing past whitespace in the input. Here is a typical whitespace-clearing loop:

```
(define ch (readRawChar))
(while (space? ch))
    (assign ch (readRawChar))
    )

; last character read wasn't whitespace
; so push it back to be read again later

(pushBack ch)
```

The *pushBack* function takes a string as its sole argument, but only pushes back the first character of the string; subsequent characters in the string are ignored.

Chapter 12

Scopes, Environments, and Objects

A *scope* holds the current set of variables and their values. As with Scheme, Scam implements scope using environments. For all Scam programs, the built-in functions are stored in the outermost scope, while the functions defined in the main library, *main.lib*, are stored in the next-to-the-outermost scope. Finally, the user is given a global scope of his or her own. These three scopes make up the global scope for all Scam programs.

One can write a very simple Scam program to see the three levels of the global scope that are in existence for all programs:

```
(print "USER SCOPE: ")
(ppTable this)
(print "LIBRARY SCOPE: ")
(ppTable (dot this __context)
(print "BUILT-IN SCOPE: ")
(ppTable (dot (dot this __context) __context))
```

Running this program yields the following output:

```
USER SCOPE: <object 8675>
               __label  : environment
             __context  : <object 4777>
               __level  : 0
         __constructor  : nil
                this  : <object 8675>
                 ...
LIBRARY SCOPE: <object 4777>
             __label  : environment
           __context  : <object 62>
             __level  : 0
       __constructor  : nil
                this  : <object 4777>
                code  : <function code($s)>
        stream-null?  : <function stream-null?(s)>
          stream-cdr  : <function stream-cdr(s)>
          stream-car  : <function stream-car(s)>
         cons-stream  : <function cons-stream(# a $b)>
                 ...
BUILT-IN SCOPE: <object 62>
```

```
          __label  : environment
        __context  : nil
          __level  : 0
    __constructor  : nil
             this  : <object 62>
          ScamEnv  : [ORBIT_SOCKETDIR=/tmp/orbit-lusth,...]
         ScamArgs  : [scam,scope.s]
      stack-depth  : <builtIn stack-depth()>
         set-cdr!  : <builtIn set-cdr!(spot value)>
         set-car!  : <builtIn set-car!(spot value)>
              ...
```

We can see from this output that every environment has five predefined variables. So that they are accessible and available for manipulation with reduced likelihood of being trashed, the first four begin with two underscores. The predefined variables are:

__label Environments are objects in Scam. The *__label* field is used to distinguish the native objects from each other. For environments, the label value is *environment*. Other label values are *closure*, *thunk*, *error*, and *exception*.

__context This field holds the enclosing scope. For the outermost scope, the value of this field is *nil*.

__level This fields holds the nesting level of dynamic scopes or, in other words, the depth of the call tree for a particular function.

__constructor This field holds the closure for which the environment was constructed. If the environment did not arise from a function call, the value of this field is *nil*.

this A self-reference to the current environment.

As variables are defined, their names and values are inserted into the most local scope at the time of definition.

Nesting scopes

If a definition occurs in a nested begin block, it belongs to the scope in which the begin block is found, recursively. Thus, when the following code is evaluated:

```
(define (f w)
    (define x 1)
    (begin
        (define y 2)
        (begin
            (define z 3)
            )
        )
    (ppTable this)
    )
(f 0)
```

We see the following output:

```
<object 9726>
         __label  : environment
       __context  : <object 9024>
         __level  : 1
   __constructor  : <function f(w)>
            this  : <object 9726>
               z  : 3
               y  : 2
               x  : 1
               w  : 0
```

Note that the nested defines of y and z were promoted to the scope of the function body, mirroring the behavior of Python.

To force a block to have its own scope, as in C, C++, and Java, one can use the *scope* function:

```
(define (g w)
    (define x 1)
    (scope
        (define y 2)
        (begin
            (define z 3)
            )
        )
    (ppTable this)
    )
(g 0)
```

Now the output reflects that fact that y and z are in their own separate scope:

```
<object 9805>
         __label  : environment
       __context  : <object 9024>
         __level  : 1
   __constructor  : <function g(w)>
            this  : <object 9805>
               x  : 1
               w  : 0
```

Because y and z are now in an enclosed scope, they are no longer visible.

Chapter 13

Objects

In the Scam world, an object is simply a collection of related variables. You've already been exposed to objects, although you may not have realized it. When you created a variable, you modified the environment, which is an object. When you defined a function, you created an object. To view an object, we use the predefined function *pp*.[1] Evaluating the code:

```
(define (square x)
    (* x x)
    )

(ppTable square)
```

yields something similar to the following output:

```
<object 10435>
         __label  : closure
       __context  : <object 10381>
            name  : square
      parameters  : (x)
            code  : (begin (* x x))
```

We see that the *square* function is made up of five *fields*.[2] These fields are: *__label*, *__context*, *name*, *parameters*, and *code*.

Usually, an object lets you look at its individual components. For example:

```
(println "square's formal parameters are: " (get 'parameters square))
```

yields:

```
square's formal parameters are: (x)
```

We use the function *get* to extract the fields of an object. The first argument to *get* is an expression that should resolve to a symbol to be gotten, while the second is the object holding the field.

[1] The *pp* in the function names stands for *pretty print* which means to print out something so it is 'pretty looking'.

[2] Some people use the term *component* or *instance variable* instead of *field*. Also, if you try this, you may see different numbers than 10435 and 10381. These numbers represent the address of the object in memory.

Creating objects

It is easy to create your own objects. First you must make a *constructor*. A constructor is just a function that returns the predefined variable *this*. Suppose you want a constructor to create an object with fields *name* and *age*. Here is one possibility:

```
(define (person)
    (define name)
    (define age)
    this
    )
```

We can create an object simply by calling the constructor:

```
(define p (person))
```

The variable *p* now points to a *person* object and we can use *p* and the *set* operator to set the fields of the person object:

```
(set 'name "Boris" p)
(set 'age 33 p)
(inspect (get 'name p))
```

The *set* function takes three arguments. The first should evaluate to the field name, the second to the new field value, and the third to the object to update. Evaluating this code yields the following output:

```
(get 'name  p) is Boris
```

Scam allows an alternative form for getting values from an object. This form uses function call syntax. For example, the following two expressions are equivalent:

```
(get 'name p)
(p 'name)
```

The advantage of the latter form is that if the retrieved field is itself an object, you can chain field names together. For example if *n* points to a linked list:

```
(define (node value next) this)
(define n (node 'a (node 'b (node 'c nil))))
```

you can get the value of the third node in the list with either expression:

```
(get 'value (get 'next (get 'next n)))
(n 'next 'next 'value)
```

To set the value of a field using a similar syntax, one calls the *set** function. Here, both calls set the value of the third node in the list to 5:

```
(set 'value 5 (get 'next (get 'next n)))
(set* n 'next 'next 'value 5)
```

Note that for *set*, the object is the last argument while for *set**, the object is the first.

Initializing fields

It is often convenient to give initial values to the fields of an object. Here is another version of *person* that allows us to do just that when we create the object:

```
(define (person name age) this)

(define p (person "Boris" 33))

(inspect (p 'name))
```

The output is the same:

```
(p 'name) is Boris
```

In general, if a field is to be initialized when the object is constructed, make that field a formal parameter. If not, make the field a locally declared variable.

Adding Methods

Objects can have methods as well.[3] Here's a version of the *person* constructor that has a *birthday* method.

```
(define (person name age)
    (define (birthday)
        (println "Happy Birthday, " name "!")
        (++ age)
        )
    this
    )

(define p (person "Boris" 33))
((p 'birthday))
(inspect (p 'age))
```

The output of this code is:

```
Happy Birthday, Boris!
(p 'age) is 34
```

In summary, one turns a function into a constructor by returning *this* from a function. The local variables, including formal parameters, become the fields of the function while any local functions serve as methods.

[3] A method is just another name for a local function.

The variable *this* is not this

It is tempting to think that the predefined variable *this* is equivalent to **this** in Java. In Scam, *this* always refers to the current scope/environment, while in Java, **this** refers to the current object. The difference is noticable in object methods. For example, consider the following constructor:

```
(define (f x)
    (define (g y)
        this
        )
    this
    )
```

If we create an *f* object:

```
(define a (f 0))
```

and then save the return value of the *g* method:

```
(define b ((f 'g) 1))
```

then *a* and *b* are not the same objects; *b* would be a 'sub-object', as it were, of *a*. In fact:

```
(eq? a b)
```

returns false, while:

```
(eq? a (b '__context))
```

evaluates to true. To have the *g* method return a pointer to the object created by *f*, we would have *g* return its context:

```
(define (f x)
    (define (g y)
        __context
        )
    this
    )
```

Objects and Types

If you were to ask an object, "What are you?", most would respond, "I am an environment!". The *type* function is used to ask such questions:

```
(define p (person "betty" 19))
(inspect (type p))
```

yields:

```
(type p) is environment
```

This is because the predefined variable *this* always points to the current environment and when we return *this* from a function, we are returning an environment. Because environments are objects and vice versa, making objects in Scam is quite easy.

While the *type* function is often useful, we sometimes would like to know what kind of specific object an object is. For example, we might like to know whether or not *p* is a *person* object. That is to say, was *p* created by the *person* function/constructor?. One way to do this is to ask the constructor of the object if it is the *person* function. Luckily, all objects carry along a pointer to the function that constructed them:

```
(define p (person "veronica" 20))
(inspect (p '__constructor 'name))
```

yields:

```
(p '__constructor 'name) is person
```

So, to ask if *p* is a person, we would use the following expression:

```
(if (and (eq? (type p) 'environment)
         (eq? (p '__constructor 'name) 'person)) ...
```

Since this construct is rather wordy, there is a simple function, named *is?*, that you can use instead:

```
(if (is? p 'person) ...
```

The *is* function works for non-objects too. All of the following expressions are true:

```
(is? 3 'INTEGER)
(is? 3.4 'REAL)
(is? "hello" 'STRING)
(is? 'blue 'SYMBOL)
(is? (list 1 2 3) 'CONS)
(is? (array "a" "b" "c") 'ARRAY)
(is? (person 'veronica 20) 'object)
(is? (person 'veronica 20) 'environment)
(is? (person 'veronica 20) 'person)
```

Other objects in Scam

While environments constitute the bulk of objects in Scam, two other object types are built into Scam. They are closures (seen the the first section) and error objects. The main library adds in a third object type known as a thunk.

An error object is generated when an exception is caught. The fields of an error object are code, value, and trace. A thunk is an expression and an environment bundled together. Thunks are used to delay evaluation of an expression for a later time. The fields of a thunk are *code* and *__context*. You can learn more about error objects and thunks in subsequent chapters.

Fun with *objects*

Because of the flexibility of Scam, one can add Java-like display behavior to objects. In Java, if an object has a method named *toString*, then if one attempts to print the object, the *toString* method is called to generate the print value of the object.

Here is an example:

```
(define (person name age)
    (define (birthday)
        (println "Happy Birthday, " name "!")
        (++ age)
        )
    (define (toString)
        (string+ name "(age " age ")")
        )
    this
    )

(define p (person "boris" 33))
```

Given the above definition of *person*, printing the value of *p*:

```
(println p)
```

yields:

```
<object 23452>
```

Uh oh. The *toString* method wasn't called! This is because the current version of *println*, defined in *main.lib*, does not understand the *toString* method. No problem here, we'll just reassign *display*, since *println* calls *print* and *print* calls *display* to do the actual printing:

```
(define (display  item)
    (if (and (object? item) (local? 'toString item))
        (__display ((item 'toString)))
        (__display item)
        )
    )
```

The main library binds the original version of *display* to the symbol *__display* for safe-keeping.

Note that this new version of *display* is only found in the local environment, so the current versions of *println* and *print*, defined in an outer scope, cannot see the new version of *display* in the current scope. To do so would be a scope violation. We solve this problem by *cloning* the two printing functions. The process of cloning produces new closures with the local environment as the definition environment. In all other respects, the cloned function is identical. Thus, when the new *print* calls *display*, the new, local version will be found.

```
(include "reflection.lib")
```

```
(define print (clone print))
(define println (clone println))

(println p)
```

The *reflection* library must be included to access the *clone* function.

Now, printing the object p yields:

```
boris (age 33)
```

Chapter 14

Encapsulation, Inheritance and Polymorphism

A formal view of object-orientation

Scam is a fully-featured object-oriented language. What does that mean exactly? Well, to begin with, a programming language is considered object-oriented if it has these three features:

1. encapsulation
2. inheritance
3. polymorphism

Encapsulation in this sense means that a programmer can bundle *data* and *methods* into a single entity. We've seen that a Scam function can have local variables and local functions. So, if we consider local variables (including the formal parameters) as data and local functions as methods, we see that Scam can encapsulate in the object-oriented sense.

Inheritance is the ability to use the data and methods of one kind of object by another as if they were defined in the other object to begin with.

Polymorphism means that an object that inherits appears to be both kinds of object, the kind of object it is itself and the kind of object from which it inherits.

Simple encapsulation

The previous chapter was concerned with encapsulation; let us review.

A notion that simplifies encapsulation in Scam is to use environments themselves as objects. Since an environment can be thought of as a table of the variable names currently in scope, along with their values, and an object can be thought of as a table of instance variables and method names, along with their values, the association of these two entities is not unreasonable.

Thus, to create an object, we need only cause a new scope to come into being. A convenient way to do this is to make a function call. The call causes a new environment to be created, in which the arguments to the call are bound to the formal parameters and under which the function body is evaluated. Our function need only return a pointer to the current execution environment to create an object. Under such a scenario, we can view the the function definition as a class definition with the formal parameters serving as instance variables and locally defined functions serving as instance methods.

Scam allows the current execution environment to be referenced and returned. Here is an example of object creation in Scam:

```
(define (bundle a b)
    (define (total base) (+ base a b))
    (define (toString) (string+ "a:" a ", b:" b))
    this    ;return the execution environment
    )

(define obj (bundle 3 4))

(inspect ((obj 'display)))  ;call the display function
(inspect ((obj 'total) 0))  ;call the total function
```

The variable *this* is always bound to the current execution enironment or scope. Since, in Scam, objects and environments the same, this can be roughly thought of as a self reference to an object. The the object can be called as if it were a function as long as the arguments in the call resolve to field names. The *inspect* function prints the unevaluated argument followed by its evaluation.

Running the above program yields the following output:

```
((obj 'display)) is a:3, b:4
((obj 'total) 0) is 7
```

It can be seen from the code and the output that encapsulation via this method produces objects that can be manipulated in an intuitive manner.

It should be stated that encapsulation is considered merely a device for holding related data together; whether the capsule is transparent or not is not considered important for the purposes of this paper. Thus, in the above example, all components are publicly visible.

Three common types of inheritance

Any specification of inheritance semantics must be (relatively) consistent with the afore-mentioned intuition about inheritance. With regards to inheritance behavior pragmatics, there seems to be three forms of inheritance behavior that make up this intuition. Taking the names given by Bertrand Meyer in "The Many Faces of Inheritance: A Taxonomy of Taxonomies", the three are *extension*, *reification*, and *variation*. In extension inheritance, the heir simply adds features in addition to the features of the ancestor; the heir is indistinguishable from the ancestor, modulo the original features. In reification inheritance, the heir completes, at least partially, an incompletely specified ancestor. An example of reification inheritance is the idea of an abstract base class in Java. In variation inheritance, the heir adds no new features but overrides some of the ancestor's features. Unlike extension inheritance, the heir is distinguishable from the ancestor, modulo the original features. The three inheritance types are not mutually exclusive; as a practical matter, all three types of inheritance could be exhibited in a single instance of general inheritance. Any definition of inheritance should capture the intent of these forms. As it turns out, it is very easy to implement these three forms of inheritance in Scam.

Scam uses a novel approach to inheritance. Other languages processors pass a pointer to the object in question to all object methods. This pointer is known as a self-reference. This passing of a self-reference may be hidden from the programmer or may be made explicit. In any case, Scam displenses with self-references and implements inheritance through the manipulation of scope.

Extension inheritance

In order to provide inheritance by manipulating scope, it must be possible to both get and set the static scope, at runtime, of objects and function closures. There are two functions that will help us perform those tasks. They are *getEnclosingScope* and *setEnclosingScope* and are defined in the supplemental library, *inherit.lib*. While at first glance it may seem odd to change a static scope at runtime, these functions translate into getting and setting the__*context* pointer of an environment (or closure).

Recall that in extension inheritance, the subclass strictly adds new features to a superclass and that a subclass object and a superclass object are indistinguishable, behavior-wise, with regards to the features provided by the superclass. Consider two objects, *child* and *parent*. The extension inheritance of *child* from *parent* can be implemented with the following pseudocode:

```
setEnclosingScope(parent,getEnclosingScope(child));
setEnclosingScope(child,parent);
```

As a concrete example, consider the following Scam program:

```
(include "inherit.lib")

(define (c) "happy")
(define (parent)
    (define (b) "slap")
    this
    )
(define (child)
    (define (a) "jacks")
    (define temp (parent))
    (setEnclosingScope temp (getEnclosingScope this))
    (setEnclosingScope this temp)
    this
    )

(define obj (child))

(inspect ((obj 'b)))
(inspect ((obj 'a)))
(inspect ((obj 'c)))
```

Running this program yields the following output:

```
((obj 'a)) is jacks
((obj 'b)) is slap
((obj 'c)) is happy
```

The call to *a* immediately finds the child's method. The call to *b* results in a search of the child. Failing to find a binding for *b* in *child*, the enclosing scope of *child* is searched. Since the enclosing scope of *child* has been reset to *parent*, *parent* is searched for *b* and a binding is found. In the final call to *c*, a binding is not found in either the child or the parent, so the enclosing scope of *parent* is searched. That has been reset to CHILD's enclosing scope. There a binding is found. So even if the parent object is created in a scope different from the child, the correct behavior ensues.

For an arbitrarily long inheritance chain, *p1* inherits from *p2*, which inherits from *p3* and so on, the most distant ancestor of the child object receives the child's enclosing scope:

```
setEnclosingScope(pN,getEnclosingScope(p1))
setEnclosingScope(p1,p2);
setEnclosingScope(p2,p3);
...
setEnclosingScope(pN-1,pN)
```

It should be noted that the code examples in this and the next subsections hard-wire the inheritance manipulations. As will be seen further on, Scam automates these tasks.

Reification inheritance

As stated earlier, reification inheritance concerns a subclass fleshing out a partially completed implementation by the superclass. A consequence of this finishing aspect is that, unlike extension inheritance, the superclass must have access to subclass methods. A typical approach to handling this problem is rather inelegant, passing a reference to the original object as hidden, or not so hidden, parameter to all methods. Within method bodies, method calls are routed through this reference. Inheritance in Python is done just this way; the object reference is bound to the first formal parameter in all object methods.

That said, our approach for extension inheritance does not work for reification inheritance. Suppose a parent method references a method provided by the child. In Scam, when a function definition is encountered, a closure is created and this closure holds a pointer to the definition environment. It is this pointer that implements static scoping in such interpreters.

For parent methods, then, the enclosing scope is the parent. When the function body of the method is being evaluated, the reference to the method supplied by the child goes unresolved, since it is not found in the parent method. The enclosing scope of the parent method, the parent itself, is searched next. The reference remains unresolved. Next the enclosing scope of the parent is searched, which has been reset to the enclosing scope of the child. Again, the reference goes unresolved (or resolved by happenstance should a binding appear in some enclosing scope of the child).

The solution to this problem is to reset the enclosing scopes of the parent methods to the child. In pseudocode:

```
setEnclosingScope(parent,getEnclosingScope(child));
setEnclosingScope(child,parent);
for each method m of parent
    setEnclosingScope(m,child)
```

Now, reification inheritance works as expected. Here is an example:

```
(include "inherit.lib")

(define (parent)
    (define (ba) (string+ (b) (a)))
    (define (b) "slap")
    this
    )
(define (child)
    (define (a) "jacks")
```

```
        (define temp (parent))
        (setEnclosingScope temp (getEnclosingScope this))
        (setEnclosingScope this temp)
        (setEnclosingScope (temp 'ba) this)
        this
        )

    (define obj (child))

    (inspect ((obj 'ba)))
```

The output of this program is:

```
    ((obj 'ba)) is "slapjacks"
```

As can be seen, the reference to *a* in the function *ba* is resolved correctly, due to the resetting of *ba*'s enclosing scope by *child*.

For longer inheritance chains, the pseudocode of the previous subsection is modified accordingly:

```
    setEnclosingScope(pN,getEnclosingScope(p1))
    setEnclosingScope(p1,p2);
    for each method m of p2: setEnclosingScope(m,p1)
    setEnclosingScope(p2,p3);
    for each method m of p3: setEnclosingScope(m,p1)
    ...
    setEnclosingScope(pN-1,pN)
    for each method m of pN: setEnclosingScope(m,p1)
```

All ancestors of the child has the enclosing scopes of their methods reset.

Variation inheritance

Variation inheritance captures the idea of a subclass overriding a superclass method. If functions are naturally virtual (as in Java), then the overriding function is always called preferentially over the overridden function.

If *child* is redefined as follows:

```
    (define (child)
        (define (b) "jumping")
        (define (a) "jacks")
        (define temp (parent))
        (setEnclosingScope temp (getEnclosingScope this))
        (setEnclosingScope this temp)
        (setEnclosingScope (temp 'ab) this)
        this
        )
```

then the new version of *b* overrides the parent version. The output now becomes:

```
((obj 'ba)) is jumpingjacks
```

This demonstrates that both reification and variation inheritance can be implemented using the same mechanism. Another benefit is that instance variables and instance methods are treated uniformly. Unlike virtual functions in Java and C++, instance variables in those languages shadow superclass instance variables of the same name, but only for subclass methods. For superclass methods, the superclass version of the instance variable is visible, while the subclass version is shadowed. With this approach, both instance variables and instance methods are virtual, eliminating the potential error of shadowing a superclass instance variable. Here is an example:

```
(include "inherit.lib")

(define (parent)
    (define x 0)
    (define (toString) (string+ "x:" x))
    this
    )
(define (child)
    (define x 1)
    (define temp (parent))
    (setEnclosingScope temp (getEnclosingScope this))
    (setEnclosingScope this temp)
    (setEnclosingScope (temp 'toString) this)
    this
    )

(define p-obj (parent))
(define c-obj (child))

(inspect ((p-obj 'toString)))
(inspect ((c-obj 'toString)))
```

The output:

```
((p-obj 'toString)) is x:0
((c-obj 'toString)) is x:1
```

demonstrates the virtuality of the instance variable x. Even though the program calls the superclass version of *toString*, the subclass version of x is referenced.

Implementing Inheritance in Scam

Since environments are objects in Scam, implementing the *getEnclosingScope* and *setEnclosingScope* functions are trivial:

```
(define (setEnclosingScope a b) (set '__context b a))
(define (getEnclosingScope a) (a '__context))
```

Moreover, the task of resetting the enclosing scopes of the parties involved can be automated. Scam provides a library, named *inherit.lib*, written in Scam that provides a number of inheritance mechanisms. The first

(and simplest) is ad-hoc inheritance. Suppose we have objects a, b, and$c$ and we wish a to inherit from b and c (and if both b and c provide functionality, we prefer b's implementation). To do so, we call the *mixin* function:

```
(mixin a b c)
```

A definition of *mixin* could be:

```
(define (mixin object @)  ; @ points to a list of remaining args
    (define outer (getEnclosingScope object))
    (define spot object)
    (while (not (null? (cdr @)))
        (define current (car @))
        (resetClosures current object)
        (setEnclosingScope spot current)
        (assign spot current)
        (assign @ (cdr @)
        )
    (setEnclosingScope (car @) outer)
    (resetClosures (car @) object)
    )
```

The other type of inheritance emulates the *extends* operation in Java. For this type of inheritance, the convention is that an object must declare a parent. In the constructor for an object, the parent instance variable is set to the parent object, usually obtained via the parent constructor. Here is an example:

```
(define (b)
    (define parent nil)
    ...
    this
    )
(define (a)
    (define parent (b))     ;setting the parent
    ...
    this
    )
```

Now, to instantiate an object, the *new* function is called:

```
(define obj (new (a)))
```

The *new* function follows the parent pointers to reset the enclosing scopes appropriately. Here is a possible implementation of *new*, which follows the definition of *mixin* closely:

```
(define (new object)
    (define outer (getEnclosingScope object))
    (define spot object)
    (define current (spot 'parent))
    (while (not (null? current))
```

```
        (resetClosures current object)
        (setEnclosingScope spot current)
        (assign spot current)
        (define current (spot 'parent))
        )
    (setEnclosingScope spot outer)
    (resetClosures spot object)
    )
```

Other forms of inheritance are possible as well. The flexibility of this approach does not require inheritance to be built into the language.

Darwinian versus Lamarckian Inheritance

The behavior of the inheritance scheme implemented in this paper differs from the inheritance schemes of the major industrial-strength languages in one important way. In Java, for example, if a superclass method references a variable defined in an outer scope (this can happen with nested classes), those references are resolved the same way, *whether or not* an object of that class was instantiated as a stand-alone object or as part of an instantiation of a subclass object. This is remeniscent of the inheritance theory of Jean-Baptiste Lamarck, who postulated that the environment influences inheritance. In Java, the superclass retains traces of its environment which can influence the behavior of a subclass object.

With selfless inheritance, the static scopes of the superclass objects are replaced with the static scope of the subclass object, a purely Darwinian outcome. The superclass objects contributes the methods and instance variables (say, the genes of the superclass) but none of the environmental influences. Thus, the subclass object must provide bindings for the non-local references either through its own definitions or in its definition environment.

Polymorphism

Polymorphism is a word that literally means "having multiple shapes". With regards to object-orientation, polymorphism means one kind of object can look like another kind of object. One concrete example of this involves inheritance: if object *child* inherits from object *parent*, does the *child* look like a *parent* object as well as a *child* object? In other words, can a variable that points to a parent object also point to a child object? This question is of critical importance for statically typed languages such as C++ and Java, but is not so important for dynamically-typed languages like Scam. This is because a Scam variable can point to any type of entity, so the question of whether a variable can point to either a child or a parent is moot.

That said, it is often useful in an dynamically-typed, object-oriented language to ask whether or not a variable points to an object that looks like some other object. The *is?* function, introduced in the previous chapter, can answer these questions. Consider this set of constructors:

```
(include "inherit.lib")

(define (p)
    (define parent nil)
    this
    )

(define (c)
    (define parent (p))
    this
    )
```

Here, we have *b* inheriting from *a*. If we create an *a* object and a *b* object using *new*:

```
(define p-obj (new (p)))
(define c-obj (new (c)))
```

we can now ask what kinds of objects they are:

```
(inspect (is? p-obj 'p))
(inspect (is? c-obj 'c))
```

As expected, the output is:

```
(is? p-obj 'p) is #t
(is? c-obj 'c) is #t
```

However, a typical view in the object-oriented world is that a child object *is also* a parent object, since it inherits all the fields and methods of the *parent*. The *is?* function conforms to this idea:

```
(is? c-obj 'p)
```

evaluates to true. Conversely, the typical view is that the parent object *is not* a child object. The expression:

```
(is? p-obj 'c)
```

evaluates to false.

Chapter 15

Parameter Passing

There are (at least) six historical and current methods of passing arguments to a function when a function call is made. They are:

- *call-by-value*
- *call-by-reference*
- *call-by-value-result*
- *call-by-name*
- *call-by-need*
- *call-by-name-with-thunks*

Let's examine these six methods in turn. After which, we will investigate variadic functions in Scam.

Call-by-value

This method is the only method of parameter passing allowed by C, Java, Scam, and Scheme. In this method, the formal parameters are set up as local variables that contain the value of the expressions that were passed as arguments to the function. Changes to local variables are not reflected in the actual arguments. For example, an attempt to define a function for exchanging the values of two variables passed to it might look like:

```
(define (swap a b)
    (define temp a)
    (set! a b)
    (set! b temp)
    )
```

Consider this code which uses *swap*:

```
(define x 3)
(define y 4)

(swap x y)
```

```
(inspect x)
(inspect y)
```

Under *call-by-value*, this function would not yield the intended semantics. The output of the above code is:

```
x is 3
y is 4
```

This is because only the values of the local variables a and b were swapped; the variables x and y remain unchanged as only their values were passed to the swapping function. In general, one cannot get a swap routine to work under *call-by-value* unless the addresses of the variables are somehow sent. One way of using addresses is to pass an array (in C and Scam, when an array name is used as an argument, the address of the first element is sent. In Java, the address of the array object is sent). For example, the code fragment:

```
(define x (array 1))
(define y (array 0))

(swap x y)  ;address of beginning element is sent

(println "x[0] is " (getElement x 0) " and y[0] is " (getElement y 0))
```

with *swap* defined as...

```
(define (swap a b)
    (define temp (getElement a 0))
    (setElement a 0 (getElement b 0))
    (setElement b 0 temp)
    )
```

would print out:

```
x[0] is 0 and y[0] is 1
```

In this case, the addresses of arrays x and y are stored in the local variables a and b, respectively. This is still call-by-value since even if the address stored in a, for example, is modified, x still "points" to the same array as before. Here is an example:

```
(define (change a)
   (assign a (array 13))
   )

(define x (array 42))

(inspect (getElement x 0))
```

yields:

```
(getElement x 0) is 42)
```

Note that C has an operator that extracts the address of a variable, the & operator. By using &, one can write a swap in C that does not depend on arrays:

```
void swap(int *a,int *b)
    {
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
    }
```

The call to *swap* would look like:

```
int x = 3;
int y = 4;

swap(&x,&y);

printf("x is %d and y is %d\n",x,y);
```

with output:

```
x is 4 and y is 3
```

as desired.

Note that this is still *call-by-value* since the *value* of the address of x (and y) is being passed to the swapping function.

Call-by-reference

This second method differs from the first in that changes to the formal parameters during execution of the function body are immediately reflected in actual arguments. Both C++ and Pascal allow for call-by-reference. Normally, this is accomplished, under the hood, by passing the address of the actual argument (assuming it has an address) rather than the value of the actual argument. References to the analogous formal parameter are translated to references to the memory location stored in the formal parameter. In C++, *swap* could be defined as:

```
void swap(int &a, int &b) // keyword & signifies
    {                     // call-by-reference
    int temp = a;
    a = b;
    b = temp;
    }
```

Now consider the code fragment:

```
var x = 3;  //assume x at memory location 1000
var y = 4;  //assume y at memory location 1008

//location 1000 holds a 3
//location 1008 holds a 4

swap(x,y);
cout << "x is " << x << " and y is " << y << "\n";
```

When the swapping function starts executing, the value 1000 is stored in the local variable *a* and 1008 is stored in local variable *b*. The line:

```
temp = a;
```

is translated, not into store the value of *a* (which is 1000) in variable *temp*, but rather store the value at memory location 1000 (which is 3) in variable *temp*. Similar translations are made for the remaining statements in the function body. Thus, the code fragment prints out:

```
x is 4 and y is 3
```

The swap works! When trying to figure out what happens under *call-by-reference*, it is often useful to draw pictures of the various variables and their values and locations, then update them as the function body executes.

One can simulate *call-by-reference* in Scam by delaying the arguments and capturing the calling environment. To obtain the calling environment in Scam, one simply adds a formal parameter with the name #. The calling environment is then passed silently to the function when a function call is made. This means that the # formal parameter is not matched to any actual argument and can appear anywhere in the parameter list (except after the variadic parameters @ and *$* - more on them later).

In addition to grabbing a handle to the calling environment, a swapping function also needs to delay the evaluation of the variables passed in. One delays the evaluation of an argument by naming the formal parameter matched to the argument in a special way. If the formal parameter name begins with a *$*, then the corresponding argument is delayed.

With the ability to grab the calling environment, delay the evaluation of arguments, and access the bindings in an environment (see the chapter on Objects), we can now define a *swap* function that works as intended.

```
(define (swap # $a $b)
    (define temp (get $a #))
    (set $a (get $b #) #)
    (set $b temp #)
    )
```

The *get* function takes a symbol and an environment and retrieves the value of the symbol as found in the given environment. The *set* function is analogous, but changes the value of a symbol found in the given environment. Both *get* and *set* are true functions, so the values of *$a* and *$b* are used in manipulating the given environment. Note that this version of *swap* only works if variable names (*i.e.* symbols) are passed to *swap*.

Call-by-value-result

This method is a combination of the first two. Execution of the function body proceeds as in *call*-by-value. That is, no updates of the actual arguments are made. However, after execution of the body, but just before the function returns, the actual arguments are updated with the final values of their associated formal parameters. This method of parameter passing is used for Ada *in-out* parameters.

We can achieve the affect of *call-by-value-result* in Scam by using a scheme similar to the simulation of *call-by-reference*. The modification is to assigning the values of the passed symbols to local variables at the start of the function:

```
(define (swap # $a $b)
    ; pre
    (define a (get $a #))
    (define b (get $b #))

    ; body
    (define temp a)
    (set! a b)
    (set! b temp)

    ;post
    (set $a a #)
    (set $b b #)
    )
```

The section marked `pre` sets up the locals while the section marked `body` implements the swap on the locals. The section marked `post` updates the variables that were passed to *swap*.

Usually, one cannot tell whether a language implements *call-by-reference* or *call-bay-value-result*; the resulting values are the same. One situation where the two methods of parameter passing can generate different results is if the body portion of the function references a global variable and that global variable is passed as an argument as well. This second reference to the global is known as an *alias*. Now there are two references to the same variable through two different names. Unlike *call-by-value-result*, updates to the alias in the body section are immediately reflected in the value of the global variable under *call-by-reference*.

Call-by-name

Call-by-name was used in Algol implementations. In essence, functions are treated as macros. Under *call-by-name*, the fragment:

```
(define (swap a b)
    (define temp a)
    (set! a b)
    (set! b temp)
    )

(define x 3)
(define y 4)

(swap x y)
```

```
(println "x is "  x " and y is " y)
```

...would be translated into:

```
(define (swap a b)
    (define temp a)
    (set! a b)
    (set! b temp)
    )

(define x 3)
(define y 4)

;substitute the body of the function for the call,
;renaming the references to formal parameters with the names of
;the actual args

(scope
    (define temp x)
    (assign x y)
    (assign y temp)
    )

(println "x is "  x " and y is " y)
```

Under *call-by-name*, the *swap* works as desired, so why is *call-by-name* a method that has fallen into relative disuse? One reason is complexity. What happens if a local parameter happens to have the same name as one of the actual args. Suppose *swap* had been written as:

```
(define (swap a b)
    (define x a)
    (set! a b)
    (set! b x)
    )
```

Then a naive substitution and renaming would have produced:

```
(scope
    (define x x)
    (assign x y)
    (assign y x)
    )
```

which is surely incorrect. Further problems occur if the body of the function references globals which have been shadowed in the calling function. This requires a complicated renaming scheme. Finally, *call-by-name* makes treating functions as first-class objects problematic (being difficult to recover the static environment of the called function). *Call-by-name* exists today in C++, where it is possible to *inline* function calls for performance reasons, and in macro processors.

Call-by-need

In *call-by-value*, the arguments in a function call are evaluated and the results are bound to the formal parameters of the function. In *call-by-need*, the arguments themselves are literally bound to the formal parameters, as in *call-by-name*. A major difference is that the calling environment is also bound to the formal parameters as well. This bundle of literal argument and evaluation environment is known as a *thunk*. The actual values of the arguments are determined only when such values are needed; when such a need occurs, the thunk is evaluated, causing the literal argument in the thunk to be evaluated in the stored (calling) environment. For example, consider this code:

```
(define (f x)
    (define y x)  ;x needed! x is fixed to its current value
    (set! z (* z 2))
    (+ x y)       ;x needed! x was already evaluated under call-by-need
    )
(define z 5)
(f (+ z 3))
```

Under *call-by-name*, the return value is 21, but under *call-by-need*, the return value is 16. This is because the value of z changed *after* the point when the value of x (really `(+ z 3)` was needed and the value of x was fixed from the prior evaluation of x. Under *call-by-name*, the second reference to x causes a fresh, new evaluation of z, the yielding the result of 21.

Call-by-need is exactly the method used to implement streams in the textbook *The Structure and Interpretation of Computer Programs*. It is important to remember that the evaluation of a *call-by-need* argument is done only once, with the result stored for future requests.

One can simulate *call-by-need* in Scam by delaying the evaluation of the arguments and capturing the calling environment. The combination of delayed argument and captured environment comprise a thunk in Scam. To memoize the thunk, we assign the evaluation of the delayed argument to a local variable. Here is a Scam's version of function *f*:

```
(define (f # $x)
    (define x (eval $x #))
    (set! z (* z 2))
    (+ x y)
    )
```

The built-in *eval* function is used to evaluate the thunk comprised of *$x* and #.

Call-by-name-with-thunks

Call-by-name-with-thunks has roughly the same semantics as *call-by-name*, the difference being that thunks are used instead of textual substitutions. in the body of the function. *Call-by-name-with-thunks* differs from *call-by-need* in that the result of evaluating a thunk is not memoized (*i.e.* stored for future retrieval). Instead, repeated calls for the the value of the formal parameter result in the expression in the thunk being evaluated again and again. Differences between *call-by-need* and *call-by-name-with-thunks* arise when the thunk's expression causes a change of state.

```
function f(x)
    {
```

```
    var y = x;    //x needed! Evaluate the arg in the calling env
    z = z * 2;
    return x + y; //x needed! Evaluate the arg in the calling env
    }

var z = 5;
f(z + 3);
```

Like *call-by-name*, this function call also returns 21. One can simulate *call-by-name-with-thunks* in Scam easily:

```
(define (f # $x)
    (define y (eval $x #))
    (set! z (* z 2))
    (+ (eval $x #) y)
    )
```

Here, any time the value of the delayed argument is needed, a fresh evaluation is invoked.

Variadic functions

A variadic function is a function that can take a different number of arguments from call to call. Scam allows this via two special formal parameter names. They are *@* and *$*. If the last formal parameter is *@*, then all remaining (evaluated) arguments not matched to any previous formal parameters are gathered up in a list and the variable *@* is bound to this list. For example, consider these definitions:

```
(define (variadic first @)
    (println "the first argument is " first)
    (println "the remaining arguments are:")
    (while (valid? @)
        (println "    " (car @))
        (set! @ (cdr @))
        )
    )
(define x 1)
(define y 2)
```

The call `(variadic x)` produces:

```
the first argument is 1
the remaining arguments are:
```

while the call `(variadic x y (+ x y))` produces:

```
the first argument is 1
the remaining arguments are:
     2
     3
```

Similar to *@*, the formal parameter *$* is expected to be the last formal parameter. The difference is that the arguments bundled up into a list are delayed. Suppose one replaced all occurrences of *@* with *$* in the definition of *variadic*. Then, the call `(variadic x y (+ x y))` would produce:

```
the first argument is x
the remaining arguments are:
    y
    (+ x y)
```

Chapter 16

Overriding Functions

Suppose one wishes to count how many additions are performed when code in a module is executed. One way to do this is to override the built-in addtion function:

```
(define plus-count 0)
(define (+ a b)
    (assign plus-count (+ plus-count 1))
    (+ a b)
    )
```

The problem here is that the original binding of + is lost when the new version is defined. Calling + now will result in an infinite recursive loop.

One solution is to save the original binding of + before the new version is defined:

```
(define old+ +)
(define (+ a b)
    (assign plus-count (old+ plus-count 1))
    (old+ a b)
    )
```

With the original version of + bound to *old+*, now *a* and *b* can be added together properly.

Scam automates this process with two functions, *redefine* and *prior*. If the new version of a function is "redefined" rather than defined, the previous binding of the function is saved in the function closure that results. The *prior* function is then used to retrieve this binding. Here is a rewrite of the above code using these functions:

```
(include "reflection.lib")

(redefine (+ a b)
    (assign plus-count ((prior) plus-count 1))
    ((prior) a b)
    )
```

The *redefine* and *prior* functions can be accessed by including *reflection.lib*.

Implementing *redefine* and *prior*

Recall that closures are objects in Scam. The *redefine* function works by adding a field to the closure object generated by a function definition. It begins by delaying evaluation of the paramater list and the function body and then extracting the function name from the parameter list.

```
(define (redefine # $params $)
    ;obtain the function name
    (define f-name (car $params))
    ;find the previous binding of the function name
    ;if no prior binding, use the variadic identity function
    (if (defined? f-name #)
        (define f-prior (get f-name #))
        (define f-prior (lambda (@) @))
        )
    ;now generate the function closure
    (define f (eval (cons 'define (cons $params $)) #))
    ;add the previous binding to the closure
    (addSymbol '__prior f-prior f)
    f
    )
```

It continues by looking up the function name in the calling scope, #, binding that value to the symbol *f-prior*. If no binding exists, an identity function is bound to *f-prior*. Next, the desired function definition is processed by building the code for a function definition from the delayed parameter list and the delayed body. Finally, a new field is added to the function closure and bound to the prior function.

The *prior* function then looks for the added symbol and returns its binding. It does so by extracting the constructor of the calling environment and then retrieving the value of the symbol that was added by *redefine*:

```
(define (prior #)
    (define f (# '__constructor))
    (f '__prior)
    )
```

Cloning functions

If you override a function defined in an enclosing scope, only calls to the overridden function *in the current scope* see the new definition. Calls made in the enclosing scope to the overridden function see the old version. A scope violation would occur otherwise. To solve this problem, one can override the offending function or more simply, clone it. Cloning a function creates a new definition in the current scope. The only difference is the definition environment is changed to the current environment, the function parameter list and the body remain unchanged.

To clone a function, one calls the clone function, passing in the function to be cloned. Consider this code:

```
(include "reflection.lib")

(define (f x) x)
(inspect this)
(inspect (f '__context))
```

```
(scope
    (inspect (local? 'f this))
    (define f (clone f))
    (inspect (local? 'f this))
    (inspect this)
    (inspect (f '__context))
    )
```

The output generated will be something like:

```
this is <object 11698>
(f '__context) is <object 11698>
(local? (quote f) this) is #f
(local? (quote f) this) is #t
this is <object 13317>
(f '__context) is <object 13317>
```

The first two calls to *inspect* show that *f*s definition environment is the outer scope. The next call to *inspect* shows that *f* is not defined in the inner scope. The last three calls show that *f* is now defined with the proper definition environment.

For an example of using clone, see the chapter on Objects.

Chapter 17

Concurrency

Scam provides for concurrency using lightweight threads. The following subsections describes the built-in concurrency and concurrency control functions and give details on their use.

Built-in support for threads

The following functions can be used to control threads and their interactions:

thread This one-argument function takes in an expression and evaluates in parallel to the calling thread and returns the thread id of the new thread. The code below:

```
(thread (println "Hello World 1"))
(thread (println "Hello World 2"))
(thread (println "Hello World 3"))
```

yields something similar to the following output:

```
HellHelo Worllod  3
World 1
Hello World 2
```

The problem here is that threads are executing in parallel. This means that when you print out from one thread to the console another thread could be as well; and you get the overlap seen above. To get around you can either use the built in semaphore or the function *displayAtomic*.

gettid This no-argument function returns the thread ID of the current thread. The code below:

```
(thread (println (gettid)))
```

yields something similar to the following output:

```
2
```

lock This no-argument function acquires the built-in semaphore.

unlock This no-argument function releases the built-in semaphore. The code below demonstrates both *lock* and *unlock*:

```
(thread (begin (lock) (println "Hello World 1") (unlock)))
(thread (begin (lock) (println "Hello World 2") (unlock)))
(thread (begin (lock) (println "Hello World 3") (unlock)))
```

yields something similar to the following output:

```
Hello World 2
Hello World 3
Hello World 1
```

Note: these three threads will be executed in parallel, but if you do not join on the threads using *tjoin* then the main process may terminate before the threads terminate. Joining on the threads can be accomplished by saving the thread id's of each thread, and then calling *tjoin* on the thread id's, as in the following example:

```
(define t1 (thread (begin (lock) (println "Hello World 1") (unlock))))
(define t2 (thread (begin (lock) (println "Hello World 2") (unlock))))
(define t3 (thread (begin (lock) (println "Hello World 3") (unlock))))

(tjoin t1)
(tjoin t2)
(tjoin t3)
```

tjoin This one-argument function causes the current thread to wait until a particular thread terminates. If the desired thread has already terminated, the function immediately returns. The desired thread is specified by passing its thread ID to the function. The code below:

```
(define firstTID (thread (println (fib 25))))
(tjoin firstTID)
(thread (println (fib 10)))
```

yields something similar to the following output:

```
75025
55
```

Note that the thread which is evaluating fibonacci of 10 must wait until the thread which is evaluating fibonacci of 25 has finished.

displayAtomic This variadic function is similar to *display*; however, it ensures that there will be no overlap of output if the user only uses *displayAtomic* for printing. The code below:

```
(thread (displayAtomic "Hello World 1\n"))
(thread (displayAtomic "Hello World 2\n"))
(thread (displayAtomic "Hello World 3\n"))
```

yields something similar to the following output:

```
Hello World 2
Hello World 1
Hello World 3
```

Thread pools

In order to avoid the sometimes inevitable system-level restrictions on the maximum number of lightweight threads allowed per process, Scam supports the creation of thread pools. A thread pool pre-allocates a fixed number of threads, and maintains a queue of expressions. Expressions may be pushed onto the queue, and the thread pool will automatically pop the expressions off in order to be executed concurrently. The code below illustrates the use of a pool:

```
(define pool (tpool 10))
((pool 'push) (fib 10))
((pool 'push) (fib 11))
((pool 'push) (fib 12))
((pool 'shutdown))
```

Note: You **must** call the *shutdown* method of your thread pool. If you fail to do this then the threads in the pool may not finish before the main process terminates.

The following constructors and methods are associated with pools:

tpool This one-argument constructor creates a new thread pool object. The single argument specifies the number of concurrent threads allowed.

push This variadic function takes in an expression and an optional number of call-back functions. The call-back functions are called with the result of the evaluated expression.

```
...
((pool 'push) (fib 12) println)
...
```

Would result in calling the function *'println'*, passing the return value of *'fib'*.

push* This variadic function takes in an expression, a environment, and an optional number of expressions. The required expression is evaluated under the given environment. The optional expressions are then evaluated with the return value of the evaluated required expression.

empty? This no-argument function returns true if there are no expressions in the work queue or in the running queue, otherwise it returns false.

join This no-argument function turns off acceptance of new expressions until the running queue is empty.

shutdown This no-argument function waits for the active threads to finish before letting the thread pool expire.

Parallel execution of expressions

For compatibility with MIT Scheme, expressions in Scam can be evaluated in parallel with the variadic function *pexecute*:

```
(pexecute expr1 expr2 .... exprN)
```

which is equivalent to the following:

```
; store the thread id's
(define tids nil)
(begin
    (set! tids (cons (thread expr1) tids))
    (set! tids (cons (thread expr2) tids))
    ...
    (set! tids (cons (thread exprN) tids))
    ; join on the threads
    (while (!= tids nil)
       (tjoin (car tids))
       (set! tids (cdr tids))
       )
    )
```

Each of the expressions will execute in parallel in separate processes. The expressions passed to *pexecute* are calls to no argument functions, or lambdas.

```
(pexecute f g (lambda () ...))
```

In the above example, the functions *f* and *g* and the body of the lambda will all be executed in parallel.

Another function, *pexecute**, is similar to *pexecute*, except that it serializes each expression, in the order given. The call:

```
(pexecute* expr1 expr2 .... exprN)
```

is equivalent to the following:

```
(begin
    (expr1)
    (expr2)
    ...
    (exprN)
    )
```

The *pexecute** function is useful for debugging concurrency problems.

Debugging concurrency problems

Locking and unlocking of threads can be difficult to debug. For this reason, Scam has a built-in function for determining which thread has the current lock. This function, *debugSemaphore* enables and disables the debugging mechanism.

```
(debugSemaphore #t)
(debugSemaphore #f)
```

The first call turns debugging on while the second turns debugging off. When on, attempts to acquire the semaphore produce output (on stderr) of the form:

```
thread XXXX is acquiring...
```

where XXXX is replaced by the process id of the acquiring process. If the semaphore is actually acquired, debugging emits:

```
thread XXXX has acquired.
```

On the releasing side, debugging emits messages of the form:

```
thread XXXX is releasing...
thread XXXX has released.
```

When a process executing in parallel throws an exception, *pexecute* will produce an error message similar to:

```
file philosophers.scm,line 356: parallel execution of thread 3 failed
try using pexecute* for more information
```

If a thread terminates with an exception, calling *pexecute** may reveal the exception that caused the failure. The *pexecture** call simulates concurrency, but actually runs the given expressions sequentially in the parent process.

Chapter 18

The Main Library

Scam's main library contains a mish-mash of useful functions. It is automatically included.

Assignment-type functions

`(+= var value)`

The *+=* function is used to add the given value to the given variable. For example, suppose the variable *x* has the value 10. After evaluating the expression `(+= x 3)`, the value of *x* would be 13. The expression `(+= x n)`, is equivalent to the expression `(set` x (+ x n))!. The functions -=, *=, and /= are also defined with analogous semantics.

`(++ var)`

The *+=* function increments the given variable. For example, suppose the variable *x* has the value 10. After evaluating the expression `(++ x)`, the value of *x* would be 11. The expression `(+= x n)`, is equivalent to the expression `(set` x (+ x n))!. The functions -=, *=, and /= are also defined with analogous semantics.

Mathematical Functions

`randomMax()`

```
(define __builtin __context)
(define __main-lib this)
(define nil? null?)
(define (valid? x) (not (null? x)))
(define (head x) (car x))
(define (tail x) (cdr x))
(define (join x y) (cons x y))
(define ^ expt)
(define ** expt)
(define (set! # $x y @)
    (define env (if (null? @) # (car @)))
    (cond
        ((dot? $x)
            (define last (dot-assign-setup (eval (cadr $x) #) (cddr $x) #))
            ;(inspect last)
            (set (cadr last) y (car last))
```

```
            )
        (else
            (set $x y env)
            )
        )
    )
(define assign set!)

(define (for # init $test $increment $)
    (while (eval $test #)
        (evalList $ #)
        (eval $increment #)
        )
    )

(define (for-each2 # $indexVar items $)
    (define result #f)
    (while (!= items nil)
        (set $indexVar (car items) #)
        (set 'result (evalList $ #))
        (set 'items (cdr items))
        )
    result
    )

(define (for-each f x)
    (define (iter items)
        (cond
            ((null? items) nil)
            (else (f (car items)) (iter (cdr items)))
            )
        )
    (iter x)
    )

(define (+= # $v value) (set $v (+ (eval $v #) value) #))
(define (-= # $v value) (set $v (- (eval $v #) value) #))
(define (*= # $v value) (set $v (* (eval $v #) value) #))
(define (/= # $v value) (set $v (/ (eval $v #) value) #))
(define (++ # $v) (set $v (+ (eval $v #) 1) #))
(define (-- # $v) (set $v (- (eval $v #) 1) #))

; object-related functions

(define __type type)

(define (type x)
    (if (and (eq? (__type x) 'CONS) (eq? (car x) 'object))
        (get '__label x)
        (__type x)
        )
    )

(define (class x) (get '__label x))
```

```
(define (dot-assign-setup obj fields env)
    (while (valid? (cdr fields))
        (define field (car fields))
        (inspect field)
        (if (pair? field)
            (set 'obj (get (eval field env) obj))
            (set 'obj (get field obj))
            )
        (set! fields (cdr fields))
        )
    (if (pair? (car fields))
        (list obj (eval (car fields) env))
        (cons obj fields)
        )
    )
(define (dot obj $)
    ;(inspect obj)
    (while (valid? $)
        ;(inspect (car $))
        (set! obj (get (car $) obj))
        (set! $ (cdr $))
        )
    ;(println "leaving dot")
    ;(inspect obj)
    obj
    )

(define (is? x y)
    (cond
        ((null? x) #f)
        ((not (environment? x)) (eq? (type x) y))
        ((and (environment? x) (or (eq? y 'environment) (eq? y 'object))) #t)
        ((and (valid? (dot x __constructor)) (eq? (dot x __constructor name) y)) #t)
        (else (and (local? 'parent x) (is? (dot x parent) y)))
        )
    )

(define (object? x) (and (pair? x) (eq? (car x) 'object)))
(define (closure? x) (and (object? x) (eq? (class x) 'closure)))
(define (error? x) (and (object? x) (eq? (class x) 'error)))
(define (environment? x) (and (object? x) (eq? (class x) 'environment)))


(define (and # $)
    (define (iter items)
        (cond
           ((null? items) #t)
           ((eval (car items) #) (iter (cdr items)))
           (else #f)
           )
        )
    (iter $)
    )
```

```
(define (or # $)
    (define (iter items)
        (cond
           ((null? items) #f)
           ((eval (car items) #) #t)
           (else (iter (cdr items)))
           )
        )
    (iter $)
    )
(define (dec x) (- x 1))
(define (inc x) (+ x 1))

(define __display display)
(define (print @)
    (while (valid? @)
        (display (car @))
        (set! @ (cdr @))
        )
    'print-done
    )

(define (println @)
    (apply print @)
    (print "\n")
    )

(define (let # $inits $)
    (define v nil)
    (define e (scope this))
    (set '__context # e)
    (for-each2 v $inits
        (addSymbol (car v) (eval (car (cdr v)) #) e)
        ;(println "adding " (car v) " from " $inits)
        ;(println "    its value is " (eval (car (cdr v)) #))
        ;(inspect e)
        )
    (evalList $ e)
    )

(define (let* # $inits $)
    (define v nil)
    (define e (scope this))
    (set '__context # e)
    (for-each2 v $inits
        (addSymbol (car v) (eval (car (cdr v)) e) e)
        )
    (evalList $ e)
    )

(define (evalList items env)
    (while (valid? (cdr items)) ; for tail recursion
         (eval (car items) env)
         (set 'items (cdr items))
```

```
         )
    (eval (car items) env)
    )

(define (negative? n) (< n 0))
(define (positive? n) (> n 0))

(define (newline) (println))
(define remainder %)
(define (append a b)
    (cond
        ((null? a) b)
        (else (cons (car a) (append (cdr a) b)))
        )
    )
(define (last-pair x)
    (cond
        ((null? x) nil)
        ((null? (cdr x)) x)
        (else (last-pair (cdr x)))
        )
    )
(define (reverse x)
    (define (iter store rest)
        (cond
            ((null? rest) store)
            (else (iter (cons (car rest) store) (cdr rest)))
            )
        )
    (iter nil x)
    )

(define (map op @)
    (define (map1 items)
        (cond
            ((null? items) nil)
            (else (cons (op (car items)) (map1 (cdr items))))
            )
        )
    (define (iter items)
        (cond
            ((null? (car items)) nil)
            (else (cons (apply op (map car items)) (iter (map cdr items))))
            )
        )
    (cond
        ((= (length @) 1) (map1 (car @)))
        (else (iter @))
        )
    )
(define (abs x) (if (< x 0) (- x) x))
(define (even? n) (= (% n 2) 0))
(define (odd? n) (= (% n 2) 1))
(define (integer? x) (eq? (type x) 'INTEGER))
```

```
(define (real? x) (eq? (type x) 'REAL))
(define (number? x) (or (integer? x) (real? x)))
(define (string? x) (eq? (type x) 'STRING))
(define (symbol? x) (eq? (type x) 'SYMBOL))
(define (array? x) (eq? (type x) 'ARRAY))
(define (true? x) x)
(define (false? x) (not x))
(define (dot? x) (and (pair? x) (member? (car x) '(dot .))))
(define (literal? x)
    (or (null? x) (eq? x #t) (eq? x #f) (string? x) (array? x)
        (integer? x) (real? x) (and (pair? x) (eq? (car x) 'quote))))
(define (atom? x) (not (or (pair? x) (string? x) (array? x))))
(define (car-cdr items @)
    (while (valid? @)
        (cond
            ((= (car @) 0) (set 'items (car items)))
            (else (set 'items (cdr items)))
            )
        (set '@ (cdr @))
        )
    items
    )

(define (caar x) (car-cdr x 0 0))
(define (cadr x) (car-cdr x 1 0))
(define (caddr x) (car-cdr x 1 1 0))
(define (cadddr x) (car-cdr x 1 1 1 0))
(define (caddddr x) (car-cdr x 1 1 1 1 0))
(define (cadddddr x) (car-cdr x 1 1 1 1 1 0))

(define (cddr x) (cdr (cdr x)))
(define (cddr x) (car-cdr x 1 1))
(define (cdddr x) (car-cdr x 1 1 1))
(define (cddddr x) (car-cdr x 1 1 1 1))
(define (cdddddr x) (car-cdr x 1 1 1 1 1))

(define (equal? a b)
    (cond
        ((null? a)
            ;(println "returning" (null? b))
            (null? b))
        ((pair? a)
            ;(println "pair returning "
            ;(and (pair? b) (equal? (car a) (car b)) (equal? (cdr a) (cdr b))))
            (and (pair? b) (equal? (car a) (car b)) (equal? (cdr a) (cdr b))))
        ((string? a)
            (string-equal? a b))
        ((array? a)
            (array-equal? a b))
        (else
            ;(println "else returning "(eq? a b))
            (eq? a b))
        )
    )
```

```
(define (array-equal? a b)
    (cond
        ((null? a) (null? b))
        ((null? b) #f)
        (else (and (equal? (car a) (car b)) (array-equal? (cdr a) (cdr b))))
        )
    )

(define (string-compare a b)
    (cond
        ((and (null? a) (null? b)) 0)
        ((null? a) (- 0 (ascii (getElement b 0))))
        ((null? b) (ascii (getElement a 0)))
        (else
            (if (== (ascii (getElement a 0)) (ascii (getElement b 0)))
                (string-compare (cdr a) (cdr b))
                (- (ascii (getElement a 0)) (ascii (getElement b 0)))
                )
            )
        )
    )

(define (sqrt x) (expt x 0.5))

(define (cons-stream # a $b)
    (cons a (lambda () (eval $b #)))
    )

(define (stream-car s) (car s))
(define (stream-cdr s) ((cdr s)))
(define (stream-null? s) (null? s))

(define (code $s) $s)

(define (member? x items)
    (valid? (member x items))
    )

(define (member x items)
    (cond
        ((null? items)
            nil
            )
        ((eq? x (car items))
            items
            )
        (else
            (member x (cdr items))
            )
        )
    )

(define (nequal? a b) (not (equal? a b)))
```

```
(define (getElement items @)
    (define __getElement (get 'getElement (get '__context __context)))
    (while (valid? @)
        ;(inspect items)
        (set 'items (__getElement items (car @)))
        (set '@ (cdr @))
        )
    items
    )

(define __string+ string+)
(define (string+ str @)
    (while (valid? @)
        (set 'str (__string+ str (string (car @))))
        (set '@ (cdr @))
        )
    str
    )

(define __string string)
(define (string x)
    (cond
        ((pair? x) (list-to-string x))
        (else (__string x))
        )
    )
(define (list-to-string x)
    (define (iter z)
        (cond
            ((null? (cdr z))
                (string+ (string (car z)) ")")
                )
            ((pair? (cdr z))
                (string+ (string (car z)) " " (iter (cdr z)))
                )
            (else
                (string+ (string (car z)) " . " (string (cdr z)) ")")
                )
            )
        )
    (cond
        ((thunk? x) (string+ "<thunk " (address x) ">"))
        ((closure? x) (string+ "<closure " (address x) ">"))
        ((error? x) (string+ "<error " (address x) ">"))
        ((environment? x) (string+ "<environment " (address x) ">"))
        ((object? x) (string+ "<object " (address x) ">"))
        (else (string+ "(" (iter x)))
        )
    )

(define (thunk code environment) this)
(define (thunk? item) (is? item 'thunk))
(define (force item) (eval (dot item code) (dot item environment)))
```

```
(define . dot)

(define (assoc x y)
    (cond
        ((null? y) #f)
        ((equal? x (caar y)) (car y))
        (else (assoc x (cdr y)))
        )
    )

(define (make-assoc xs ys)
    (cond
        ((null? xs) nil)
        (else (cons (list (car xs) (car ys)) (make-assoc (cdr xs) (cdr ys))))
        )
    )

(define (local? id env)
    (member? id (localNames env))
    )
(define (localNames env)
    (cadr env)
    )
(define (localValues env)
    (caddr env)
    )
(define (defined? id env)
    (not (error? (catch (eval id env))))
    )
```

Chapter 19

Randomness

Scam has a few functions for generating pseudo-random numbers. Although the word *randomly* is used in the chapter, *pseudo-randomly* would be a more accurate term.

Built-in Random Functions

The built-in functions are *randomInt*, *randomMax*, and *randomSeed*.

`randomMax()`

This function, which takes no arguments, returns the largest integer that can be returned from the *randomInt* function.

`randomInt()`

This function, which takes no arguments, returns a randomly generated number between 0 and `(randomMax)`, inclusive.

`randomSeed(seed)`

This function, which takes a positive integer as an argument, resets the state of Scam's pseudo-random number generator. If a program uses *randomInt* without resetting the random state, the same sequence of pseudo-random numbers will be generated each time the program is run. To generate a different sequence each time, a common trick is to set the state with the current time at the start of the program:

```
(randomSeed (integer (time)))
```

Note that `(integer (time))` changes rather slowly, as computers go, since it marks the number of seconds since the beginning of the epoch. So setting the seed once at the beginning of a program is probably OK; setting the seed multiple times within a program may be problematic.

The *random* library

The random library, included with the expression:

```
(include "random.lib")
```

adds the following functions: *randomReal*, *randomRange*, *shuffle*, and *flip*.

`randomReal()`

This function, which takes no arguments, returns a real number between 0 (inclusive) and 1 (exclusive).

`randomRange(low,high)`

This function returns an integer between *low* (inclusive) and *high* (exclusive). To randomly retrieve a value from a collection named *items*, one might use the expression:

```
(getElement items (randomRange 0 (length items)))
```

`shuffle(items)`

This function, which takes a list or an array as an argument, randomly rearranges the values in the given collection.

`flip()`

This function, which takes no arguments, randomly returns integer 0 or integer 1. It is used to simulate coin flips.

Index