5 Locally Defined Procedures

5.1 Overview

When we bind a variable to some value using `define`, we are able to use that variable to represent the value to which it is bound either directly in response to a Scheme prompt or within a program that we are writing. Does this mean that we have to think of new names for every variable we use when we write many programs? No. Scheme gives us a mechanism for limiting where bindings are in effect. In this chapter, we look at ways of binding variables so that the binding holds only within a program or part of a program. The main tools for doing this are two special forms with keywords `let` and `letrec`. After introducing them, we use them to implement polynomials as a data type in Scheme. We then apply the polynomial methods we develop to a discussion of binary numbers, which form the basis of machine computation.

5.2 Let and Letrec

You may have wondered how Scheme knows what value to associate with various occurrences of a variable. When some value is assigned to a variable, we may think of that information being stored in a table with two columns: the left one for variable names and the right one for the associated values. Such a table is called an *environment*. A number of variables (such as those bound to procedures) like `+`, `*`, `car`, and `cons` are predefined. These definitions are kept in a table which we call the *initial global environment*. This initial global environment is in place whenever you start up Scheme. When a given variable is encountered in an expression, Scheme looks through its environment to see

if the variable has been bound to a value. Naturally, the variable **+** is bound to the arithmetic operation we usually associate with the addition procedure, and so on.

In addition to having the predefined Scheme variables, we have seen how to use **define** to bind a variable to a desired value. The expression (**define** *var val*) binds the variable *var* to the value *val*. We can again think of the variables we define ourselves as being placed in a table which we call the *user global environment* and when a variable is encountered in an expression, the *global environment* (which includes both the user and initial global environments) is scanned to see if that variable is bound to a value. If a binding cannot be found, a message is written saying that the variable is unbound in the current environment. The user global environment remains in effect until the user exits from Scheme.

Variables are also used as parameters to procedures that are defined by a lambda expression. For example, in the lambda expression

```
(lambda (x y) (+ x y))
```

the variables **x** and **y** occurring in the body **(+ x y)** of the lambda expression are *locally bound* (or *lambda bound*) in the expression **(+ x y)** since the **x** and **y** occur in the list of parameters of that lambda expression. If we apply the procedure, which is the value of this lambda expression, to the arguments 2 and 3, as in

```
((lambda (x y) (+ x y)) 2 3)
```

we can think of a new table being made, called a *local environment*, which is associated with this procedure call. In this local environment, **x** is locally bound to 2 and **y** is locally bound to 3. Then substituting 2 for **x** and 3 for **y** gives **(+ x y)** the value 5, and

```
((lambda (x y) (+ x y)) 2 3)
```

returns the value 5.

A variable occurring in a lambda expression that is not lambda bound by that expression is called *free* in that expression. If we consider the expression

```
(lambda (f y) (f a (f y z)))
```

the variables **f** and **y** are lambda bound in the expression, and the variables **a** and **z** are free in the expression. When the application

```
((lambda (f y) (f a (f y z))) cons 3)
```

is evaluated, the operator (which is the lambda expression) and its two operands are first evaluated. When the lambda expression is evaluated, bindings are found for the free variables in a nonlocal environment. Then, with these bindings for the free variables, the body of the lambda expression is evaluated with **f** bound to the procedure, which is the value of **cons**, and **y** bound to 3. If either of the free variables is not bound in a nonlocal environment, a message to that effect appears when the application is made. On the other hand, if **a** is bound to 1 and **z** is bound to **(4)** in a nonlocal environment, then this application evaluates to **(1 3 4)**.

We used the term *nonlocal* environment in the previous paragraph when we referred to the bindings of the free variables in the body of a lambda expression. Those bindings may be found in the global environment or in a local environment for another lambda expression. This is illustrated by the following example:

```
((lambda (x)
   ((lambda (y)
      (- x y))
    15))
 20)
```

The variable **x** is free in the body of the inner lambda expression, but its binding is found in the local environment for the outer lambda expression. The value of the expression is 5.

In the example

```
(lambda (x y) (+ x y))
```

the local bindings hold only in the body **(+ x y)** of the lambda expression, and when we leave the body, we can for the moment think of the local environment as being discarded. The expression **(+ x y)** is said to be in the scope of the variable **x** (and also of **y**). In general, an expression is said to be in the *scope* of a variable **x** if that expression is in the body of a lambda expression in which **x** occurs in the list of parameters.

By looking at a Scheme program, one can tell whether a given expression is in the body of some lambda expression and determine whether the variables in that expression are lambda bound. A language in which the scope of the variables can be determined by looking only at the programs is called *lexically scoped.* Scheme is such a language.

Scheme provides several other ways of making these local bindings for variables, although we shall later see that these are all ultimately related to lambda bindings. The two that we discuss here are let expressions and letrec expressions. To bind the variable *var* to the value of an expression *val* in the expression *body*, we use a let expression (which is a special form with keyword `let`) with the syntax:

(let ((*var* *val*)) *body*)

To make several such local bindings in the expression *body*, say var_1 is to be bound to val_1, var_2 to val_2, ..., var_n to val_n, we write

(let ((var_1 val_1) (var_2 val_2) ...(var_n val_n)) *body*)

The scope of each of the variables var_1, var_2, ..., var_n is only *body* within the let expression. For example, the expression

```
(let ((a 2) (b 3))
  (+ a b))
```

returns 5. Here `a` is bound to 2 and `b` is bound to 3 when the body `(+ a b)` is evaluated. Another example is

```
(let ((a +) (b 3))
  (a 2 b))
```

returns 5, since `a` is bound to the procedure associated with `+` and `b` is bound to 3. Similarly, in the expression

```
(let ((add2 (lambda (x) (+ x 2)))
      (b (* 3 (/ 2 12))))
  (/ b (add2 b)))
```

the variable `add2` is bound to the procedure to which `(lambda (x) (+ x 2))` evaluates, which increases its argument by 2, and `b` is bound to 0.5, and the whole expression returns 0.2.

The local binding always takes precedence over the global or other nonlocal bindings, as illustrated by the following sample computation:

```
[1] (define a 5)
[2] (add1 a)
6
[3] (let ((a 3))
      (add1 a))
4
[4] (add1 a)
6
```

```
[5] (let ((a 5))
      (begin
        (writeln (add1 a))
        (let ((a 3))
          (writeln (add1 a)))
        (add1 a)))
6
4
6
```

The **define** expression makes a binding of a to 5. When a is encountered in (add1 a) in [2], its value is found in the global environment and 6 is returned. In [3], a is locally bound to 3, and the expression (add1 a) is evaluated with this local binding to give the value 4. The scope of the variable a in the let expression is only the body of the let expression. Thus in [4], the value of the variable a in (add1 a) is again found in the global environment, where a is bound to 5, so the value returned for (add1 a) is 6. In [5], we see a version of the same computation in which no global bindings of a are made, but here the local binding takes precedence over the nonlocal bindings.

We get a better understanding of the meaning of the let expression

```
(let ((a 2) (b 3))
  (+ a b))
```

when we realize that it is equivalent to an application of a lambda expression:

```
((lambda (a b) (+ a b)) 2 3)
```

To evaluate this application, we first bind a to 2 and b to 3 in a local environment and then evaluate (+ a b) in this local environment to get 5.

In general, the let expression

(let ((var_1 val_1) (var_2 val_2) ... (var_n val_n)) *body*)

is equivalent to the following application of a lambda expression:

((lambda (var_1 var_2 ... var_n) *body*) val_1 val_2 ... val_n)

From this representation, we see that any free variable appearing in the operands val_1, val_2, ..., val_n is looked up in a nonlocal environment. For example, let's consider

```
[1] (define a 10)
[2] (define b 2)
[3] (let ((a (+ a 5)))
      (* a b))
30
```

```
[4] (let ((a 10) (b 2))
      (let ((a (+ a 5)))
        (* a b)))
30
```

In this example, a is bound globally to 10 in [1], and b is bound globally to 2 in [2]. Then in [3], the expression (+ a 5) is first evaluated.[1] The variable a is free in the expression (+ a 5), so the value to which a is bound must be looked up in the nonlocal (here global) environment. There we find that a is bound to 10, so (+ a 5) is 15. The next step is to make a local environment where a is bound to 15. We are now ready to evaluate the body of the let expression (* a b). We first try to look up the values of a and b in the local environment. We find that a is locally bound to 15, but b is not found there. We must then look in the nonlocal (here global) environment, and there we find that b is bound to 2. With these values, (* a b) is 30, so the let expression has the value 30. In [4], we see a similar program in which the free variables are looked up in a nonlocal but not global environment. Looking back at the let expressions, we see how the lexical scoping helps us decide which environment (local or nonlocal) to use to look up each variable.

It is important to keep track of which environment to use in evaluating an expression, for if we do not do so, we might be surprised by the results. Here is an interesting example:

```
[1] (define addb
      (let ((b 100))
        (lambda (x)
          (+ x b))))
[2] (let ((b 10))
      (addb 25))
125
```

Because b is bound to 10 in [2] and (addb 25) is the body of the let expression with this local environment, one might be tempted to say that the answer in [2] should have been 35 instead of 125. In [1], however, the lambda expression falls within the scope of the let expression in which b is bound to

[1] The symbol + is also free in (+ a 5), and its value is found in the initial global environment to be the addition operator. The number 5 evaluates to itself. Similarly, the symbol * is free in the body, and its value is found in the initial global environment to be the multiplication operator.

100. This is the binding that is "remembered" by the lambda expression, and when it is later applied to the argument 25, the binding of 100 to b is used and the answer is 125.

Let's look at [1] again. The variable addb is bound to the value of the lambda expression, thereby defining addb to be a procedure. The value of this lambda expression must keep track of three things as it "waits" to be applied: (1) the list of parameters, which is (x), (2) the body of the lambda expression, which is (+ x b), and (3) the nonlocal environment in which the free variable b is bound, which is the environment created by the let expression in which b is bound to 100. The value of a lambda expression is a procedure (also called a *closure*), which consists of the three parts just described. In general, the value of any lambda expression is a procedure (or closure) that consists of (1) the list of parameters (which follows the keyword lambda), (2) the body of the lambda expression, and (3) the environment in which the free variables in the body are bound at the time the lambda expression is evaluated. When the procedure is applied, its parameters are bound to its arguments, and the body is evaluated, with the free variables looked up in the environment stored in the closure. Thus in [2], (addb 25) produces the value 125 because the addb is bound to the procedure in which b is bound to 100.

Consider the following nested let expressions:

```
(let ((b 2))
  (let ((add2 (lambda (x) (+ x b)))
        (b 0.5))
    (/ b (add2 b))))
```

The first let expression sets up a local environment that we call Environment 1 (Figure 5.1).

b ⟶ [2]

Figure 5.1 Environment 1

The inner let expression sets up another local environment, which we call Environment 2. The first entry in this environment is add2, which is bound to the value of (lambda (x) (+ x b)). The x in (+ x b) is lambda bound in that lambda expression, and the value of b can be found in Environment 1.

But the inner let expression is in the body of the first let expression, so Environment 1 is in effect and we find that the value associated with b in Environment 1 is 2. Thus we have Environment 2 (Figure 5.2).

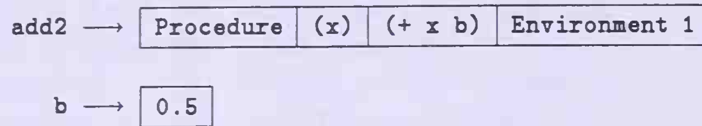

Figure 5.2 Environment 2

All of the variables in the expression to which add2 is bound are either bound in that expression itself (as was x) or are bound outside of the let expression (as was b). We are now ready to evaluate the expression (/ b (add2 b)). In which environment do we look up b? We always search the environments from the innermost let or lambda expression's environment outward, so we search Environment 2 first, finding that b is bound to 0.5. Thus the whole expression is (/ 0.5 2.5), which evaluates to 0.2.

As an example of how let is used in the definitions of procedures, we reconsider the definition of the procedure remove-leftmost, which was given in Program 4.15. Recall that our objective is to produce a list the same as the list ls except that it has removed from it the leftmost occurrence of item. In the base case, when ls is empty, the answer is the empty list. If (car ls) is equal to item, (car ls) is the leftmost occurrence of item and the answer is (cdr ls). If neither of the cases is true, there are two possibilities: either (car ls) is a pair, or it is not a pair. If it is a pair, we want to determine whether it contains item. In Program 4.15, we used member-all? to determine this. Another way is to check whether (car ls) changes when we remove the leftmost occurrence of item from it. If so, then item must belong to (car ls), in which case the answer is

```
(cons (remove-leftmost item (car ls)) (cdr ls))
```

But if we use this approach, we have to evaluate

```
(remove-leftmost item (car ls))
```

twice, once when making the test and again when doing the consing. To avoid the repeated evaluations of the same thing, we use a let expression to bind a variable, say rem-list, to the value of

```
(remove-leftmost item (car ls))
```

and use rem-list each time the value of this expression is needed. Here is the new code for remove-leftmost:

Program 5.3 remove-leftmost

```
(define remove-leftmost
  (lambda (item ls)
    (cond
      ((null? ls) '())
      ((equal? (car ls) item) (cdr ls))
      ((pair? (car ls))
       (let ((rem-list (remove-leftmost item (car ls))))
         (cons rem-list (cond
                          ((equal? (car ls) rem-list)
                           (remove-leftmost item (cdr ls)))
                          (else (cdr ls))))))
      (else (cons (car ls) (remove-leftmost item (cdr ls)))))))
```

In a let expression

(let ((*var val*)) *body*)

any variables that occur in *val* and are not bound in the expression *val* itself must be bound outside the let expression (i.e., in a nonlocal environment), for in evaluating *val*, Scheme looks outside the let expression to find the bindings of any free variables occurring in *val*. Thus

```
(let ((fact (lambda (n)
              (if (zero? n)
                  1
                  (* n (fact (sub1 n)))))))
  (fact 4))
```

will return a message that fact is unbound. You should try entering this code to become familiar with the messages that your system returns. This message refers to the fact occurring in the lambda expression (written here in italics),

which is not bound outside of the let expression.[2] Thus if we want to use a recursive definition in the "*val*" part of a let-like expression, we have to avoid the problem of unbound variables that we encountered in the above example. We can avoid this difficulty by using a letrec expression (a special form with keyword `letrec`) instead of a let expression to make the local binding when recursion is desired.

The syntax for `letrec` is the same as that for `let`:

```
(letrec ((var1 val1) (var2 val2) ...  (varn valn)) body)
```

but now any of the variables var_1, var_2, …, var_n can appear in any of the expressions val_1, val_2, …, val_n, and refer to the locally defined variables var_1, var_2, …, var_n, so that recursion is possible in the definitions of these variables. The scope of the variables var_1, var_2, …, var_n now includes val_1, val_2, …, val_n, as well as *body*. Thus,

```
(letrec ((fact (lambda (n)
                 (if (zero? n)
                     1
                     (* n (fact (sub1 n)))))))
  (fact 4))
```

has the value 24.

We can also have mutual recursion in a letrec expression, as the next example illustrates:

```
(letrec ((even? (lambda (x)
                  (or (zero? x) (odd? (sub1 x)))))
         (odd? (lambda (x)
                 (and (not (zero? x)) (even? (sub1 x))))))
  (odd? 17))
```

has the value `#t`.

In Program 5.4 we take another look at the iterative version of the factorial procedure discussed in Program 4.19, this time written with `letrec`. Here we are able to define the procedure `fact` with parameter `n` and define the iterative helping procedure `fact-it` within the letrec expression. This enables us to

---

[2] If we call `(fact 0)`, the value 1 is returned, since the consequent of the if expression is true and the alternative, in which the call to `fact` is made, is not evaluated. In this case no error message would result.

Program 5.4 `fact`

```
(define fact
  (lambda (n)
    (letrec ((fact-it
               (lambda (k acc)
                 (if (zero? k)
                     acc
                     (fact-it (sub1 k) (* k acc))))))
      (fact-it n 1))))
```

Program 5.5 `swapper`

```
(define swapper
  (lambda (x y ls)
    (letrec
      ((swap
         (lambda (ls*)
           (cond
             ((null? ls*) '())
             ((equal? (car ls*) x) (cons y (swap (cdr ls*))))
             ((equal? (car ls*) y) (cons x (swap (cdr ls*))))
             (else (cons (car ls*) (swap (cdr ls*))))))))
      (swap ls))))
```

define an iterative version of **fact** without having to use a globally defined helping procedure. There is an advantage to keeping the number of globally defined procedures small to avoid name clashes. Otherwise you might forget that you used a name for something else earlier and assign that name again.

The letrec expression provides a more convenient way of writing code for procedures that take several arguments, many of which stay the same throughout the program. For example, consider the procedure **swapper** defined in Program 2.8, which has three parameters, **x**, **y**, and **ls**, where **x** and **y** are items and **ls** is a list. Then `(swapper x y ls)` produces a new list in which **x**'s and **y**'s are interchanged. Note that in Program 2.8 each time we invoked **swapper** recursively, we had to rewrite the variables **x** and **y**. We can avoid this rewriting if we use **letrec** to define a local procedure, say **swap**, which takes only one formal argument, say **ls***, and rewrite the definition of the procedure **swapper** as shown in Program 5.5.

The parameter to `swap` is `ls*`, and when the locally defined procedure `swap` is called in the last line of the code, its argument is `ls`, which is lambda bound in the outer lambda expression. We could just as well use the variable `ls` instead of `ls*` as the parameter in `swap` since the lexical scoping specifies which binding is in effect. When we call `swapper` recursively in the old code, we write all three arguments, whereas when we call `swap` recursively in the new code, we must write only one argument. This makes the writing of the program more convenient and may make the code itself more readable.

In this section, we have seen how to bind variables locally to procedures using the special forms with keywords `let` and `letrec`. We use these important tools extensively in writing programs that are more efficient and easier to understand.

Exercises

Exercise 5.1

Find the value of each of the following expressions, writing the local environments for each of the nested let expressions. Draw arrows from each variable to the parameter to which it is bound in a lambda or let expression. Also draw an arrow from the parameter to the value to which it is bound.

```
a. (let ((a 5))
     (let ((fun (lambda (x) (max x a))))
       (let ((a 10)
             (x 20))
         (fun 1))))
```

```
b. (let ((a 1) (b 2))
     (let ((b 3) (c (+ a b)))
       (let ((b 5))
         (cons a (cons b (cons c '()))))))
```

Exercise 5.2

Find the value of each of the following letrec expressions:

```
a. (letrec
     ((loop
        (lambda (n k)
          (cond
            ((zero? k) n)
            ((< n k) (loop k n))
            (else (loop k (remainder n k)))))))
     (loop 9 12))
```

```
b. (letrec
     ((loop
        (lambda (n)
          (if (zero? n)
              0
              (+ (remainder n 10) (loop (quotient n 10)))))))
     (loop 1234))
```

Exercise 5.3

Write the two expressions in Parts a and b of Exercise 5.1 as nested lambda expressions without using any let expressions.

Exercise 5.4

Find the value of the following letrec expression.

```
(letrec ((mystery
           (lambda (tuple odds evens)
             (if (null? tuple)
                 (append odds evens)
                 (let ((next-int (car tuple)))
                   (if (odd? next-int)
                       (mystery (cdr tuple)
                                (cons next-int odds) evens)
                       (mystery (cdr tuple)
                                odds (cons next-int evens))))))))
  (mystery '(3 16 4 7 9 12 24) '() '()))
```

Exercise 5.5

We define a procedure **mystery** as follows:

```
(define mystery
  (lambda (n)
    (letrec
      ((mystery-helper
         (lambda (n s)
           (cond
             ((zero? n) (list s))
             (else
               (append
                 (mystery-helper (sub1 n) (cons 0 s))
                 (mystery-helper (sub1 n) (cons 1 s))))))))
      (mystery-helper n '()))))
```

What is returned when **(mystery 4)** is invoked? Describe what is returned when **mystery** is invoked with an arbitrary positive integer.

Exercise 5.6: `insert-left-all`
Rewrite the definition of the procedure `insert-left-all` (See Exercise 4.6.) using a locally defined procedure that takes the list `ls` as its only argument.

Exercise 5.7: `fib`
As in Program 5.4 for `fact`, write an iterative definition of `fib` using `fib-it` (See Program 4.24.) as a local procedure.

Exercise 5.8: `list-ref`
Program 3.7 is a good definition of `list-ref`. Unfortunately, the information displayed upon encountering a reference out of range is not as complete as we might expect. In the definitions of `list-ref`, which precede it, however, adequate information is displayed. Rewrite Program 3.7, using a letrec expression, so that adequate information is displayed.

5.3 Symbolic Manipulation of Polynomials

One of the advantages of a list-processing language like Scheme is its convenience for manipulating symbols in addition to doing the usual numerical calculations. We illustrate this feature by showing how to develop a symbolic algebra of polynomials. By a *symbolic algebra* we mean a program that represents the items under discussion as certain combinations of symbols and then performs operations on these items as symbols rather than as numerical values.

We begin by reviewing what is meant by a polynomial. An expression $5x^4$ is referred to as a *term* in which 5 is the *coefficient* and the exponent 4 is the *degree*. In general, a term is an expression of the form $a_k x^k$, where the coefficient a_k is a real number and the degree k is a nonnegative integer. The symbol x is treated algebraically as if it were a real number. Thus we may add two terms of the same degree, as illustrated by $5x^4 + 3x^4 = 8x^4$. In general, the sum of two terms of like degree is a term of the same degree with coefficient that is the sum of the coefficients of the two terms. This rule is expressed in symbols by

$$a_k x^k + b_k x^k = (a_k + b_k)x^k$$

A term can also be multiplied by a real number, as illustrated by $7(5x^4) = 35x^4$. In general, when we multiply the term $a_k x^k$ by the real number c, the product is a term that has coefficient ca_k and the same degree; thus

$$c(a_k x^k) = (ca_k)x^k$$