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

$$\texttt{(let ((}var_1\ val_1\texttt{) (}var_2\ val_2\texttt{) ... (}var_n\ val_n\texttt{)) }body\texttt{)}$$

is equivalent to the following application of a lambda expression:

$$\texttt{((lambda (}var_1\ var_2\ \ldots\ var_n\texttt{) }body\texttt{) }val_1\ val_2\ \ldots\ val_n\texttt{)}$$

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).

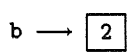

Figure 5.1 Environment 1

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).

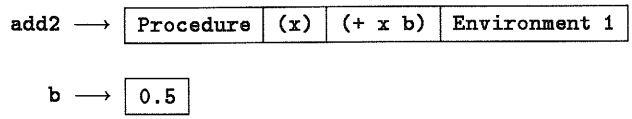

Figure 5.2 Environment 2

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

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

We may also multiply two terms using the following rule: the product of two terms is a term with degree equal to the sum of the degrees of the two terms and with coefficient equal to the product of the coefficients of the two terms. This is expressed symbolically by

$$(a_j x^j)(b_k x^k) = (a_j b_k) x^{j+k}$$

Here is how it looks in a numerical example: $(3x^4)(7x^5) = 21x^9$. It is customary to write a term of degree 0 by writing only its coefficient. The term $a_1 x^1$ of degree 1 is usually written as $a_1 x$, omitting the exponent 1 on x. Thus $3x^0$ is written as 3, and $5x^1$ is written as $5x$. Terms of positive degree with coefficients 0 are generally omitted.

Two terms of like degree can be added to produce a term of the same degree, but two terms of different degrees do not produce a term when added. Instead, we can only indicate the addition by placing a plus sign between the two terms. A *polynomial* is a sum of a finite number of terms, usually arranged in order of decreasing degree. The *degree* of the polynomial is the maximum of the degrees of its terms. Thus the polynomial $3x^4 + 5x^2 + 12$ has degree 4, and the terms of degree 3 and 1 have coefficient 0 and are not written. In general, a polynomial of degree n is of the form

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0$$

where the coefficients a_k, for $k = 0, \ldots, n$ denote real numbers. The sum of two polynomials is the polynomial obtained by adding all of the terms of both polynomials, using the rule given above for adding terms of like degree. Thus the sum of

$$3x^4 + 5x^2 + 12 \quad \text{and} \quad 7x^5 + 6x^4 - x^2 + 11x - 15$$

is the polynomial

$$7x^5 + 9x^4 + 4x^2 + 11x - 3$$

The term of highest degree in a polynomial is known as its *leading term*, and the coefficient of the leading term is known as its *leading coefficient*. The leading term of $3x^4 + 5x^2 + 12$ is $3x^4$, and its leading coefficient is 3.

Our goal is to write programs that produce the sum and product of polynomials. We saw the definition of the sum in the discussion above, and later we shall define the product. As in our development of exact arithmetic in Chapter 3, we again assume that certain constructor and selector procedures for polynomials have been predefined and return to their definition later in this section when we consider the actual representation of the polynomials in

the computer. We proceed to describe what these selector and constructor procedures do when applied to a polynomial.

There are three selector procedures: `degree`, `leading-coef`, and `rest-of-poly`. If `poly` is a polynomial, then `(degree poly)` is the degree of `poly` and `(leading-coef poly)` is the leading coefficient of `poly`. There is a zero polynomial, `the-zero-poly`, which has degree zero and leading coefficient zero. Finally, `(rest-of-poly poly)` is the polynomial obtained from a polynomial of positive degree, `poly`, when its leading term is removed. If `poly` is of degree zero, `(rest-of-poly poly)` is `the-zero-poly`.

The constructor procedure is called `poly-cons`. If n is a nonnegative integer, a is a real number, and `p` is `the-zero-poly` or a polynomial of degree less than n, then `(poly-cons` n a `p)` is the polynomial obtained by adding the leading term ax^n to the polynomial `p`. In particular, for any polynomial `poly`, the value of

```
(poly-cons (degree poly)
           (leading-coef poly)
           (rest-of-poly poly))
```

is `poly`. We shall adopt another convention, which says that a polynomial of positive degree cannot have zero as its leading coefficient. Thus if `poly` has degree less than `n` for some positive `n`, then `(poly-cons n 0 poly)` evaluates to `poly`.

Using these procedures, we proceed to develop our symbolic algebra of polynomials. We begin by devising a test to see if a polynomial is `the-zero-poly`. All we have to ask is whether both its degree and leading coefficient are zero. Thus we define `zero-poly?` in Program 5.6.

Program 5.6 `zero-poly?`

```
(define zero-poly?
  (lambda (poly)
    (and (zero? (degree poly)) (zero? (leading-coef poly)))))
```

Program 5.7 shows how we build a term having degree `deg` and coefficient `coef`. The term so defined is itself a polynomial of degree `deg` consisting of only one term. A polynomial consisting of one term is also referred to as a *monomial*. If we are given a polynomial `poly`, we get its leading term by applying the procedure `leading-term`, which we define in Program 5.8 using `make-term`.

Program 5.7 `make-term`

```
(define make-term
  (lambda (deg coef)
    (poly-cons deg coef the-zero-poly)))
```

Program 5.8 `leading-term`

```
(define leading-term
  (lambda (poly)
    (make-term (degree poly) (leading-coef poly))))
```

We next define the procedure `p+` such that if `poly1` and `poly2` are two polynomials, then `(p+ poly1 poly2)` is the sum of `poly1` and `poly2`. Let us recall that the sum of two terms bx^k and cx^k of the same degree k is a term $(b+c)x^k$ also of degree k. The sum of two polynomials is then the polynomial obtained by adding the terms of like degree in the two polynomials. Our algorithm for adding the two polynomials `poly1` and `poly2` is:

- If `poly1` is `the-zero-poly`, their sum is `poly2`; if `poly2` is `the-zero-poly`, their sum is `poly1`.
- If the degree of `poly1` is greater than the degree of `poly2`, their sum is a polynomial that has the same leading term as `poly1`, and the rest of their sum is the sum of `(rest-of-poly poly1)` and `poly2`.
- If the degree of `poly2` is greater than the degree of `poly1`, their sum is a polynomial that has the same leading term as `poly2` and the rest of their sum is the sum of `(rest-of-poly poly2)` and `poly1`.
- If `poly1` and `poly2` have the same degree n, the degree of their sum is n, and the leading coefficient of their sum is the sum of the leading coefficients of `poly1` and `poly2`, and the rest of their sum is the sum of `(rest-of-poly poly1)` and `(rest-of-poly poly2)`.

This algorithm for the sum, `p+`, of `poly1` and `poly2` leads to Program 5.9.

In this program, the use of the let expression enabled us to write each of the cond clauses more concisely and more clearly. For example, had we not used the let expression, the first cond clause would have looked like

Program 5.9 p+

```
(define p+
  (lambda (poly1 poly2)
    (cond
      ((zero-poly? poly1) poly2)
      ((zero-poly? poly2) poly1)
      (else (let ((n1 (degree poly1))
                  (n2 (degree poly2))
                  (a1 (leading-coef poly1))
                  (a2 (leading-coef poly2))
                  (rest1 (rest-of-poly poly1))
                  (rest2 (rest-of-poly poly2)))
              (cond
                ((> n1 n2) (poly-cons n1 a1 (p+ rest1 poly2)))
                ((< n1 n2) (poly-cons n2 a2 (p+ poly1 rest2)))
                (else
                 (poly-cons n1 (+ a1 a2) (p+ rest1 rest2)))))))))
```

```
((> (degree poly1) (degree poly2))
 (poly-cons (degree poly1)
            (leading-coef poly1)
            (p+ (rest-of-poly poly1) poly2)))
```

Such use of let expressions often makes programs more readable.

We next define the product **p*** of two polynomials **poly1** and **poly2**. The product of the terms $a_k x^k$ and $a_m x^m$ is the term $(a_k \times a_m)x^{k+m}$. To multiply a term $4x^2$ times a polynomial $3x^5 + 2x^3 + 4x + 5$, we multiply each of the terms of the polynomial by $4x^2$ and add the resulting terms to get

$$12x^7 + 8x^5 + 16x^3 + 20x^2$$

Now to multiply two polynomials,

$$4x^2 + 3x + 2 \quad \text{and} \quad 3x^5 + 2x^3 + 4x + 5$$

we first multiply each term of the first by the entire second polynomial to get the three polynomials

$$12x^7 + 8x^5 + 16x^3 + 20x^2$$
$$9x^6 + 6x^4 + 12x^2 + 15x$$
$$6x^5 + 4x^3 + 8x + 10$$

and then we add these three polynomials to get the desired product:

$$12x^7 + 9x^6 + 14x^5 + 6x^4 + 20x^3 + 32x^2 + 23x + 10$$

We now translate the above example into an algorithm for multiplying any two polynomials `poly1` and `poly2`. It will be convenient to define locally the product `t*` of a term `trm` and a polynomial `poly`. The algorithm for this is:

- If `poly` is `the-zero-poly`, then `(t* trm poly)` is just `the-zero-poly`.
- Otherwise the degree of their product is the sum of the degrees of `trm` and `poly`. The leading coefficient of their product is the product of the coefficient of `trm` and the leading coefficient of `poly`. The rest of their product is just the product of `trm` and the rest of `poly`.

Once `t*` has been defined, the product `p*` of `poly1` and `poly2` can be defined using the following algorithm:

- If `poly1` is `the-zero-poly`, then `(p* poly1 poly2)` is just `the-zero-poly`.
- Otherwise, we multiply the leading term of `poly1` by `poly2`, and add that to the product of the rest of `poly1` and `poly2`.

This leads us to Program 5.10 for `p*`. In this program, `t*` is defined locally using a letrec expression since it is a recursive definition. The lambda expression for the procedure `p*` is placed within the body of the letrec expression for `t*` so that the local procedure it defines is available for use in `p*-helper`. The letrec expression that defines `p*-helper` is used because the variable `poly2` is not changed in the recursive invocations of the procedure being defined. Thus it is better programming style not to carry it along as a parameter. We define the local procedure `p*-helper` that has as its only parameter the polynomial `p1` that is bound to `poly1` when it is later invoked.

Program 5.11 defines a unary operation `negative-poly` such that when `poly` is a polynomial, `(negative-poly poly)` is its negative: the polynomial with the signs of all of its coefficients changed. We compute it by multiplying `poly` by the polynomial that is a term of degree 0 and leading coefficient -1.

Now that we have the negative of a polynomial, we can define the difference `p-` between `poly1` and `poly2` as shown in Program 5.12.

We now consider how to find the value of a polynomial `poly` when a number is substituted for the variable `x`. To evaluate the polynomial $4x^3+8x^2-7x+6$ for $x = 5$, we substitute 5 for x and get $4(5^3)+8(5^2)-7(5)+6$. The polynomial can be evaluated at a given value of x by computing each term $a_k x^k$ separately

Program 5.10 p*

```
(define p*
  (letrec
    ((t* (lambda (trm poly)
           (if (zero-poly? poly)
               the-zero-poly
               (poly-cons
                 (+ (degree trm) (degree poly))
                 (* (leading-coef trm) (leading-coef poly))
                 (t* trm (rest-of-poly poly)))))))
    (lambda (poly1 poly2)
      (letrec
        ((p*-helper (lambda (p1)
                      (if (zero-poly? p1)
                          the-zero-poly
                          (p+ (t* (leading-term p1) poly2)
                              (p*-helper (rest-of-poly p1)))))))
        (p*-helper poly1)))))
```

Program 5.11 negative-poly

```
(define negative-poly
  (lambda (poly)
    (let ((poly-negative-one (make-term 0 -1)))
      (p* poly-negative-one poly))))
```

Program 5.12 p-

```
(define p-
  (lambda (poly1 poly2)
    (p+ poly1 (negative-poly poly2))))
```

and then adding the results. For example, we have

$$4(5^3) + 8(5^2) - 7(5) + 6 = 4 \times (5 \times 5 \times 5) + 8 \times (5 \times 5) - 7 \times (5) + 6$$

but this is very inefficient. If we evaluate this by computing each x^k by multiplying x by itself $k - 1$ times and then multiplying the result by a_k, we

would be using k multiplications. For a polynomial of degree n, we must add the number of multiplications for each term, which is $1 + 2 + 3 + \cdots + n = n(n+1)/2$ multiplications, to which we add the n additions needed to add up the terms, and we get a grand total of $n(n+1)/2 + n$ operations. We can reduce this number of operations significantly by using the method of nested multiplication, also known as *Horner's rule*, or *synthetic division*.

Before we derive the method of nested multiplication, we consider as an example the polynomial $P(x) = 4x^3 + 8x^2 - 7x + 6$. If we write the constant term first and factor an x out of the rest of the terms, we get $P(x) = 6 + x(-7 + 8x + 4x^2)$. We next factor an x out of the terms after the -7 in the parentheses, to get $P(x) = 6 + x(-7 + x(8 + x(4)))$. For $x = 5$, this becomes $6 + 5(-7 + 5(8 + 5(4)))$. Whereas evaluating the polynomial $P(x)$ in its original form required nine operations, in this last form only six operations are required—three multiplications and three additions.

In the general case of a polynomial of degree n, we note that all terms of degree 1 or more contain a factor of x, so we can factor it out to get our polynomial in the following form:

$$a_0 + x(a_1 + a_2x + a_3x^2 + \cdots + a_nx^{n-1})$$

We repeat this process, starting with the term a_1, to represent our polynomial as:

$$a_0 + x(a_1 + x(a_2 + a_3x + \cdots + a_nx^{n-2}))$$

By continuing to factor out an x from the terms after the constant term, we finally arrive at the result:

$$a_0 + x(a_1 + x(a_2 + x(a_3 + \cdots + x(a_{n-1} + xa_n)\ldots)))$$

In this method of evaluating the polynomial, we have n multiplications and n additions, so there are altogether $2n$ operations. In this way, the number of operations grows linearly with n (that is, like n to the first power, or using the notation of Chapter 3, $O(n)$) while in the previous way, it grew quadratically (that is, like n to the second power, or $O(n^2)$).

We next define a procedure `poly-value` such that `(poly-value poly num)` is the value of the polynomial `poly` when `num` is substituted for `x`. A clue for defining this procedure recursively (in fact, iteratively) comes from observing that if we think of the last expression $a_{n-1} + xa_n$ as a coefficient b, then the expression is just a polynomial of degree $n-1$ having leading coefficient b and all terms of degree less than $n-1$ the same as those in `poly`. In implementing this, we obtain a_{n-1} by taking the leading coefficient of `(rest-of-poly poly)`. But this works only if `(rest-of-poly poly)` has degree

Program 5.13 poly-value

```
(define poly-value
  (lambda (poly num)
    (letrec
      ((pvalue (lambda (p)
                 (let ((n (degree p)))
                   (if (zero? n)
                       (leading-coef p)
                       (let ((rest (rest-of-poly p)))
                         (if (< (degree rest) (sub1 n))
                             (pvalue (poly-cons
                                      (sub1 n)
                                      (* num (leading-coef p))
                                      rest))
                             (pvalue (poly-cons
                                      (sub1 n)
                                      (+ (* num (leading-coef p))
                                         (leading-coef rest))
                                      (rest-of-poly rest)))))))))
      (pvalue poly))))
```

$n-1$, that is, when $a_{n-1} \neq 0$. Thus, if **(rest-of-poly poly)** has degree less than $n-1$, we use xa_n for b. Thus the code for **poly-value** in Program 5.13 treats two cases depending upon the degree of **rest**.

This program is iterative since when **pvalue** is called in the if clauses, no further operation is performed after the application of **pvalue**. Moreover, because the procedure of one argument **pvalue** appears first in both the consequent and the alternative of the last if expression, it can be pulled out of the if expression so that the body of the last let expression reads

```
(pvalue (if (< (degree rest) (sub1 n))
            (poly-cons
              ...)
            (poly-cons
              ...)))
```

The last thing we illustrate before thinking about the representation of the polynomial is how to build a given polynomial. For example, if we want to define **p1** to be the polynomial

$$5x^3 - 3x^2 + x - 17$$

Program 5.14 The five basic definitions (Version I)

```
(define the-zero-poly '(0))

(define degree
  (lambda (poly)
    (sub1 (length poly))))

(define leading-coef
  (lambda (poly)
    (car poly)))

(define rest-of-poly
  (lambda (poly)
    (cond
      ((zero? (degree poly)) the-zero-poly)
      ((zero? (leading-coef (cdr poly)))
       (rest-of-poly (cdr poly)))
      (else (cdr poly)))))

(define poly-cons
  (lambda (deg coef poly)
    (let ((deg-p (degree poly)))
      (cond
        ((and (zero? deg) (equal? poly the-zero-poly)) (list coef))
        ((>= deg-p deg)
         (error "poly-cons: Degree too high in" poly))
        ((zero? coef) poly)
        (else
          (cons coef
                (append (list-of-zeros (sub1 (- deg deg-p)))
                        poly)))))))
```

we simply write

```
(define p1 (poly-cons 3 5
             (poly-cons 2 -3
               (poly-cons 1 1
                 (poly-cons 0 -17 the-zero-poly)))))
```

Using the concept of data abstraction again, we have been able to develop a symbolic algebra of polynomials without knowing how the polynomials are represented. We did this by assuming that we had the selector and constructor

procedures and the zero polynomial. We shall now see several ways in which these can be defined.

A polynomial is completely determined if we give a list of its coefficients, where we enter a zero when a term of a given degree is missing. The degree of the polynomial is then one less than the length of the list of coefficients. Thus the polynomial

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0, \qquad a_n \neq 0$$

can be represented by the list $(a_n\ a_{n-1}\ \cdots\ a_1\ a_0)$. For example, the polynomial $5x^3 - 7x^2 + 21$ is represented by the list (5 −7 0 21), where the zero corresponds to the term $0x^1$ that is suppressed in $5x^3 - 7x^2 + 21$.

If this representation of a polynomial as a list of its coefficients is adopted, we can make the five basic definitions for the symbolic algebra of polynomials as shown in Program 5.14. Since we required that the leading coefficient of a polynomial be different from zero, we put a zero test in the second cond clause in the definition of `rest-of-poly` to skip over the missing zeros. In the definition of `poly-cons`, the third cond clause guards against a leading coefficient of zero, and the last line uses the procedure `list-of-zeros` defined in Program 3.5 to fill in missing zeros if `deg` differs from the degree of `p` by more than 1.

The above representation of a polynomial by its list of coefficients has an obvious disadvantage when we try to represent the polynomial $x^{1000} + 1$. We would have to construct a list of 1001 numbers, all zero except the first and last. The above representation is perfectly adequate when we are dealing with polynomials of low degree, but it becomes cumbersome when we have to write "sparse" polynomials of high degree. The following representation of polynomials is more convenient for such higher-degree polynomials; we represent the polynomial by a list of pairs of numbers. In each pair, the first element is the degree of a term, and the second element is the coefficient of that term. The pairs corresponding to terms with zero coefficients are not included, except for `the-zero-poly`, which is represented by `((0 0))`. The pairs are ordered so that the degrees decrease. Thus the polynomial

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0, \qquad a_n \neq 0$$

has the representation $((n\ a_n)\ (n{-}1\ a_{n-1})\ \cdots\ (1\ a_1)\ (0\ a_0))$, for those terms with $a_i \neq 0$. We then can write the five basic definitions for the algebra of polynomials as shown in Program 5.15.

There may be other representations of polynomials that are more convenient to use in special circumstances. The advantage of our approach using data abstraction is that we only need to define `the-zero-poly`, `degree`,

Program 5.15 The five basic definitions (Version II)

```
(define the-zero-poly '((0 0)))

(define degree
  (lambda (poly)
    (caar poly)))

(define leading-coef
  (lambda (poly)
    (cadar poly)))

(define rest-of-poly
  (lambda (poly)
    (if (null? (cdr poly))
        the-zero-poly
        (cdr poly))))

(define poly-cons
  (lambda (deg coef poly)
    (let ((deg-p (degree poly)))
      (cond
        ((and (zero? deg) (equal? poly the-zero-poly))
         (list (list 0 coef)))
        ((>= deg-p deg)
         (error "poly-cons: Degree too high in" poly))
        ((zero? coef) poly)
        (else
          (cons (list deg coef) poly))))))
```

`leading-coef`, `rest-of-poly`, and `poly-cons`, and the rest of our algebra of polynomials is still valid with no modifications.

Exercises

Exercise 5.9

Implement the algebra of polynomials in the two ways indicated in the text. For each implementation, test each of the procedures p+, p−, and p* with the polynomials

$$p_1(x) = 5x^4 - 7x^3 + 2x - 4$$

$$p_2(x) = x^3 + 6x^2 - 3x$$

and using `poly-value`, find $p_1(-1)$, $p_1(2)$, $p_2(0)$, and $p_2(-2)$.

Exercise 5.10
Look closely at the definition of `p+` (see Program 5.9). When `n1` is greater than `n2`, the variables `a2` and `rest2` are ignored. Similarly, when `n1` is less than `n2`, the variables `a1` and `rest1` are ignored. Rewrite `p+` so that this wasting of effort disappears. *Hint*: You will need to use `let` within the consequents of cond clauses.

Exercise 5.11: `poly-quotient`, `poly-remainder`
Define a procedure `poly-quotient` that finds the quotient polynomial when `poly1` is divided by `poly2` and a procedure `poly-remainder` that finds the remainder polynomial when `poly1` is divided by `poly2`.

Exercise 5.12
Another representation of polynomials as lists that can be used is a list of coefficients in the order of increasing degree. The list of pairs representation given above can also be written in order of increasing degree. Consider the advantages and disadvantages of these representations compared to those given above.

Exercise 5.13
How would the constructors and selectors be defined if we use

`(cons deg coef)` instead of `(list deg coef)`

in our second representation using lists of pairs?

Exercise 5.14
The definition of `t*` in Program 5.10 is flawed. Each time `t*` is invoked recursively it evaluates both `(degree trm)` and `(leading-coef trm)`, although these values never change. In addition, the variable `trm` does not need to be passed to `t*` because `trm` never changes. Explain how these two flaws are eliminated in the following definition of `p*`.

```
(define p*
  (let
    ((t* (lambda (trm poly)
           (let ((deg (degree trm))
                 (lc (leading-coef trm)))
             (letrec
               ((t*-helper
                  (lambda (poly)
                    (if (zero-poly? poly)
                        the-zero-poly
                        (poly-cons
                          (+ deg (degree poly))
                          (* lc (leading-coef poly))
                          (t*-helper (rest-of-poly poly)))))))
               (t*-helper poly))))))
    (lambda (poly1 poly2) ...)))
```

Exercise 5.15: `append-to-list-of-zeros`
In the first version of `poly-cons` presented in Program 5.14, `poly` is appended to a list of zeros. The procedure `list-of-zeros` requires one recursion to build the list of zeros and `append` requires another. Two recursions over the list of zeros is inefficient. The program can be rewritten so as to require only one recursion over the list of zeros. One suggestion for doing so is to combine the construction of the list of zeros and the appending of `poly` into one procedure `append-to-list-of-zeros`, which takes two parameters, `n` and `x` and produces a list that contains `x` preceded by `n` zeros. This procedure can be written either recursively or iteratively. Try your hand at both versions and test them in `poly-cons`.

5.4 Binary Numbers

Information is stored in the computer in the form of binary numbers. One may loosely think of the memory cells in which information is stored as a row of switches, each having two positions: on and off. If a switch is on, it represents the digit 1, and if it is off, it represents the digit 0. The information contained in one such switch is called a *bit*, and eight bits of information usually constitute a *byte* of information. Since there are 2^8 different settings for eight switches, we can represent 256 different values by using one byte. In this section, we shall discuss the representation of numbers in binary form, and more generally, as numbers with an arbitrary base.

First recall that in the decimal system, each digit in a number is a placeholder representing the number of times a certain power of 10 is counted. Thus 4,723 is the same as

$$4 \times 10^3 + 7 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$$

The 10 is called the *base* of the number system. We can think of any number represented in this way as a polynomial in which the variable x has been replaced by 10. In the same way, we can represent any number as a polynomial in which the variable x has been replaced by the base b and the coefficients are taken to be numbers between 0 and $b-1$. It is customary to write a number in the base b system using the placeholder concept as a string of digits, each digit being the corresponding coefficient in the polynomial. For example, for base 2 (*binary numbers*), the digits are 0 and 1 and the polynomial

$$1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

can be represented as 1100101.

In general, for binary numbers, the number

$$a_n 2^n + a_{n-1} 2^{n-1} + \cdots + a_1 2 + a_0$$

is written in the form $a_n a_{n-1} \ldots a_1 a_0$. We first consider the problem of finding the decimal number when we are given its binary representation. This is precisely the problem of evaluating a polynomial when the variable x has the value 2. We can use the results of the last section if we represent our binary number as a polynomial of degree n that has the coefficient a_k for the term of degree k, for $k = 0, 1, \ldots, n$. We define a procedure `digits->poly` that takes a list of the digits of a binary number as its argument and returns a polynomial of degree one less than the number of digits and that has the given digits as its coefficients. The polynomial is constructed by the local procedure `make-poly`, which has as its parameters the degree `deg` of the polynomial, which is one less than the number of digits in the binary number, and `ls`, which is the list of digits of the binary number. If we already have the polynomial for the binary number obtained when the first digit is removed, that is, for parameters `(sub1 deg)` and `(cdr ls)`, we get the polynomial for the parameters `deg` and `ls` by adding the term having degree `deg` and coefficient `(car ls)`. This leads us to the definition given in Program 5.16.

Now to convert from the binary representation of a number to the decimal number, we use the procedure `binary->decimal` given in Program 5.17, which takes a list of the binary digits as its argument and returns the decimal

Program 5.16 `digits->poly`

```
(define digits->poly
  (lambda (digit-list)
    (if (null? digit-list)
        (error "digits->poly: Not defined for the empty list")
        (letrec
          ((make-poly
             (lambda (deg ls)
               (if (null? ls)
                   the-zero-poly
                   (poly-cons deg (car ls)
                     (make-poly (sub1 deg) (cdr ls)))))))
          (make-poly (sub1 (length digit-list)) digit-list)))))
```

Program 5.17 `binary->decimal`

```
(define binary->decimal
  (lambda (digit-list)
    (poly-value (digits->poly digit-list) 2)))
```

number. As an example, we find the decimal number for the representation of the binary number 11001101.

```
(binary->decimal '(1 1 0 0 1 1 0 1)) ⟹ 205
```

If we have a polynomial, say $1x^2+1$, which corresponds to the binary number 101, how do we recover the list of binary digits `(1 0 1)`? To do this, we define a procedure `poly->digits` that takes a polynomial `poly` corresponding to a binary number and returns a list of the digits of that binary number. For example,

```
(poly->digits (digits->poly '(1 1 0 1 0 1))) ⟹ (1 1 0 1 0 1)
```

Consider `(rest-of-poly poly)`; if its degree is one less than the degree of `poly`, then `(poly->digits poly)` is just

```
(cons (leading-coef poly) (poly->digits (rest-of-poly poly)))
```

Program 5.18 `poly->digits`

```
(define poly->digits
  (lambda (poly)
    (letrec
      ((convert
         (lambda (p deg)
           (cond
             ((zero? deg) (list (leading-coef p)))
             ((= (degree p) deg)
              (cons (leading-coef p)
                    (convert (rest-of-poly p) (sub1 deg))))
             (else
              (cons 0 (convert p (sub1 deg))))))))
      (convert poly (degree poly)))))
```

Otherwise, we have to cons zeros onto the list to take into account the gap in the degrees between the leading term and the next term with nonzero coefficient. In order to do this, it is convenient to introduce a local procedure, which we call **convert**. It keeps track of the degree of the term being considered, even if the coefficient is zero. Thus **convert** has two parameters: **p**, which is a polynomial, and **deg**, which is an integer representing the degree of the term. We define **poly->digits** as shown in Program 5.18.

We also want to convert from the decimal number to its binary representation. We shall do this with the procedure **decimal->binary**, which takes a decimal number and returns a list of the digits in its binary representation. We can easily derive the algorithm if we recall that we want to find the coefficients $a_n, a_{n-1}, \ldots, a_0$ in the polynomial corresponding to the number q, which we now write using nested multiplication:

$$q = a_0 + 2(a_1 + \cdots + 2(a_{n-1} + 2a_n)\cdots)$$

Observe that a_0, which must be either 0 or 1, is just the remainder r_0 when q is divided by 2, since $q = a_0 + 2(somenumber)$. Recall that r_0 is 0 if q is even, and it is 1 if q is odd. If we let q_0 be the quotient when q is divided by 2, then we have

$$q_0 = a_1 + 2(a_2 + \cdots + 2(a_{n-1} + 2a_n)\cdots)$$

We now repeat this process to find that a_1 is the remainder r_1 when q_0 is divided by 2, and so forth. In general, if q_k is the quotient when q_{k-1} is

Quotient	Remainder
197	
98	1
49	0
24	1
12	0
6	0
3	0
1	1
0	1

Figure 5.19 Conversion of 197 to its binary representation

divided by 2 and if r_k is the remainder when q_{k-1} is divided by 2, then $a_k = r_k$.

For example, to convert the decimal number 197 to binary form, we do our work in two columns; the first gives the quotient, and the second gives the remainder when the successive numbers are divided by 2. Figure 5.19 shows this computation. Each line in the table represents the quotient and the remainder when the previous quotient is divided by 2. The binary representation of the number is found by reading the remainders from the bottom of the table to the top: 11000101. We will then have

```
(decimal->binary 197) ⟹ (1 1 0 0 0 1 0 1)
```

Implementing this algorithm is accomplished in Program 5.20 by building up the polynomial corresponding to the binary number term by term as the remainders are obtained. The first term we build has degree 0 and the degrees increase by one each time a new remainder is found. Thus we are able to define **decimal->binary** with the help of **dec->bin**, which has a second parameter **deg** that keeps track of the degree of the term, starting from zero and increasing by one in each recursive invocation.

We now have

```
(decimal->binary (binary->decimal '(1 0 1 1 0))) ⟹ (1 0 1 1 0)
(binary->decimal (decimal->binary 143)) ⟹ 143
```

Two other number systems that are commonly used in computing are the *octal* (base 8) and the *hexadecimal* (base 16) systems. For octal, the base b in the polynomial representation is replaced by 8, and in hexadecimal, the base b is

Program 5.20 `decimal->binary`

```
(define decimal->binary
  (lambda (num)
    (letrec
      ((dec->bin
         (lambda (n deg)
           (if (zero? n)
               the-zero-poly
               (p+ (make-term deg (remainder n 2))
                   (dec->bin (quotient n 2) (add1 deg)))))))
      (poly->digits (dec->bin num 0)))))
```

replaced by 16. The digits $0, 1, 2, 3, 4, 5, 6, 7$ are used for octal numbers and the digits $0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F$ for hexadecimal numbers. Here A stands for 10, B for 11, ..., F for 15.

Exercises

Exercise 5.16
Convert each of the following decimal numbers to base 2.

a. 53

b. 404

Exercise 5.17
Convert each of the following base 2 numbers to decimals.

a. 10101010

b. 1101011

Exercise 5.18: `octal->decimal`, `hexadecimal->decimal`
Look over the programs for `binary->decimal` and `decimal->binary` and see what changes have to be made to get definitions for the four procedures:

```
octal->decimal
hexadecimal->decimal
decimal->octal
decimal->hexadecimal
```

Since we are representing our hexadecimal numbers as lists of digits, we can use the number `10` for `A`, `11` for `B`, and so on, so that `(12 5 10)` is the list

representation of the hexadecimal number `C5A`. Define one pair of conversion procedures `base->decimal` and `decimal->base` that take two arguments, the number to be converted and the base, where the base can be any positive integer. Then define a procedure `change-base` that changes a number `num` from base `b1` to base `b2`, where `num` is a list of digits. Thus `(change-base num b1 b2)` is a list of digits that gives the base `b2` representation of `num`. Test your program on:

```
(change-base '(5 11) 16 8) ⟹ (1 3 3)
(change-base '(6 6 2) 8 2) ⟹ (1 1 0 1 1 0 0 1 0)
(change-base '(1 0 1 1 1 1 1 0 1) 2 16) ⟹ (1 7 13)
```

Exercise 5.19: **`binary-sum, binary-product`**
Define two procedures, `binary-sum` and `binary-product`, that take two binary numbers as arguments and return the sum and product of those numbers in binary form. This can be done in two ways. First, you could convert both numbers to decimal form, perform the arithmetic operation, and then convert to binary form. You could, on the other hand, treat the binary numbers as polynomials and perform the arithmetic operations on these polynomials, using the appropriate carrying rules for binary numbers. Write programs for `binary-sum` and `binary-product` using both approaches.

Exercise 5.20: **`binary->decimal, decimal->binary`**
We have presented the conversions from binary to decimal and from decimal to binary as applications of the algebra of polynomials developed in this chapter. Write the two procedures `binary->decimal` and `decimal->binary` directly from the definitions of binary and decimal numbers, using the list representation for binary numbers and not making use of the polynomial algebra.
