# 10 FUNCTIONAL PROGRAMMING: LISP

## 10.1 DESIGN: CONTROL STRUCTURES

### Atoms Are the Only Primitives

LISP is remarkable for the simplicity of its basic control structures. The only primitive control structures are literals and unquoted atoms since these are the only constructs that do not alter the control flow. Literals represent themselves: For example, numbers and quoted atoms and lists are names for themselves. Unquoted atoms are bound to either functions (if they have the `expr` property) or data values (if they have the `apval` property). There are only two basic control-structure constructors: the conditional expression and the recursive application of a function to its arguments.

### The Conditional Expression Is a Major Contribution

In the historical discussion of LISP, we mentioned that LISP was the first language to contain a conditional expression. This was an important idea since it meant that everything could be written as an expression. Previous languages, such as FORTRAN, and some newer languages, such as Pascal, require the user to drop from the expression level to the statement level in order to make a choice. Languages that force expressions to be broken up in this way can often make programs less readable. Mathematicians have recognized for many years the value of conditional expressions and have often used them. For example, here is a typical definition of the *signum* (sign) function:

$$\mathrm{sg}(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{if } x = 0 \\ -1, & \text{if } x < 0 \end{cases}$$

In LISP this function is defined:

```
(defun sg (x)
   (cond ((plusp x)   1)
         ((zerop x)   0)
         ((minusp x) -1)) )
```

The LISP conditional has more parentheses than are really necessary; instead of

(cond ($p_1$ $e_1$) ... ($p_n$ $e_n$))

it would have been quite adequate if LISP had been designed to use

(cond $p_1$ $e_1$ ... $p_n$ $e_n$)

However, the latter notation complicates the interpreter slightly and, as we said in the beginning of Chapter 9, nobody imagined that the *S*-expression notation would become the standard way of writing LISP programs. For convenience, Common LISP provides (if $p$ $e_1$ $e_2$) as a synonym for the two-branch conditional (cond ($p$ $e_1$) (t $e_2$))

## The Logical Connectives Are Evaluated Conditionally

McCarthy took the conditional expression as one of the fundamental primitives of LISP. Therefore, it was natural to define the other logical operations in terms of it. For example, the function (or $x$ $y$) has the value t (true) if either or both of $x$ and $y$ have the value t; in any other case, or has the value nil (false). Another way to say this is if $x$ has the value t, then the or has the value t, otherwise the or has the same value as $y$. Write out a truth table for each of these definitions of or to see that they are equivalent. The latter definition allows or to be defined as a conditional:

(or $x$ $y$) = (if $x$ t $y$)

- ***Exercise 10-1:*** Define the and function in terms of the conditional.
- ***Exercise 10-2:*** Define the not function in terms of the conditional.
- ***Exercise 10-3:*** Use truth tables to show that the conditional definitions of and, or, and not give the same results as the usual definitions.

These definitions of the logical connectives have an important property they do not possess in their usual interpretation. That is the fact that their operands are *evaluated sequentially.* To see the importance of this, consider the following example. Suppose we wish to know if a list either has key as its first element or is null. We might write this as

```
(or (eq (car L) 'key) (null L) )
```

This is incorrect, however, since if L is null, it is an error to apply (car L). In most languages (e.g., Algol and Pascal), the only way around this problem is to write a conditional

instead of the `or` (and in Pascal, without a conditional expression, it is quite awkward). In *LISP the solution is to write the operands to* `or` *in the opposite order:*

```
(or (null L) (eq (car L) 'key) )
```

*Why does this work? This application is equivalent to*

```
(if (null L) t (eq (car L) 'key))
```

That is, first evaluate `(null L)`; if it is true, then we know the result is true, so we return `t` *without ever having attempted to evaluate* `(eq (car L) 'key)`. The latter expression is evaluated only if `(null L)` is false, in which case the application is valid since `L` is guaranteed to have at least one element.

This interpretation of the logical connectives is known as the *conditional* or *sequential interpretation*, as opposed to the *strict interpretation*, which always evaluates both arguments. The more "lenient" sequential interpretation of the connectives often simplifies the definition of functions, although it may make it more difficult to prove things about them.

LISP allows both `and` and `or` to have more than two arguments. Therefore,

```
(and p1 p2 ... pn)
```

evaluates the $p_i$ in order. As soon as one returns false (i.e., `nil`), the `and` immediately returns false without evaluating the rest of the arguments. Similarly,

```
(or p1 p2 ... pn)
```

evaluates the arguments until it encounters one that returns true.

## Iteration Is Done by Recursion

Except for the conditional, LISP needs none of the control structures found in conventional programming languages.[1] In particular, all forms of iteration are performed by recursion. We have seen this in previous examples, such as the `append` and `getprop` functions.

Consider the `getprop` function:

```
(defun getprop (p x)
   (if (eq (car x) p)
       (cadr x)
       (getprop p (cddr x)) ))
```

This is analogous to a **while**-loop in a Pascal-like language; the function continues to call itself recursively until a termination condition is satisfied.

Although Common LISP has an analog of a **for**-loop, it generally does not need one. In most programming languages, **for**-loops are used to control an index for array subscripting.

---

[1] This is not entirely true. Most LISPs have a vestigial imperative facility (the `prog` feature) complete with **goto**s and assignment statements. Since this facility is conventional and uncharacteristic of LISP, we will not discuss it further.

In LISP it is more common to write a recursive procedure that performs some operation on every element of a list. For example, suppose we need to add all of the elements of a list of numbers, which is the kind of thing we would use a **for**-loop for in Pascal. In LISP we can write a recursive function `plus-red` that does the *plus reduction* of a list:

```
(defun plus-red (a)
  (if (null a)
      0
      (plus (car a) (plus-red (cdr a)) )) )
```

Here are two example applications of `plus-red`:

```
(plus-red   '(1 2 3 4 5))
  15
(plus-red   '(3 3 3 3))
  12
```

Trace through the execution of these examples to be sure that you understand them.

Notice that `plus-red` is quite general: It works on any list of numbers, regardless of its size. There is no "upper-bound" written into the program as there would be in most languages, and there is no explicit indexing that can go wrong. Thus it obeys the Zero-One-Infinity Principle.

The `plus-red` function is an example of a *reduction*, that is, the use of a binary function to reduce a list to a single value. The next example we will see is an example of *mapping* a function, that is, applying a function to every element of a list and returning a list of the results.

Suppose we want to add one to each element of a list. This can be done by `add1-map`:

```
(defun add1-map (a)
  (if (null a)
      nil
      (cons (add1 (car a)) (add1-map (cdr a)) )) )
```

For example,

```
(add1-map '(1 9 8 4))
  (2 10 9 5)
```

Again, there are no explicit bounds, no controlled variables, and no indexing.

We have seen an example where we took a list and *reduced* it to one value. We have also seen an example where we took a list and *mapped* it into another list of the same size. Next we will see an example of *filtering* a list, that is, forming a sublist containing all the elements that satisfy some property. In this case, we want to form a list containing all of the negative elements of the given list. That is,

```
(minusp-fil '(2 -3 7 -1 -6 4 8))
  (-3 -1 -6)
```

The function to do this is straightforward; we just make the consing of each element conditional on whether it is negative:

```
(defun minusp-fil (a)
  (cond ((null a) nil)
        ((minusp (car a))
         (cons (car a) (minusp-fil (cdr a)) ))
        (t (minusp-fil (cdr a)) )) )
```

All of the examples we have seen so far are the equivalent of a single loop; next we will see the equivalent of two **for**-loops, one nested in the other. Suppose we want a function `all-pairs` that forms all pairs that can be formed from the elements of two lists. For example,

```
(all-pairs '(a b c) '(x y z))
  ((a x) (a y) (a z) (b x) (b y) (b z) (c x) (c y) (c z))
```

You may recognize this as the *Cartesian product* of the two lists. In a conventional language, this would be accomplished with two nested loops, one iterating over the elements of the first list and the other iterating over the elements of the second list. The same is accomplished in LISP with two nested recursive functions: `all-pairs` will do the outer recursion over the elements of the first list and `distl` ("distribute from the left") will do the inner iteration over all of the elements of the second list. For example,

```
(distl 'b '(x y z))
  ((b x) (b y) (b z))
```

*Here are the definitions:*

```
(defun all-pairs (M N)
  (if (null M)
      nil
      (append (distl (car M) N)
              (all-pairs (cdr M) N)) ))
(defune distl (x N)
  (if (null N)
      nil
      (cons (list x (car N))
            (distl x (cdr N)) )) )
```

(The `list` function makes a list out of its arguments.)

■ ***Exercise 10-4:*** Write a function `times-red` that computes the times reduction of a list. For example,

```
(times-red '(1 2 3 4))
  24
```

What should be the result of `(times-red '())` for the natural recursive definition to work?

■ ***Exercise 10-5:*** Write an expression that computes the product of adding 1 to each element of a list. For example,

```
(times-add1 '(1 2 3))
  24
```

since 24 = (1 + 1) × (2 + 1) × (3 + 1). *Note:* You do not have to define any functions beyond those already defined.

■ ***Exercise 10-6:*** Write a function `append-red`, for example,

```
(append-red '( (to be) (or not) (to be) ))
  (to be or not to be)
```

■ ***Exercise 10-7:*** Write a function `zerop-map` that returns a Boolean list representing whether the corresponding elements of the argument list are zero (as tested by `zerop`). For example,

```
(zerop-map '(4 7 0 3 -2 0 1))
  (nil nil t nil nil t nil)
```

■ ***Exercise 10-8:*** Write a `plus-map` function that adds the corresponding elements of two lists and returns a list of the results. For example,

```
(plus-map '(1 2 3 4) '(1 9 8 4))
  (2 11 11 8)
```

Notice that this is different from the other mapping functions because `plus` is binary whereas `add1`, `zerop`, and so forth are unary. What will you do if the lists are of unequal length? Justify your answer.

■ ***Exercise 10-9:*** Trace in detail the following application of `all-pairs`:

```
(all-pairs '(Bob Ted) '(Carol Alice))
  ((Bob Carol) (Bob Alice) (Ted Carol) (Ted Alice))
```

## Hierarchical Structures Are Processed Recursively

Most of the uses of recursion we have seen so far could have been written iteratively (e.g., using a **while**-loop) almost as easily. Next, we will discuss the use of recursion to process *hierarchical structures* that would be difficult to handle iteratively.

As an example we will design the `equal` function, which determines whether two arbitrary values are the same. How is this different from the `eq` primitive? Recall that the `eq` primitive works only on atoms; its application to nonatomic values is undefined (in most LISP systems). The `equal` function will do much more: It will tell us if two arbitrary list structures are the same. For example,

```
(equal '(a (b c) d (e)) '(a (b c) d (e)) )
  t
(equal '(to be or not to be) '((to be) or (not (to be))) )
  nil
```

```
(equal '(1 2 3) '(3 2 1) )
  nil
(equal 'Paris '(Don Smith) )
  nil
(equal 'Paris 'London )
  nil
(equal nil '(be 2) )
  nil
```

The `equal` function is applicable to *any* two arguments (see, however, p. 350).

We will design the `equal` function in the same way we have designed other recursive functions: by solving those cases that are easy and then reducing the complicated cases to the easy cases. What are the easy cases? Atoms can be handled immediately since they can be compared with the `eq` function. Therefore, if both $x$ and $y$ are atoms, then `(equal` $x$ $y$`)` reduces to `(eq` $x$ $y$`)`. If either $x$ or $y$ is an atom and the other is not, then we know they cannot be equal. This can be summarized as follows:

if $x$ and $y$ are both atoms, then
  `(equal` $x$ $y$`)` = `(eq` $x$ $y$`)`
if $x$ is an atom and $y$ isn't,
or $y$ is an atom and $x$ isn't,
  then `(equal` $x$ $y$`)` = `nil`

These two can be combined using LISP's sequential `and`:

```
(and (atom x) (atom y) (eq x y))
```

Because of the sequential interpretation, the `eq` application will not be evaluated unless both $x$ and $y$ are atoms. If either is not an atom or they are not the same atom, then the above expression returns false. Notice that this also takes care of the case where either $x$ or $y$ is `nil` since in LISP `nil` is considered an atom.

Let's consider the case where neither $x$ nor $y$ is an atom. Clearly, we want to compare $x$ and $y$ element by element. That is, we want to compare `(car` $x$`)` and `(car` $y$`)` to see that they are equal; if they are, we can eliminate them and call `equal` recursively:

```
(equal (cdr x) (cdr y))
```

This is guaranteed eventually to reduce at least one of $x$ or $y$ to the null list, which is the case we have already handled. But how are we to compare `(car` $x$`)` and `(car` $y$`)`? We cannot use the `eq` function since it is defined only for atoms and `(car` $x$`)` or `(car` $y$`)` might be a list. What is needed is a function that will compare either atoms or lists for equality, and this is the very `equal` function we are defining. Therefore, we will call `equal` recursively on the `car`s of $x$ and $y$. This solves the problem when $x$ and $y$ are both lists:

if $x$ and $y$ are both lists then,

```
(and (equal (car x) (car y))
     (equal (cdr x) (cdr y)) )
```

In this case, the sequential `and` increases the efficiency of the program since if the `cars` of the lists are not equal, the program will not bother comparing the `cdrs` of the lists.

We now have two mutually disjoint cases since either at least one of $x$ and $y$ is an atom or they are both not atoms. Since we have all of the cases covered, we can combine them into a definition of the `equal` function:

```
(defun equal (x y)
  (or (and (atom x) (atom y) (eq x y))
      (and (not (atom x)) (not (atom y))
           (equal (car x) (car y))
           (equal (cdr x) (cdr y)) )) )
```

This function (which is provided by all LISP systems) is quite complicated to define without recursion. If you do not believe it, try it!

■ ***Exercise 10-10:*** Write a recursive function `count` to count the number of atoms in a list no matter what their level of nesting. For example,

```
(count '(a b (c 4) ((99)) nil t) )
  7
```

■ ***Exercise 10-11:*** Explain the behavior of `equal` when applied to circularly linked lists (i.e., lists pointing to a part of themselves).

## Recursion and Iteration Are Theoretically Equivalent

The difficulty of programming a function such as `equal` without recursion might lead us to suspect that recursion is more powerful than iteration, that is, that there are things that can be done recursively that cannot be done iteratively. In fact, this is *not* the case. Consider the stack implementation of recursion that we discussed in Chapter 6. There we saw that all the information relevant to a procedure call, such as its parameters, local storage, and result, could be maintained in a stack of activation records. In theory, any recursive program can be converted into an iterative program by maintaining such a stack. This is effectively what an Algol or Pascal compiler does; it converts recursion into iteration. Although this reduction is a *theoretical* possibility, it is not *practical* to do by hand in most cases, since the resulting program is so much more complicated. From the programmer's viewpoint, recursion *is* more powerful than iteration.

Since recursion can in principle be reduced to iteration, we might wonder if iteration is more powerful than recursion. This is also false. Earlier in this section, we showed how a number of common **while**-loop- and **for**-loop-like structures could be expressed, although you can probably see the general idea. More general structures expressible with **goto**s can also be reduced to recursion, but, like the reduction of recursion to iteration, this is not a result of much practical value.[2]

---

[2] The theoretical equivalence of iteration and recursion is explored in more detail in MacLennan (1990), Section 3.9.

■ ***Exercise 10-12:*** Write the `equal` procedure iteratively, that is, using **while**-loops. Maintain your own stack of intermediate results.

## Functional Arguments Allow Abstraction

Earlier in this section, we saw a definition of the `add1-map` function, which applies `add1` to each element of a list. In an exercise you programmed a `zerop-map` function that applies `zerop` to each element of a list. Notice that these functions were identical except for the particular function, `add1` or `zerop`, applied. Mapping any other unary function (e.g., `not`) would also fit the same pattern. We can see here an application of the Abstraction Principle: Since this same pattern keeps recurring, it is to our benefit to *abstract* it out and give it a name. This effectively raises the level at which we are programming and decreases the chances of error. If we repeat essentially the same thing over and over, there is a tendency to become careless and make a mistake. It is much better to do the thing once in a general way that can be used over and over.

Thus, we will define a function `mapcar` that applies a given function to each element of a list and returns a list of the results. It is not hard to see how to do this since the pattern of the recursion is just an abstraction from the patterns of `add1-map` and `zerop-map`:

```
(defun mapcar (f x)
  (if (null x)
      nil
      (cons (f (car x)) (mapcar f (cdr x)) )) )
```

The function is called `mapcar` because it applies `f` to the `car` of the list each time. Most LISP systems provide `mapcar` for the user, although on some the order of the arguments is reversed.

With this definition of `mapcar` our previous examples can be expressed without having to write a recursive definition:

```
(mapcar 'add1 '(1 9 8 4))
  (2 10 9 5)
(mapcar 'zerop '(4 7 0 3 -2 0 1))
  (nil nil t nil nil t nil)
(mapcar 'not (mapcar 'zerop '(4 7 0 3 -2 0 1)))
  (t t nil t t nil t)
```

■ ***Exercise 10-13:*** Notice that as we have defined the `mapcar` function, it will not work for *binary* functions so we cannot use it to write `plus-map`. Write a function `mapcar2` that can be used for this purpose. For example,

```
(mapcar2 'plus '(1 2 3 4) '(1 9 8 4))
  (2 11 11 8)
```

By using certain features of LISP that we will not discuss here, it is possible to define a `mapcar` function that takes a variable number of arguments so that it will work with any function. The `mapcar` provided by some LISPs works in this way.

■ ***Exercise 10-14:*** Define a function `filter` such that `(filter` $p$ $x$`)` is a list composed of just those elements of $x$ that satisfy the predicate $p$. For example,

```
(filter 'minusp '(2 -3 7 -1 -6 4 8))
   (-3 -1 -6)
```

■ ***Exercise 10-15:*** Define the `reduce` function that reduces a list by a given binary function.[3] For example,

```
(reduce 'plus 0 '(1 2 3 4 5))
   15
```

In general, `(reduce` $f$ $a$ $x$`)` means $(f\, x_1\ (f\, x_2 \ldots\ (f\, x_n\, a)\ \ldots))$.

■ ***Exercise 10-16:*** Show that `(reduce 'cons` $b$ $a$`)` is equivalent to `(append` $a$ $b$`)`.

## Functional Arguments Allow Programs To Be Combined

We have seen that one of the advantages of functional arguments is that they suppress details of loop control and recursion. Next we will see that they also simplify the combination of already implemented programs. The first example we will consider is the *inner product* of two lists.

The *inner product of two lists* is defined to be the sum of the products of the corresponding elements of the two lists:

$$u \cdot v = \sum_{i=1}^{n} u_i\, v_i$$

The `mapcar2` function that we defined in the preceding exercises can be used to take the products of the corresponding elements:

```
(mapcar2 'times u v)
   (w1 w2 ... wn)
```

where each $w_i = u_i v_i$. Therefore, the inner product is just the `plus` reduction of the products:

```
(define ip (u v)
   (reduce 'plus 0 (mapcar2 'times u v)) )
```

For example,

```
(ip '(1 2 3) '(-2 3 4))
   16
```

■ ***Exercise 10-17:*** Show that `ip` computes the result above by tracing its execution.

---

[3] This `reduce` function is similar, but not identical, to the `reduce` of Common LISP.

■ ***Exercise 10-18:*** Define, without explicit recursion, a function $f$ that computes

$$f(u,v) = \prod_{i=1}^{n} (u_i + v_i)$$

■ ***Exercise 10-19:*** Define, without explicit recursion, a function `pairlis` that computes

`(pairlis` $u\ v\ w$`)` =
$((u_1\ v_1)\ \ (u_2\ v_2)\ \ \ldots\ \ (u_m\ v_m)\ \ w_1\ w_2\ \ldots\ w_n)$

in which $u$ and $v$ are $m$ element lists and $w$ is an $n$ element list. That is, the function of `(pairlis` $u\ v\ w$`)` is to pair up corresponding elements of $u$ and $v$ and append them to the front of $w$. *Hint:* `(list` $u_i\ v_i$`)` returns the pair $(u_i\ v_i)$.

## Lambda Expressions Are Anonymous Functions

Suppose that we need to `cons` the value bound to `val` onto all the elements of a list `L`; this is an obvious application for `mapcar`. We can do this by writing `(mapcar 'consval L)` but only if we have already defined

```
(defun consval (x) (cons val x))
```

It is inconvenient to give a name to a function every time we want to pass it to `mapcar` (or to `reduce`, etc.). It clutters up the name space with short function definitions that are used only once. An obvious solution would be just to pass the function's body, `(cons val x)`, to `mapcar`:

```
(mapcar '(cons val x) L)
```

The trouble with this is that it's ambiguous; we do not know which names are the parameters to the `cons` and which are globals. That is, `(cons val x)` could equally well represent any of the following functions:

```
(defun f0 ()      (cons val x))
(defun f1 (x)     (cons val x))
(defun f2 (val)   (cons val x))
(defun f3 (val x) (cons val x))
(defun f4 (x val) (cons val x))
```

Notice that they all have exactly the same body—`(cons val x)`. What is required is some way of specifying which of the names in `(cons val x)` are parameters and which are global variables.

A mathematical theory called the "lambda calculus" provides a solution to this problem.[4] It supplies a notation for *anonymous functions*, that is, for functions that have not been bound to names. LISP uses the notation

```
(lambda (x) (cons val x))
```

[4] An introduction to the lambda calculus can be found in MacLennan (1990), Part II, and in many other books.

to mean that function of x whose value is `(cons val x)`. This is equivalent to the function `f1`. If `f2` were the function we wanted, we would write

```
(lambda (val) (cons val x))
```

These *lambda expressions* are values that can be manipulated like any other LISP lists; in particular, they can be passed as parameters.

Now we can solve our original problem. To cons `val` onto each element of a list `L` we write

```
(mapcar '(lambda (x) (cons val x)) L)
```

Similarly, to double all of the elements of `L`, we write

```
(mapcar '(lambda (n) (times n 2)) L)
```

Next we will consider a more complicated example of the use of lambda expressions. Recall that the application `(distl` $x$ $N$ `)` returns a list

$((x\ N_1)\ (x\ N_2)\ \ldots\ (xN_n))$

This is clearly a case of mapping a function on the list $N$, the function being that which takes any $y$ into the pair $(x\ y)$. Thus, the function we want to map is

```
(lambda (y) (list x y))
```

(The `list` function makes a list out of its arguments.) Hence, the definition of `distl` is

```
(defun distl (x N)
   (mapcar '(lambda (y) (list x y))
           N))
```

Now let's consider the `all-pairs` function. First, it is necessary to apply `(distl` $M_i$ $N$`)` for each element $M_i$ of $M$; this will give us a list $L$ of the form

$(L_1\ L_2\ \ldots\ L_n)$

where each $L_i$ is of the form

$((M_i N_1)\ (M_i N_2)\ \ldots\ (M_i N_n))$

The result we want, `(all-pairs` $M$ $N$`)`, is the result of appending all of the $L_i$, that is, the append-reduction of $L$:

```
(reduce 'append nil L)
```

The list $L$ results from forming a list of the results of `(distl` $M_i$ $N$`)` for each element of $M$. This can be accomplished with `mapcar`:

```
L = (mapcar '(lambda (x) (distl x N)) M)
```

All of these results can be assembled into the following definition of `all-pairs`:

```
(defun all-pairs (M N)
  (reduce 'append nil
          (mapcar '(lambda (x) (distl x N)) M)) )
(defun distl (x N)
  (mapcar '(lambda (y) (list x y))
          N))
```

Actually, the definition of `distl` is not necessary; it too can be replaced by a lambda expression:

```
(defun all-pairs (M N)
  (reduce 'append nil
          (mapcar '(lambda (x)
                     (mapcar '(lambda (y) (list x y))
                             N))
                  M)) )
```

This is probably carrying things too far, however. Even if the `distl` function is not used anywhere but in `all-pairs`, it is probably a good idea to give it a name so that the definition of `all-pairs` does not get too confusing. LISP expressions are like mathematical formulas: They must be kept small to stay readable.

- ***Exercise 10-20:*** Use a lambda expression to write an application to square each element of a list `L`.
- ***Exercise 10-21:*** Use lambda expressions, `append`, `cons`, and `reduce` to reverse a list `L`.

## Functionals Can Replace Lambda Expressions

We have seen that in many cases the use of functional arguments allows control-flow patterns to be abstracted out, given names, and used over and over again. We have also seen that the use of lambda expressions can often eliminate otherwise useless function bindings. Recently, programmers have begun using these ideas more systematically, a practice called *functional programming.* A *functional* is defined to be a function that has either (or both) of the following: (1) one or more functions as arguments; (2) a function as its result.[5] We have already seen several examples of functions that have functions as arguments, for example, `mapcar`, `reduce`, and `filter`.

Let's consider a few examples from earlier discussions. In our discussion of lambda expressions, we wanted to map onto a list a function that consed `val` onto its argument. We did this by mapping the lambda expression

```
(lambda (x) (cons val x))
```

[5] MacLennan (1990) is a comprehensive introduction. The term "functional" is used in a somewhat different sense in mathematics.

In the `all-pairs` example, we performed a similar action—mapping across a list a function that formed a list with `x`:

```
(lambda (y) (list x y))
```

In each case we turned a *binary* function (e.g., `cons`) into a *unary* function by *binding* one of its parameters to a value. In the first case, the first argument of `cons` was bound to `val`, and in the second case, the first argument of `list` was bound to `x`. Since we have changed one function into another function, we are really doing what a functional does—operating on a function to return another function. We can automate this process (applying the Abstraction Principle again) by defining a functional `bu` that converts a binary function into a unary function. In other words, instead of

```
(lambda (x) (cons val x))
```

we will write

```
(bu 'cons val)
```

In general, `(bu` $f$ $x$`)` binds the first argument of $f$ to $x$.

To define `bu` we have to be very clear about its effect. The result of `(bu` $f$ $x$`)` is a unary function that, when applied to an argument $y$, returns $(f\ x\ y)$. In other words,

`((bu` $f$ $x$`)` $y$ = $(f\ x\ y)$

[Notice that the function being applied to $y$ is the function returned by `(bu` $f$ $x$`)` ![6]] That function of $y$ whose value is $(f\ x\ y)$ is just

`(lambda (y)` $(f\ x\ y)$`)`

The definition of `bu` then follows immediately (the application of `function` is discussed in Section 10.2 on Name Structures; for now just interpret it as a sign that a function is being returned):

```
(defun bu (f x) (function (lambda (y) (f x y)) ))
```

The mapping onto `L` of the function to cons `val` onto a list can now be written without the use of lambda expressions or auxiliary function definitions:

```
(mapcar (bu 'cons val) L)
```

Similarly, the definition of `distl` can be simplified:

```
(defun distl (x N) (mapcar (bu 'list x) N) )
```

Notice that the use of the `bu` functional has eliminated an entire class of errors—mistakes in writing a lambda expression that converts a binary function to a unary function. This is one of the principal values of abstraction.

---

[6] Some LISP dialects, including Common LISP, do not permit using a function call as the function in another function call. In these dialects it would be necessary to write `(funcall (bu` $f$ $x$`)` $y$`)`.

There is also a lambda expression in the definition of `all-pairs`:

```
(lambda (x) (distl x N))
```

Here again a binary function is made into a unary function by fixing one of its parameters. In this case, however, it is the *second* parameter that is being fixed. The `bu` functional will not accomplish this since it fixes the *first* parameter. Of course, it would be no trouble to define a new functional, for example `bu2`, that fixes the second argument of a binary function.

Continuing this process would lead to a proliferation of functionals: `bu2`, `bu3`, and so on. We would not need this `bu2` functional if the arguments to `distl` were reversed; we could then use `bu`.

In functional programming it is quite common to need to reverse the arguments to a binary function, so the best solution seems to be to define a functional `rev` such that `(rev` *f*`)` is *f* with its arguments reversed. In other words, `(rev` *f*`)` applied to *x* and *y* yields `(`*f y x*`)`. That is, `(rev` *f*`)` is that function of *x* and *y* whose value is `(`*f y x*`)`:

```
(rev f) = (lambda (x y) (f y x))
```

The LISP definition follows easily:

```
(defun rev (f) (function (lambda (x y) (f y x)) ))
```

Now, fixing the second argument of `distl` is the same as fixing the first argument of `(rev 'distl)`, so the second argument of `distl` can be bound to `N` by

```
(bu (rev 'distl) N)
```

With this information, the definition of `all-pairs` can be written:

```
(defun all-pairs (M N)
  (reduce 'append nil
          (mapcar (bu (rev 'distl) N) M)) )
```

We can increase the flexibility with which functions can be combined if a uniform style is used for all functionals. We will define functional forms of `mapcar`, `mapcar2`, and `reduce` that, like `rev`, take a function and return a function. For example, `(map 'add1)` will be a function that adds 1 to each element of a list.[7] In other words

```
((map 'add1) L) = (mapcar 'add1 L)
```

Another way to say the above is that `(map` *f*`)` is that function that takes any list *L* into the result of mapping *f* onto that list:

```
(defun map (f) (function (lambda (L) (mapcar f L)) ))
```

The value of these functional operators is their *combinatorial power*, and for this reason they are often called *combinators* or *combining forms*. They make it simple to combine

[7] This `map` is different from the `map` in most LISP dialects, including Common LISP.

existing programs to accomplish new tasks. For example, to define a function `vec-dbl` that doubles all of the elements of a vector, we can use[8] `(map (bu 'times 2))`:

```
(set 'vec-dbl (map (bu 'times 2)) )
```

- ***Exercise 10-22:*** Define the functional `bu2`.
- ***Exercise 10-23:*** Define the functional `dup` so that `(dup` *f*`)` applied to *x* is `(`*f x x*`)`:

  `((dup` *f*`)` *x*`)` = `(`*f x x*`)`

Show that `(dup 'times)` is the squaring function.

- ***Exercise 10-24:*** Use functionals to define a function to square every element of a vector.
- ***Exercise 10-25:*** Define a functional that returns the *composition* of two functions `(comp` *f g*`)`. That is,

  `((comp` *f g*`)` *x*`)` = `(`*f* `(`*g x*`))`

- ***Exercise 10-26:*** Define a functional `map2` that when applied to a binary function *f* returns the corresponding binary map:

  `((map2` *f*`)` *x y*`)` = `(mapcar2` *f x y*`)`

- ***Exercise 10-27:*** Use functionals to define a function `vec-sum` that returns the sum of two vectors represented as lists. That is, it returns a list whose elements are the sum of the corresponding elements of the two input vectors.
- ***Exercise 10-28:*** Suppose that matrices are represented by lists of lists (i.e., vectors of vectors as in Pascal). Use functionals to define a function `mat-sum` that computes the sum of two matrices by adding their corresponding elements.
- ***Exercise 10-29:*** Define a functional `red` such that `(red` *f a*`)` is the *f*-reduction function starting with an initial value *a*. For example, `(red 'plus 0)` is the function that adds the elements of a list together.
- ***Exercise 10-30:*** Show that the following function computes the sum of the squares of the elements of a list:

  ```
  (comp (red 'plus 0) (map (dup 'times)) )
  ```

- ***Exercise 10-31:*** In a previous section, we defined an inner product function. Redefine this using functionals.
- ***Exercise 10-32:*** Define a functional `const` that returns a constant function. That is,

---

[8] Many LISP systems, including Common LISP, will not allow a function defined by `set` to be applied in the normal way. In these cases, the `expr` property must be defined explicitly by `(putprop 'vec-dbl (map (bu 'times 2)) 'expr)`.

`(const k)`, when applied to any argument, returns $k$. Show that the following function computes (albeit inefficiently) the length of a list:

```
(set 'length (comp (red 'plus 0) (map (const 1)) ))
```

■ ***Exercise 10-33*:*** *Discuss the readability of programs that make heavy use of functionals. What are the advantages and disadvantages? Suggest guidelines to improve the* readability of these programs.

## Backus Developed a Functional Programming Style

In 1977 John Backus, the principal designer of FORTRAN (Chapter 2) and of the BNF notation (Chapter 4), received the Association for Computing Machinery's Turing Award.

In his acceptance speech, Backus was highly critical of contemporary programming languages. He said that they "are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak." He proposed "an alternate functional style of programming . . . founded on the use of combining forms for creating programs."[9] Backus identified several areas in which programming languages could be improved.

First, he said that conventional languages have a "word-at-a-time" programming style. We have seen this in most of the languages we have studied, where, for example, an array is processed by performing some action on each of its elements individually, with all of the indexing, controlled variables, and loop control this requires. We have seen an alternative style of programming in this section. When we write `(map2 'plus)` to add two vectors, we can think of the vectors as wholes rather than being concerned with the details of iterating over their elements. Operations such as `mapcar` and `reduce` deal with *entire data structures* as units (although, of course, the machine will have to process all of the individual elements). Since we do not have to think about the parts of data structures, we are programming at a *higher level of abstraction*. One of the earliest languages to provide facilities like this was APL (early 1960s), which was the source for several of the functions we have discussed.

A second major goal of Backus's work has been to allow programs to be manipulated, proved, and even derived, by using simple algebraic manipulations. In other words, it should be about as simple to do things with programs as it is to do high school algebra. In this case, though, instead of adding, subtracting, multiplying, and dividing numbers, we are composing, reversing, mapping, and reducing functions. This is also referred to as *function-level* versus *object-level* programming, since the programmer manipulates functions rather than objects (i.e. data). To make these manipulations clearer, Backus introduced an algebraic notation that makes the structure of a program clearer than the Cambridge Polish of LISP; we will see examples later. Further, Backus observed that variables (i.e., formal parameters and Algol scope rules) complicate manipulating and reasoning about programs. Therefore, another goal of his notation has been to eliminate the need for variables; thus, his approach has sometimes been called *variable-free programming*. By this he did not mean just the elimination

[9] The history of and motivation for functional programming are discussed in more detail in MacLennan (1990).

of alterable variables and assignment statements; all applicative languages do this. He meant, in addition, the elimination of formal parameters. We have seen several examples of this, such as eliminating lambda expressions and their associated variables by using the `bu` and `rev` functionals.

Many of Backus's functionals correspond closely to the functionals we have discussed; they will serve as an introduction to his notation:

| Name | LISP | Backus |
|---|---|---|
| application | $(f\ x)$ | $f{:}x$ |
| mapping | (map $f$) | $\alpha f$ |
| reduction | (red $f$ $a$) | $/f$ |
| composition | (comp $f$ $g$) | $f \circ g$ |
| binding | (bu $f$ $k$) | (bu $f$ $k$) |
| constant | (const $k$) | $\overline{k}$ |
| lists | (a b c d) | ⟨a,b,c,d⟩ |
| built-in functions | plus, times, ... | +, ×, ... |
| selectors | cdr, car, cadr, ... | tail, **1**, **2**, ... |

Notice that there is only one functional, $\alpha$ (meaning "apply to all"), for mapping functions onto lists. Backus has avoided the need for a whole series of functionals `map2`, `map3`, and so on, and followed the Zero-One-Infinity Principle by a simple expedient: All functions are unary. Functions that we normally think of as having several arguments instead have one—a list of the arguments. For example, the '+' function takes a list as an argument—the list of numbers to be added; for example,

$$+ : \langle 3, 5 \rangle = 8$$

Let's work through an example. In a previous exercise, we defined an inner product function using functionals. We will do the same thing now in Backus's notation. The goal is to define a function ip such that ip: $\langle u, v \rangle$ is the inner product of the two vectors $u$ and $v$. The first step is to multiply the corresponding elements of the two vectors. But without `map2` how can we do this? The data structure we are given looks like this:

$$\langle\langle u_1, u_2, \ldots, u_n \rangle \langle v_1, v_2, \ldots, v_n \rangle\rangle$$

that is, a list of two lists. We need to get the corresponding $u_i$ and $v_i$ together, that is,

$$\langle\langle u_1, v_1 \rangle, \langle u_2, v_2 \rangle, \ldots, \langle u_n, v_n \rangle\rangle$$

which is a list of two-element lists. In other words, we need to convert a $2 \times n$ matrix into an $n \times 2$ matrix. For this purpose Backus provided a transpose function, trans:

$$\text{trans: } \langle\langle \ldots u_i \ldots \rangle, \langle \ldots v_i \ldots \rangle\rangle = \langle \ldots \langle u_i\ v_i \rangle \ldots \rangle$$

The next step is to apply the multiplication operation to each of these pairs; this is accomplished with $\alpha$, which corresponds to `map`:

$$(\alpha\times){:}(\text{trans: } \langle u, v \rangle) = \langle u_1 v_1, \ldots, u_n v_n \rangle$$

To compute the inner product, we just add up all of the products with a plus reduction, '/+' (the initial value 0 is implicit in Backus's notation):

$$\text{ip: } \langle u, v\rangle = (/+)\text{:}(\ (\alpha\times)\text{: (trans: } \langle u, v\rangle))$$

Recall that we want to eliminate variables; this can be done in this case with functional composition. Since

$$(f\circ g)\text{: } x = f\text{:}(g\text{:}x)$$

we can write the definition of the inner product:

$$\text{ip} = (/+) \circ (\alpha\times) \circ \text{trans}$$

Notice the distinctive characteristics of this program: It has no loops, no explicit sequencing, no assignments, and no variables.

For our next example, we will consider the multiplication of two matrices. Suppose $M$ is an $l \times m$ matrix and $N$ is an $m \times n$ matrix, and we want to form the product, $P$, an $l \times n$ matrix. The matrix product is defined as follows:

$$P_{ik} = \sum_{j=1}^{m} M_{ij}N_{jk}$$

The right-hand side of this should look familiar: It is just the inner product of the $i$th row of $M$ and the $k$th column of $N$. Now, the $k$th column of $N$ is just the $k$th row of the transpose of $N$. If we let

$$N' = \text{trans:}N$$

we can see that

$$P_{ik} = \text{ip: } \langle M_i, N_k'\rangle$$

where $M_i$ is the $i$th row of $M$ and $N_k'$ is the $k$th row of $N$ transpose.

How can we build up the product matrix $P$? Since it is structured as a list of lists, it is reasonable to construct it by two nested mappings, the inner one constructing the individual rows and the outer one combining these rows into the product matrix. The $i$th row of the product is

$$P_i = \langle P_{i1}, P_{i2}, \ldots, P_{in}\rangle$$

That is,

$$P_i = \langle\ \text{ip: } \langle M_i, N_1'\rangle, \ldots, \text{ip: } \langle M_i, N_n'\rangle\rangle$$

It is clear that this can be produced by a map:

$$P_i = (\alpha\ \text{ip})\text{:}\langle\langle M_i, N_1'\rangle, \ldots, \langle M_i, N_n'\rangle\rangle$$

Notice that the argument to $\alpha$ ip is a list of pairs and that the first elements of all of the pairs are the same. They can be factored out by Backus's *distribute left* function, which is defined

$$\text{distl: } \langle x, \langle a, b, \ldots, z\rangle\rangle = \langle\langle x,a\rangle, \langle x,b\rangle, \ldots, \langle x,z\rangle\rangle$$

Therefore,

$$\begin{aligned} P_i &= (\alpha \text{ ip}):(\text{distl}: \langle M_i, N' \rangle) \\ &= (\alpha \text{ ip}) \circ \text{distl} : \langle M_i, N' \rangle \end{aligned}$$

Let $f = (\alpha \text{ ip}) \circ \text{distl}$. Notice that the final matrix $P$ is

$$\begin{aligned} P &= \langle P_1, P_2, ..., P_l \rangle \\ &= \langle f:\langle M_1, N' \rangle, f:\langle M_2, N' \rangle, ..., f:\langle M_l, N' \rangle \rangle \end{aligned}$$

The function $f$ can be factored out by mapping:

$$P = (\alpha f):\langle \langle M_1, N' \rangle, \langle M_2, N' \rangle, ..., \langle M_l, N' \rangle \rangle$$

Since each pair ends in $N'$, each can be factored out with the *distribute right* function:

$$P = (\alpha f) \circ \text{distr}:\langle M, N' \rangle$$

This gives us the product matrix in terms of $M$ and $N$.′ How can we get it in terms of $M$ and $N$? For this the *constructor* functional can be used; it uses a sequence of functions to construct a list:

$$[f_1, ..., f_n]: x = \langle f_1 : x, ..., f_n : x \rangle$$

In our example,

$$\langle M,N' \rangle = [\mathbf{1}, \text{trans} \circ \mathbf{2} ]: \langle M,N \rangle$$

since $\mathbf{1}$: $\langle M,N \rangle = M$ (i.e., $\mathbf{1}$ is Backus's notation for `car`), and

$$\text{trans} \circ \mathbf{2} : \langle M,N \rangle = \text{trans}:N = N'$$

since $\mathbf{2}$ is Backus's notation for `cadr`. Therefore, the product of $M$ and $N$ is

$$\begin{aligned} P = (\alpha f) \circ \text{distr} \circ [ \mathbf{1}, \text{trans} \circ \mathbf{2} ]: \langle M,N \rangle \\ \text{where } f = (\alpha \text{ ip}) \circ \text{distl} \end{aligned}$$

In other words, a (variable-free) program for computing a matrix product is

$$\begin{aligned} \text{mat-prod} = (\alpha f) \circ \text{distr} \circ [ \mathbf{1} , \text{trans} \circ \mathbf{2} ] \\ \text{where } f = (\alpha \text{ ip}) \circ \text{distl} \end{aligned}$$

One of the goals of Backus's notation is the algebraic manipulation of programs. We can see a simple example of that here. If we substitute the definition of $f$ into mat-prod, we get

$$\text{mat-prod} = (\alpha ((\alpha \text{ ip}) \circ \text{distl})) \circ \text{distr} \circ [ \mathbf{1} , \text{trans} \circ \mathbf{2} ]$$

Since the doubly nested $\alpha$s are a little hard to read, we can use the identity

$$\alpha (f \circ g) = (\alpha f) \circ (\alpha g)$$

to simplify it. The resulting definition of the matrix product is

$$\text{mat-prod} = (\alpha \alpha \text{ ip}) \circ (\alpha \text{ distl}) \circ \text{distr} \circ [ \mathbf{1}, \text{trans} \circ \mathbf{2} ]$$

We can analyze this program for a matrix product. Some of the operations are inherent in the definition of a matrix product, such as the doubly mapped inner product and the transposition of the second matrix. Others, such as the distribute left and right operations, the selectors (**1**, etc.), and the constructor, are taking the place of parameters. We can see this by comparing Backus's program with a similar LISP program:

```
(defun mat-prod (M N) (mapcar (bu 'prod-row (trans N)) M))
(defun prod-row (Nt r) (mapcar (bu 'ip r) Nt))
```

The doubly mapped inner product and the transpose both appear here, but the other functions do not. Their purpose has been taken over by the bound variables, which are very powerful mechanisms for rearranging the order of things.

- ***Exercise 10-34*:*** Compare the Backus and LISP programs for the matrix product. What are their relative readability and writability? Which one do you suppose it is easier to prove things about? Separate out details issues of the notation (such as whether we write map or $\alpha$) from more fundamental issues (such as the absence of variables).
- ***Exercise 10-35:*** Write a program in Backus's notation to compute the mean of a list. Assume the function 'length', which returns the length of a list, is available.
- ***Exercise 10-36:*** Show that the following identity is true:

$$[f\circ\mathbf{1},\ g\circ\mathbf{2}]\circ[h,\ k] = [f\circ h,\ g\circ k]$$

- ***Exercise 10-37*:*** One of the advantages that Backus claims for his approach is that there is a *limited* set of functionals. Since LISP allows programmers to define their own functionals, it provides an *open-ended* set of functionals. Backus claims that a limited set is better because programmers can then master their use. Discuss the advantages and disadvantages between limited and open-ended sets of functionals.
- ***Exercise 10-38*:*** Read Backus's Turing Award Paper (in the August 1978 issue of the *Communications of the ACM*) and discuss it, including the ampliative and reductive aspects of this technology (Section 1.4).

## 10.2 DESIGN: NAME STRUCTURES

### The Primitives Bind Atoms to Their Values

As in the other programming languages we have discussed, the primitive name structures are the individual bindings of names to their values. In LISP these bindings are established in two ways—through property lists and through actual-formal correspondence. The former is established by pseudo-functions such as `set` and `defun`. For example, the application

```
(set 'text '(to be or not to be))
```

binds `text` to the list `(to be or not to be)` by placing the `apval` property with the value `(to be or not to be)` on the property list of `text`. Similarly, a `defun` binds a name to a function by placing the `expr` property on an atom's property list. This

property is bound to the lambda expression for the function. For example, the definition of `getprop` (p. 325) is equivalent to

```
(putprop 'getprop
  '(lambda (p x)
           (if (eq (car x) p)
               (cadr x)
               (getprop p (cddr x)) ))
  'expr)
```

The `set` and `defun` pseudo-functions are analogous to the variable and procedure declarations of a conventional programming language.

One important characteristic of bindings established through property lists is that they are *global*. Since there is at most one instance of each distinct atom in existence at a time, any change to the property list of an atom is visible throughout the program. The bindings are somewhat similar to the global subprograms and `COMMON` variables of FORTRAN (perhaps reflecting that LISP is almost an old as FORTRAN). We will see later that LISP also has a name structure constructor analogous to Algol's blocks.

## Application Binds Formals to Actuals

The other primitive binding operation is actual-formal correspondence. This is very similar to other programming languages. For example, if we define

```
(defun getprop (p x)
  (if (eq (car x) p)
      (cadr x)
      (getprop p (cddr x)) ))
```

and then evaluate the application

```
(getprop 'name DS)
```

the formal `p` will be bound to the atom `name` and the formal `x` will be bound to the value of the actual `DS` (which happens to be a *p*-list representing Don Smith's personnel record).

## Temporary Bindings Are a Simple, Syntactic Extension

We have seen two methods of binding names to values—the definition of properties and actual-formal correspondence. What we have not seen is a method of binding names in a local context such as is provided by Algol's blocks. To see the need for this, we will work out a simple example. Suppose we want to write a program to compute both roots of a quadratic equation. They are defined by the well-known equation

$$r_1, r_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

We want to define a function `roots` that returns a list containing the two roots. The general form of this function could be

```
(defun roots (a b c) (list r1 r2))
```

where $r_1$ and $r_2$ are expressions for computing the two roots. When these expressions are substituted in the above formula, we end up with a rather large, unreadable function definition:

```
(defun roots (a b c)
  (list (quotient (plus (minus b)
                        (sqrt (difference (expt b 2)
                                          (times 4 a c)) ))
                  (times 2 a))
        (quotient (difference (minus b)
                              (sqrt (difference (expt b 2)
                                                (times 4 a c)) ))
                  (times 2 a)) ))
```

Using the symbolic operator names provided by Common LISP and some other dialects is little improvement:

```
(defun roots (a b c)
  (list (/ (+ (- b) (sqrt (- (expt b 2) (* 4 a c)) ))
           (* 2 a))
        (/ (- (- b) (sqrt (- (expt b 2) (* 4 a c)) ))
           (* 2 a)) ))
```

One reason for this unwieldy expression (aside from the Cambridge Polish notation) is that the large expression corresponding to

$$\sqrt{b^2 - 4ac}$$

is repeated in each of the expressions $r_1$ and $r_2$. The usual way to avoid writing the same expression several times is to give it a name. For example,

$$\text{Let } d = \sqrt{b^2 - 4ac}$$
$$r_1 = (-b + d)/2a$$
$$r_2 = (-b - d)/2a$$

This obeys the Abstraction Principle, which says that an expression is more readable if the common parts are factored out.

How can we use the Abstraction Principle to improve our `roots` definition? What we need to do is bind a name (e.g., `d`) to the common subexpression *just long enough* to evaluate the expressions $r_1$ and $r_2$ for the two roots. The only mechanism we have seen for temporarily binding a name to a value is actual-formal correspondence. To use this, however, we will have to define an auxiliary function (e.g., `roots-aux`) to bind the formal `d`. This is what our definitions might look like:

```
(defun roots-aux (d)
  (list (quotient (plus (minus b) d)
```

```
                    (times 2 a))
          (quotient (difference (minus b) d)
                    (times 2 a)) ))

(defun roots (a b c)
  (roots-aux (sqrt (difference (expt b 2)
                               (times 4 a c)) )) )
```

Notice that we have not explicitly passed `a` and `b` to `roots-aux`; this is so because LISP functions are *called in the environment of the caller*, the `roots` function, in this case. This is discussed in more detail later.

The above definition of `roots` is simpler than our original one. However, it still leaves a lot to be desired because we have really misused a facility. We have used formal-actual correspondence where no parameterization was really involved; we have used it simply to introduce a local name. For this reason a number of LISP dialects (including Common LISP) provide a method of introducing local names analogous to our use of 'let' above. The `let` function allows a number of local names to be bound to values at one time; the second argument to `let` is evaluated in the resulting environment. That is,

(`let` $((n_1\ e_1) \ldots (n_m\ e_m))\ E$)

evaluates each of the expressions $e_i$ and binds the name $n_i$ to the corresponding value. The expression $E$ is evaluated in the resulting environment and is returned as the value of the `let`. For example,

```
(defun roots (a b c)
  (let ((d (sqrt (difference (expt b 2)
                             (times 4 a c)) )) )
       (list (quotient (plus (minus b) d)
                             (times 2 a))
             (quotient (difference (minus b) d)
                             (times 2 a)) )) )
```

We can see that the `let` function is something like an Algol block; it introduces a new scope and declares a set of names in that scope, each with an initial value. It is, in fact, closest to an Ada `declare` block that contains only constant declarations. That is,

```
declare
   n1: constant float := e1;
          :
          :
   nm: constant float := em;
begin
   E
end
```

The `let` function is really just an abbreviation for a function definition and application. That is,

(let ((*n*$_1$ *e*$_1$) ... (*n*$_m$ *e*$_m$)) *E*)

is equivalent to the definition

(defun QQQQQ (*n*$_1$ ... *n*$_m$) *E*)

(where QQQQQ is a made-up name that does not occur elsewhere) followed by the application

(QQQQQ *e*$_1$ ... *e*$_m$)

This is an example of *syntactic sugar*: The `let` construct does not allow us to do anything that could not be done before; rather, it allows us to write the same thing in a much more readable way. One of the important characteristics of LISP is that extensions such as the `let` function can be defined *within* LISP (although not with the facilities we've discussed). This makes it convenient for LISP programmers to solve their own problems without appealing to systems programmers. It also has the danger of encouraging the proliferation of "personal" LISP dialects.

## Dynamic Scoping Is the Constructor

We have already mentioned that LISP uses *dynamic scoping* instead of the static (or lexical) scoping used by the other languages we have studied. Recall (Chapter 3) that dynamic scoping means that a function is *called in the environment of its caller*, whereas static scoping means that it is *called in the environment of its definition.* For example, the `roots-aux` function had access to the names 'a' and 'b' because it was called from the `roots` procedure, which declared 'a' and 'b' as formals. In a statically scoped language, these would had to have been passed as parameters. The same is true for the `let` procedure: It would not be very useful if it did not *inherit* access to the names bound in the surrounding scope. Since a `let` is equivalent to a function application, the `let` must be invoked in the environment of the caller.

The above are some of the advantages of dynamic scoping. In Chapter 3, Section 3.3 (Algol Name Structures), we discussed some of the problems associated with dynamic scoping. We will not repeat that discussion here, but it will be worthwhile to reread it. Here we will discuss those problems peculiar to LISP.

## Dynamic Scoping Complicates Functional Arguments

Dynamic scoping can interact with functional arguments in confusing ways. Consider, for example, a functional `twice` defined so that (`twice` *f x*) returns the value of (*f* (*f x*)):

```
(defun twice (func val) (func (func val)) )
```

The call `(twice 'add1 5)` returns 7. If we want to double the number 3 twice, we can use a lambda expression

```
(twice '(lambda (x) (times 2 x)) 3)
  12
```

The state of the environment during the first application of `func` in `twice` is shown by the *contour diagram* (see Chapter 3) in Figure 10.1. Notice that the lambda expression to which `func` is bound has been *called in the environment of the caller*, in this case `twice`. This does not cause any problem, however.

Now consider a slightly different example. Instead of multiplying by 2 twice, we want to multiply by some predefined value named `val`. We have chosen `val` as the name of this value because it will *collide* with the formal of `twice`. Here is the example:

```
(set 'val 2)
  2
(twice '(lambda (x) (times val x)) 3)
  27
```

Since `val` is bound to 2, we should get the same answer as before, 12; but instead we get 27! To see why, we need to look at the contour diagram in Figure 10.2. Notice that the use of `val` in `(times val x)` has been intercepted by the dynamically inner binding of `val` as a parameter in `twice`. This is called a *collision of variables*. More specifically, we can see that the lambda expression

```
(lambda (x) (times val x))
```

is *vulnerable* to being called from an environment in which `val` is bound to something other than what was expected when we wrote it. We might object that we were foolish to use the name `val` for two different purposes, but in a large program it would be impractical to ensure that all of the names are distinct. Furthermore, we should not have to worry about the names of variables internal to the `twice` function; knowing these names violates *information hiding*.

This problem was discovered very early in the development of LISP and is known as the "funarg," or "functional argument" problem. It is really just an issue of dynamic versus static scoping. Rather than just using static scoping, however, a more expedient and ad hoc

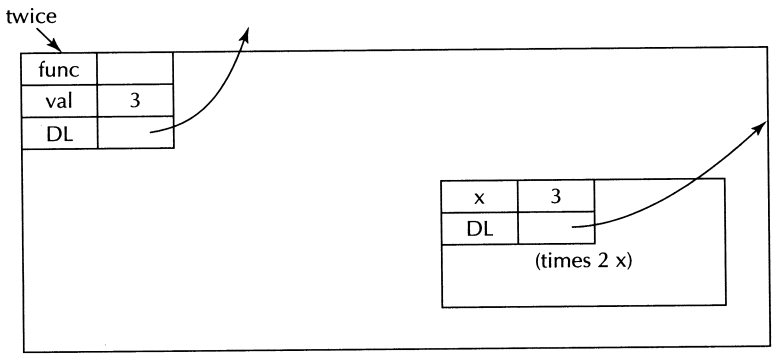


**Figure 10.1** Example of Call in the Environment of the Caller

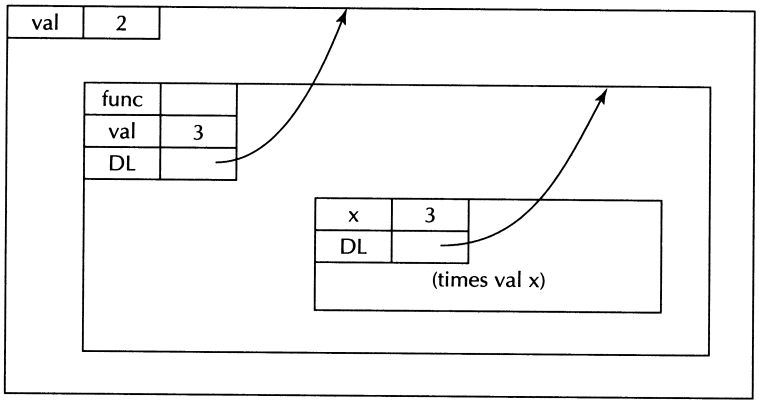


**Figure 10.2** Collision of Variables

solution was adopted. A special form called `function` was defined that binds a lambda expression to its *environment of definition.* Therefore, if we typed

```
(twice (function (lambda (x) (times val x)) ) 3)
  12
```

we would get the answer we expect. This is because `function` has bound the lambda expression to its environment of definition (in which `val` is bound to 2), so that when it is applied, it is *called in its environment of definition.* We can illustrate this in the contour diagram in Figure 10.3.

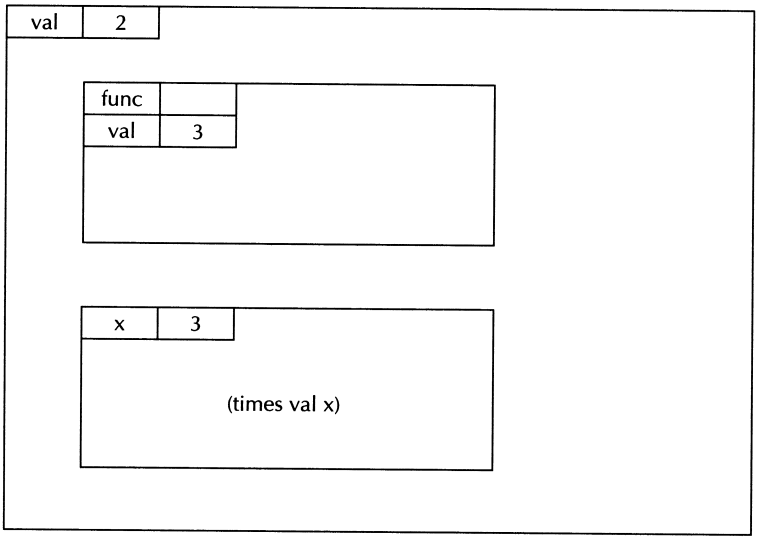


**Figure 10.3** Call in the Environment of Definition

It is for this same reason that we also used `function` in the definition of all of our functionals. Consider the definition of `bu`:

```
(defun bu (f x) (function (lambda (y) (f x y)) ))
```

The application of `function` binds the lambda expression to its environment of definition, which in this case includes the bindings of 'f' and 'x'. Without `function` the definition would be completely wrong since `bu` would return the lambda expression to the caller's environment, where 'f' and 'x' are either unbound or bound to the wrong values.

The effect of all of this is that LISP has *two* sets of scope rules: (1) it is dynamically scoped by default, and (2) it has static scoping accessible through the `function` construct. This certainly does not make for a simple, easy-to-understand language. Based on our previous discussions of scope rules, we might wonder why LISP was not simply made statically scoped. This probably would have been the simplest solution, but as often happened in LISP (and many other languages), people came to depend on the quirks of this "quick fix" so that it became part of the language. In recent years several statically scoped dialects of LISP have been designed (e.g., Scheme). In addition to solving the problems we have described, they are much easier to compile than the traditional LISP dialects.

- ***Exercise 10-39*:*** Defend one of the following three positions: (1) LISP should be statically scoped. (2) LISP should be dynamically scoped. (3) LISP should have both static and dynamic scoping.
- ***Exercise 10-40*:*** One of the goals of Backus's functional programming is to decrease the use of variables, particularly in lambda expressions. Discuss the impact of functionals on the LISP scoping problem.

## 10.3 DESIGN: SYNTACTIC STRUCTURES

### Much of LISP's Syntax Has Evolved

Recall that it was not the original intention of the *S*-expression notation that it be convenient for programming. Rather, this was assumed to be a temporary way of programming that would be replaced by the more conventional LISP 2 language. We discussed in Section 10.1 the unfortunate effect this had on the syntax of `cond`, which is designed more for the convenience of interpreters than of programmers.

Over the years there have been several improvements in LISP's syntax. For example, in the original LISP system the `define` function was used for establishing global bindings. The application

```
(define ((n1 e1) ... (nk ek)) )
```

would define each of the $n_i$ to be $e_i$. In the case of a function definition, the $e_i$ was a lambda

expression. The result is that our definition of `getprop` from Chapter 9, Section 9.3, would look like this:

```
(define ((getprop
           (lambda (p x)
                   (if (eq (car x) p)
                       (cadr x)
                       (getprop p (cddr x)) )) )) )
```

In common cases like this, there are many redundant parentheses. It is notation such as this that has led some people to claim that "LISP" stands for "Lots of Idiotic Single Parentheses"! The `defun` function is clearly an improvement on `define`, although it would be even better if it presented a *template* for application of the function. For example,

```
(defun (getprop p x)
  (if . . . .))
```

Another example of the evolution of LISP (and typical of all programming languages) is the `set` function. The original LISP system did not have the quote notation; for example,

```
'val
```

had to be written like an application:

```
(quote val)
```

This meant that a binding like `(set 'val 2)` had to be written

```
(set (quote val) 2)
```

Again, this was convenient for the interpreter but not for the programmer; look back through our examples to see how often quotes appear. Since the first argument to `set` is almost always quoted, a special form of `set` was defined, called `setq`, that did this implicitly. This permitted

```
(setq val 2)
```

Later, most LISP systems adopted the abbreviated form of quotation, which makes `setq` superfluous. Since most LISP programmers still use `setq`, however, most LISP dialects (including common LISP) still provide it.

■ ***Exercise 10-41*:*** Suggest other improvements to LISP's syntax that are still within the framework of the list notation.

## List Representation Facilitates Program Manipulation

The fact that LISP programs are represented as LISP data structures has some important advantages, which balance the reduction of readability. In particular, it is relatively simple to write LISP programs that read, preprocess, transform, and generate other LISP programs.

For example, if *A* is a list representing an application, then `(car A)` is the function being applied and `(cdr A)` is its actual parameter list:

```
(set 'A '(and (atom x) (atom y) (eq x y)) )
  (and (atom x) (atom y) (eq x y))
(car A)
  and
(cdr A)
  ((atom x) (atom y) (eq x y))
```

To write a Pascal program to separate the parts of a string representing a Pascal function call would be very complicated. The list representation makes it easier to get the important parts of LISP programs. For example, if *L* is a lambda expression, then `(cadr L)` is its list of bound variables and `(caddr L)` is its body:

```
(set 'L (lambda (x y) (f y x)) )
  (lambda (x y) (f y x))
(cadr L)
  (x y)
(caddr L)
  (f y x)
```

Although primitive in appearance, LISP syntax obeys the Elegance Principle. We will see in Chapter 11 that the list representation greatly simplifies writing a LISP interpreter in LISP, which was the original motivation for the list representation. More important, it has allowed LISP programmers to write high-level LISP programming aids and transformation tools easily and conveniently. One result of this is that some of the most advanced programming environments are provided by LISP systems.

■ ***Exercise 10-42*:*** Evaluate the representation of LISP programs as lists. Find a way to eliminate its annoying or poorly human-engineered aspects, while retaining its advantages for program manipulation.

## EXERCISES

**1.** Write a LISP function `(bvs f)` that returns the bound variables of a function *f*. The bound variables are part of the lambda expression that is the value of the `expr` property of the atom *f*.

**2*.** Write the `mapcar`, `reduce`, and `filter` functionals in a conventional programming language such as Pascal.

**3*.** Read about the APL programming language; compare it with the functional programming style discussed in this chapter.

**4*.** Ken Iverson is the inventor of APL. Read and critique Iverson's Turing Award Paper, "Notation as a Tool of Thought" (*Commun. ACM 23*, 8, August 1980).

**5*.** One solution to the problem of returning functions from functions (the "upward funarg problem") is described in Bobrow and Wegbreit's "A Model and Stack Implementation of Mul-

tiple Environments" (*Commun. ACM 16*, 10, October 1973). Evaluate the implementation described in this paper.

**6*.** Design a more readable syntax for LISP that still represents LISP programs as LISP lists.

**7*.** Program in LISP the `infix` function described in Chapter 9 (Section 9.3).

**8**.** The "`prog` feature" in LISP provides an imperative programming facility for LISP (complete with assignment statements and **goto**s). Should this feature be included in LISP? Write a report either attacking or defending the `prog` feature.