Chapter 4

Functional Programming

Most of the programming languages you are familiar with (Pascal, Ada, C) are **imperative** languages. They emphasize a programming style in which programs execute commands sequentially, use variables to organize memory, and update variables with assignment statements. The result of a program thus comprises the contents of all permanent variables (such as files) at the end of execution.

Although imperative programming seems quite natural and matches the execution process of most computer hardware, it has been criticized as fundamentally flawed. For example, John Backus (the designer of FORTRAN) holds that almost all programming languages (from FORTRAN to Ada) exhibit a "von Neumann bottleneck" in which programs follow too closely the "fetch instruction/update memory" cycle of typical CPUs. These languages do not lend themselves to simultaneous execution of different parts of the program, because any command may depend on the changes to variables caused by previous commands. (An enormous amount of effort has gone into creating algorithms that allow compilers to discover automatically to what extent commands may be executed simultaneously.) Execution speed is therefore ultimately limited by the speed with which individual instructions can be executed. Another effect of imperative programming is that to know the state of a computation, one must know the values of all the variables. This is why compilers that provide a postexecution dump of the values of all variables (or, better yet, compilers that allow variables to be examined and changed during debugging) are so handy.

In contrast, **functional** programming languages have no variables, no assignment statements, and no iterative constructs. This design is based on the concept of mathematical functions, which are often defined by separation into various cases, each of which is separately defined by appealing (possibly recursively) to function applications. Figure 4.1 presents such a mathematical definition.

On-line edition copyright © 1996 by Addison-Wesley Publishing Company. Permission is granted to print or photocopy this document for a fee of $0.02 per page, per copy, payable to Addison-Wesley Publishing Company. All other rights reserved.

Figure 4.1

```
f(n) =                                                    1
     1 if n = 1                                           2
     f(3*n+1) if n is odd, n ≠ 1                          3
     f(n / 2) if n is even                                4
```

In functional programming languages, such definitions are translated more or less directly into the syntax of the language. (The Miranda syntax is remarkably similar to this example.) The entire program is simply a function, which is itself defined in terms of other functions.

Even though there are no variables, there are identifiers bound to values, just as `n` is used in Figure 4.1. (When I use the term **variable**, I mean an identifier whose value can be changed by an assignment statement.) Identifiers generally acquire values through parameter binding. Variables are unnecessary in this style of programming because the result of one function is immediately passed as a parameter to another function. Because no variables are used, it is easy to define the effects (that is, the semantics) of a program. Often, functions are recursive. Functions have no side effects; they compute results without updating the values associated with variables. Functions are usually first-class values in functional programming languages. (First-class values are discussed in Chapter 3.)

The ML language, introduced in Chapter 3, is almost entirely functional. In that chapter, I concentrated on its type system, not on the way its lack of variables leads to a different programming style. This chapter presents examples in LISP, ML, and FP to give you a feeling for functional programming.

Functional programming is an area of current research. There is a biennial ACM Conference on LISP and Functional Programming.

1 ◆ LISP

LISP (List Processing language) was designed by John McCarthy at MIT in 1959. LISP actually represents a family of related languages, all sharing the common core of ideas first espoused in LISP 1.5. The most popular versions of LISP today are Scheme and Common LISP. Most dialects of LISP are not purely functional (variables are used sometimes, and certain functions do have side effects). I shall concentrate however on the functional flavor of programming in LISP.

The fundamental values manipulated by LISP are called **atoms**. An atom is either a number (integer or real) or a symbol that looks like a typical identifier (such as `ABC` or `L10`). Atoms can be structured into **S-expressions**, which are recursively defined as either

1. An atom, or
2. (S1.S2), where S1 and S2 are S-expressions.

Figure 4.2 shows some S-expressions.

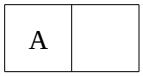
Figure 4.2

```
100                                                       1
(A.B)                                                     2
((10.AB).(XYZ.SSS))                                       3
```

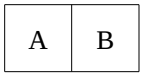
All S-expressions that are not atoms have two components: the head (called,

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

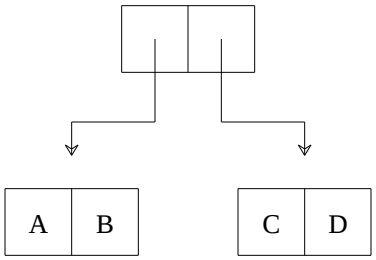
for historical reasons, the car), and the tail (called, for historical reasons, the cdr[1]). This definition leads to a simple runtime memory organization: numeric atoms are placed in one computer word, symbolic atoms are represented by a pointer to a symbol table entry, and S-expressions are represented by a pair of pointers to either atoms or subexpressions. Often, a box notation is used. The atom A is represented as follows:



(A.B) is represented as follows:



((A.B).(C.D)) is represented as follows:



The predefined symbol nil is used to represent the pointer to nothing. S-expressions represent the class of simple binary trees.

LISP provides a few predefined functions to assemble and disassemble S-expressions.

1. Car returns the head of a nonatomic S-expression. Thus car((A.B)) is A, car(((C.B).D)) is (C.B), and car(A) is undefined (because A is an atom). (These expressions are not syntactically correct LISP; I will introduce function syntax shortly.)
2. Cdr returns the tail of a nonatomic S-expression. Thus cdr((A.B)) is B, cdr(((C.B).D)) is D, and cdr(A) is undefined (because A is an atom).
3. Cons takes two S-expressions and builds a new S-expression composed of the two parameters. That is, cons(x,y) = (x.y) for any x and y (either atomic or not). Thus cons((A.B),C) = ((A.B).C). By definition, car(cons(x,y)) is x, and cdr(cons(x,y)) is y. Cons allocates space from the heap for the new cell that it needs.

Lists, the fundamental structured type in LISP, are a subset of the valid S-expressions. In particular,

---

[1] The term car stands for "contents of address register," and cdr stands for "contents of decrement register." These names refer to registers on the IBM 704 computer on which LISP was first implemented.

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

1. The empty list, `()`, is represented by the atom `nil`.
2. A list with one element, `(A)`, is equivalent to `cons(A,nil)`. A list `(A B)` is equivalent to `cons(A, cons(B,nil))`. In general, the list `(A B ... Z)` is equivalent to `cons(A, (B ... Z))`. That is, a list is extended by using `cons` to add an element to its left end.

Lists that contain lists are allowed, and in fact are frequently used. For example, `((A))` is the list that contains one element, namely the list `(A)`. `((A))` is created by first building `(A)`, which is `cons(A,nil)`. Then `(A)` is added to the empty list to make `((A))`. So `cons(cons(A,nil),nil)` generates `((A))`. Similarly, `((A B) () 11)`, which contains three elements, two of which are lists, is equal to the expression in Figure 4.3.

Figure 4.3

```
cons(cons(A,cons(B,nil)), cons(nil, cons(11,nil)))
```

The only difference is that the former expression is a literal (parsed and constructed by the LISP compiler/interpreter), and the latter is a combination of calls to runtime functions.

The Boolean values `true` and `false` are represented by the predefined atoms `t` and `nil`. Two fundamental predicates (that is, Boolean-returning functions) are `eq` and `atom`. `Eq` tests whether two atoms are the same (that is, equal). `Atom` tests whether a given S-expression is atomic.

1.1 Function Syntax

Programs as well as data are represented as lists. That is, LISP is **homoiconic:** Programs and data have the same representation. This property, rarely found in programming languages, allows a LISP program to create or modify other LISP functions. As you will see, it also allows the semantics of LISP to be defined in a particularly simple and concise manner. (Tcl, discussed in Chapter 9, is also homoiconic and enjoys the same benefits.)

To allow programs to be represented as lists, LISP function invocations aren't represented in the usual form of `FunctionName(arg1, arg2, ...)`, but rather as `(FunctionName arg1 arg2 ...)`. For example, the S-expression `(10.20)` can be built by evaluating `(cons 10 20)`.

When a list is evaluated, the first element of the list is looked up (in the runtime symbol table) to find what function is to be executed. Except in special cases (forms such as **cond**), the remaining list elements are evaluated and passed to the function as actual parameters. The value computed by the body of the function is then returned as the value of the list.

1.2 Forms

Should the call `(cons A B)` mean to join together the atoms `A` and `B`, or should `A` and `B` be looked up in the symbol table in case they are formal parameters in the current context? LISP evaluates all actual parameters, so `A` and `B` are evaluated by looking them up in the symbol table. If I want `A` and `B` to be treated as atoms rather than identifiers, I need to **quote** them, that is, prevent their evaluation. The programmer can use **quote**, called as `(quote arg)`,

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

to prevent evaluation. **Quote** is called a **form**, not a function,[2] because it is understood as a special case by the LISP interpreter. If it were a function, its parameter would be evaluated, which is exactly what **quote** is designed to prevent. The code in Figure 4.4 builds the S-expression (A.B).

Figure 4.4

```
(cons (quote A) (quote B))
```

Since programmers often need to quote parameters, LISP allows an abbreviated form of **quote**: 'A means the same as (**quote** A), so (cons 'A 'B) will also build the S-expression of Figure 4.4.

To be effective, any programming language needs some form of conditional evaluation mechanism. LISP uses the **cond** form. (Some dialects of LISP also provide an **if** form.) **Cond** takes a sequence of one or more pairs (lists of two elements) as parameters. Each pair is considered in turn. If the first component of a pair evaluates to t, then the second component is evaluated and returned as the value of **cond** (and all other pairs are ignored). If the first component evaluates to nil (that is, false), then the second component is ignored, and the next pair is considered. If all pairs are considered, and all first components evaluate to nil, then **cond** returns nil as its value.

As an example, suppose I want to create a predicate that tests whether some list bound to identifier L contains two or more elements. Figure 4.5 shows the code.

Figure 4.5

```
(cond                                                    1
     ((atom L) nil)                                      2
     ((atom (cdr L)) nil)                                3
     (t t)                                               4
)                                                        5
```

First, line 2 tests if L is an atom. If it is, it is the empty list (equal to nil), which certainly doesn't have two or more elements. Next, line 3 tests if cdr(L) is an atom. Cdr gives the list that remains after stripping off its first element. If cdr(L) is an atom, then the list had only one element, and the predicate again returns false. In all other cases, the list must have had at least two elements, so the predicate returns true. In most cases, the last pair given to **cond** has t as its first component. Such a pair represents a kind of **else** clause, covering all cases not included in earlier pairs.

1.3 Programmer-Defined Functions

Functions are first-class values in LISP (as in most functional programming languages). In particular, they can be returned as the result of functions. Therefore, LISP must allow the programmer to construct a function directly without necessarily giving it a name. The function constructor in LISP therefore builds **anonymous functions**, that is, functions that are not yet bound to names. To define a function, the programmer must provide a list containing three things: the form **lambda**, a list of the formal parameters, and the

[2] So you see that sometimes form is more important than function.

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

body of the function in the form of an expression. The anonymous function in Figure 4.6 makes a list with one element, passed in as a parameter.

Figure 4.6

```
(lambda (x) (cons x nil))
```

(The ML equivalent is `fn x => x :: nil`.) The formal parameter of the function is `x`. Parameters are passed in value mode. An implementation is likely to use reference mode and avoid copying; reference mode is safe to use because there are no commands that can change the parameters' values. Thus the function call of Figure 4.7

Figure 4.7

```
((lambda (x) (cons x nil)) 10)
```

binds `10` to the formal parameter `x`, yielding `(cons 10 nil)`, which is `(10)`. If more than one parameter is provided, they are all evaluated and bound, in left-to-right order, to the formal parameters. The expression in Figure 4.8, for instance,

Figure 4.8

```
((lambda (x y) (cons y x)) 10 20)
```

yields `(20.10)`. It is an error if too many or too few actual parameters are provided.

The anonymous function produced by the **lambda** form can be applied immediately (as I have been doing), passed as a parameter to a function, or bound to an identifier. Functions are bound to identifiers via the **def** form, which takes as parameters the function identifier and its definition (as a **lambda** form). Neither parameter should be quoted. Thus the expression in Figure 4.9

Figure 4.9

```
(def MakeList (lambda (x) (cons x nil)) )
```

defines the `MakeList` function, and `(MakeList 'AA) = (AA)`.

1.4 Scope Rules

The same identifier can be used as a function name or as a formal parameter in one or more functions. LISP therefore needs a scope rule to say which declaration is to be associated with each use of a symbol. Early dialects of LISP (in particular, LISP 1.5) used dynamic scoping: As actual parameters are bound to formal parameters, they are placed at the front of an association list that acts as the runtime symbol table for formal parameters. The association list is searched from front to back, so the most recent association of a value to a formal parameter is always found. If a formal parameter identifier appears more than once, the nearest (that is, most recent) binding of it is used. The order of call, and not static nesting, determines which declaration of a symbol is used. Consider Figure 4.10.

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

Figure 4.10

```
(def f1 (lambda (x y) (f2 11)))                                    1
(def f2 (lambda (x) (cons x y)))                                   2
(f1 1 2)                                                           3
```

When f1 is called, x is bound to 1, and y is bound to 2. Then f1 calls f2, which adds a new binding of 11 to x. Thus (cons x y) evaluates to (cons 11 2) = (11.2).

More recent dialects of LISP (including Common LISP and Scheme) use static scope rules, although Common LISP permits individual identifiers to be declared as dynamically scoped. Experience has shown that static scoping is much easier for the programmer to understand and is therefore less error-prone.

A program can also use the set function to change bindings in the association list, as in Figure 4.11.

Figure 4.11

```
(def f3 (lambda (x) (cons x (cons (set 'x 111) x))))               1
(f3 222)                                                           2
```

Formal parameter x is initially bound to 222 and becomes the first parameter to cons. Set binds 111 to x, and returns 111 as its value. The next appearance of x is now mapped to 111, and so LISP evaluates (cons 222 (cons 111 111)) = (222.(111.111)). If a symbol appears more than once in the association list, set updates its most recent binding. If a symbol isn't in the association list, it can't be bound using set.

LISP 1.5 has a more complicated scope rule. Each atom has a property list, which is a list (property name, value) pairs. An atom that has an APVAL property is evaluated to the associated value regardless of the contents of the association list. Function declarations are also stored in the property list under the EXPR property. If no EXPR property is present, the association list is searched, as shown in Figure 4.12.

Figure 4.12

```
(def f4 (lambda (x) (x 333 444) ))                                 1
(f4 'cons)                                                         2
```

When execution is in line 2, the body of the **lambda** form of line 1 is evaluated, and x's property list is searched for an EXPR entry. When none is found, the association list is tried. The binding of cons to x is found, so (cons 333 444) is evaluated.

The function get takes an atom and a property name and returns the value bound to that name on the atom's association list. If no binding is found, nil is returned.

1.5 Programming

Programming in LISP has a different flavor from programming in imperative languages. Recursion, rather than iteration, is emphasized. To perform a computation on a list, it is convenient to extract the first element of the list (using car), and then to recursively perform the computation on the remainder of the list.

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

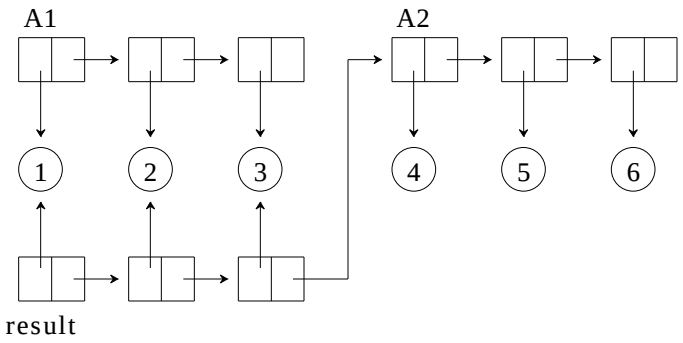
To give you an appreciation of this style of programming, I will present a few examples. First, I will create an Append function that appends two lists to form one. For example, (Append '(1 2 3) '(4 5 6)) = (1 2 3 4 5 6). (The quote is needed to prevent (1 2 3) from being treated as a function call.) I construct Append by considering cases. If the first list (call it L1) is empty (that is, equal to nil), then the result is the second list (call it L2). Otherwise, I add the first element of L1 to the list consisting of the remainder of L1 appended to L2. I therefore obtain the program shown in Figure 4.13.

Figure 4.13

```
(def Append (lambda (A1 A2) -- append lists A1 and A2          1
     (cond                                                     2
          ((null A1) A2)                                       3
          (t (cons (car A1) (Append (cdr A1) A2))))            4
))                                                             5
```

In line 3, null is a function that returns t only if its argument is nil. The list returned by Append is a curious mixture of newly allocated storage (cons always returns a new cell) and storage belonging to A1 and A2. Neither actual parameter is modified. The returned list contains new cells for all of A1's elements, the last of which points to the first cell for A2. Figure 4.14 shows the result of calling (Append '(1 2 3) '(4 5 6)).

Figure 4.14 Appending lists



The Append function can be programmed very similarly in ML, as shown in Figure 4.15.

Figure 4.15

```
val rec Append =                                               1
     fn (nil, A2) => A2                                        2
     |  (A1, A2) => (hd A1 :: Append (tl A1, A2));             3
```

The predeclared functions hd and tl are the same as LISP's car and cdr; the infix operator :: is the same as LISP's cons. Instead of using a conditional (ML has an **if** expression), I have chosen the more stylistic approach that uses patterns to distinguish cases. More sophisticated patterns allow me to avoid using hd and tl, as in Figure 4.16.

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

Figure 4.16

```
val rec Append =                                                    1
     fn (nil, A2) => A2                                             2
     |  (HA1 :: TA1, A2) => (HA1 :: Append (TA1, A2));              3
```

Next, I will build a LISP function that takes a list and returns its reversal. If the list is empty, the function returns the empty list; otherwise, the first element of the list will be the last element of the reversed list. I can make this element into a list (using `MakeList` defined earlier) then append it to the end of the reversal of the remainder of the list, arriving at the program shown in Figure 4.17.

Figure 4.17

```
(def Reverse (lambda (R) -- reverse list R                          1
     (cond                                                          2
          ((null R) R)                                              3
          (t (Append (Reverse (cdr R)) (MakeList (car R))))         4
     )                                                              5
))                                                                  6
```

The returned list is completely built out of new `cons` cells. The ML equivalent is given in Figure 4.18.

Figure 4.18

```
val rec Reverse =                                                   1
     fn nil => nil                                                  2
     | H :: T => Append(Reverse(T), [H]);                           3
```

As you can see, ML's ability to build formal parameters that correspond to components of the actual parameters and its syntax for list construction (`[H]` in line 3) give it a different feel from LISP, even though the underlying algorithm is identical.

`Reverse` only reverses the top-level elements of a list; if the elements are themselves lists, the lower-level lists aren't reversed. For example, `(Reverse '(1 (2 3 4) 5))) = (5 (2 3 4) 1)`. I can define a related function, `ReverseAll`, that reverses all lists, even if they appear as elements of another list; thus, `(ReverseAll '(1 (2 3 4) 5)) = (5 (4 3 2) 1)`. First, I define the reversal of any atom (including `nil`) as equal to that atom itself. Now when I append the `car` of a list onto the end of the reversal of the remainder of the list, I make sure to reverse the `car` first; thus, I have the code shown in Figure 4.19.

Figure 4.19

```
(def ReverseAll (lambda (RA) -- reverse RA and sublists             1
     (cond                                                          2
          ((atom RA) RA)                                            3
          (t (Append                                                4
               (ReverseAll (cdr RA))                                5
               (MakeList (ReverseAll (car RA)) ))))                 6
))                                                                  7
```

This example cannot be directly translated into ML, because ML's type scheme requires that lists be homogeneous. A programmer can, however, introduce a new ML datatype for nonhomogeneous lists; this idea is pursued in Exercise 4.13.

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

Figure 4.20 takes a list and doubles it; that is, it generates a list in which every member of the original list appears twice.

Figure 4.20

```
(def Double (lambda (L)                                    1
     (cond                                                 2
          ((null L) nil)                                   3
          (t (cons (car L) (cons (car L)                   4
               (Double (cdr L)))))                         5
     )                                                     6
))                                                         7
```

Double can be generalized in the same way as Reverse; Exercises 4.6 and 4.7 explore several generalizations.

Figure 4.21 builds Mapcar, which is itself very useful for building other functions. Mapcar takes two parameters, a function and a list, and returns the list formed by applying that function to each member of the list.

Figure 4.21

```
(def Mapcar (lambda (F L)                                  1
     (cond                                                 2
          ((null L) nil)                                   3
          (t (cons (F (car L)) (Mapcar F (cdr L))))        4
     )                                                     5
))                                                         6
```

I can use MapCar to take a list L of integers and return a list of their squares, as in Figure 4.22.

Figure 4.22

```
(MapCar (lambda (x) (* x x)) L)
```

As a final example, I will demonstrate a function Subsets that takes a set of distinct atoms (represented as a list) and creates the set of all possible subsets of the original set. That is, (Subsets '(1 2 3)) = (nil (1) (2) (3) (1 2) (1 3) (2 3) (1 2 3)). Because the lists represent sets, the order of the elements is unimportant, and any permutation of the list elements will be acceptable. Thus, (Subsets '(1 2 3)) could also return (nil (3) (2) (1) (2 1) (3 1) (3 2) (3 2 1))).

I first need a recursive definition of subset construction. That is, given a list representing all subsets of $\{1, 2, \cdots, n\}$, how can I create a list representing all subsets of $\{1, 2, \cdots, n+1\}$? It helps to notice that $Subsets(\{1, 2, \cdots, n+1\})$ will contain exactly twice as many elements as $Subsets(\{1, 2, \cdots, n\})$. Moreover, the extended set will contain all the elements of the original set plus n new sets created by inserting the element $n+1$ into each of the elements of the original set. For example, (Subsets '(1 2)) = (nil (1) (2) (1 2)). Therefore (Subsets '(1 2 3)) =

```
(Append (Subsets '(1 2)) (Distribute (Subsets '(1 2)) 3 ))
```

where (Distribute (Subsets '(1 2)) 3) = ((3) (3 1) (3 2) (3 1 2)). Finally, (Subsets nil) equals the list containing all subsets of the empty set, which is

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

(nil). I first define Distribute as shown in Figure 4.23.

Figure 4.23

```
(def Distribute (lambda (L E) -- put E in each elt of L          1
     (cond                                                       2
          ((null L) nil)                                         3
          (t (cons (cons E (car L))                              4
               (Distribute (cdr L) E))))                         5
))                                                               6
```

Distribute distributes element E through list L. If L is empty (line 3), there are no sets on the list to distribute E into, so Distribute returns nil. Otherwise (line 4), it takes the car of the list and conses E to it. It then joins the new list to the result of distributing E through the remainder of the list.

In Figure 4.24, I create an Extend function that extends a list L, which represents all subsets over n elements, to include element $n+1$, E. It does this by appending L to the list formed by distributing E through L.

Figure 4.24

```
(def Extend (lambda (L E) -- both L and L with E                 1
     (Append L (Distribute L E))                                 2
))                                                               3
```

Finally, I can define Subsets itself. The set of all subsets of the empty set (represented by nil) is the list containing only nil. For non-nil lists, I compute the list of all subsets of the cdr of the list, then extend it by adding in the car of the original list, obtaining the code in Figure 4.25.

Figure 4.25

```
(def Subsets (lambda (L) -- all subsets of L                     1
     (cond                                                       2
          ((null L) (MakeList nil))                              3
          (t (Extend (Subsets (cdr L)) (car L))))                4
))                                                               5
```

1.6 Closures and Deep Binding

Because LISP functions are represented as lists, functions can be passed as parameters to other functions and returned as the result of functions. In Figure 4.12 (page 109), it is important that cons be quoted in the call to f4, since I don't want it evaluated until its parameters are available.

Now consider a more interesting function, sc (self-compose), that takes a function and returns a new function representing the given function composed with itself. (That is, the new function has the effect of the old function applied twice.) I could write sc as shown in Figure 4.26.

Figure 4.26

```
(def sc (lambda (F) (lambda (x) (F (F x)))))
```

This code isn't quite right, because a call such as (sc car) will try to evaluate the resulting **lambda** form prematurely. If I quote the **lambda** form to obtain the code of Figure 4.27,

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

Figure 4.27

```
(def sc (lambda (F) '(lambda (x) (F (F x)))))
```

things still aren't right, because now the binding of F will be lost; by the time the internal **lambda** form is evaluated, sc has already returned, and its formal parameter has lost its meaning. I want to retain the binding of F until it is time to evaluate the **lambda** form returned by sc. To do this, I use a variant of **quote** called **function** and create sc as in Figure 4.28.

Figure 4.28

```
(def sc (lambda (F) (function (lambda (x) (F (F x))))))
```

Function creates a closure in which the bindings in effect when the **lambda** form is created are retained with the **lambda** form. The closure preserves the binding of identifiers that are nonlocal to a routine until the routine is executed. In other words, it produces a deep binding.

The Scheme dialect of LISP makes this example somewhat easier to code; see Figure 4.29.

Figure 4.29

```
(define (sc F)                                          1
     (lambda (x) (F (F x))))                            2

((sc car) '((a b) c)) -- returns 'a                     3
```

Scheme uses **define** as a shorthand that combines **def** and **lambda**; the **lambda** in line 2 introduces deep binding by returning a closure.

In some ways building a closure is harder in LISP than in statically scoped languages in which procedures are not first-class values. In statically scoped languages in which procedures are not first-class values, scope rules guarantee that an identifier cannot be referenced as a nonlocal in a part of the program that is lexically outside the scope defining the identifier. For example, assume that routine P, which is nested in routine Q, is passed as a functional parameter. All calls to P (either directly or as a parameter) must be completed before Q is terminated. Implementers can employ a simple stack of activation records (each of which contains the local data for a particular routine activation). A closure is a pointer to the code for a routine and a pointer to the proper activation record.

Chapter 3 introduced the dangling-procedure problem, in which the nonlocal referencing environment of a procedure has been deallocated from the central stack before the procedure is invoked. LISP encounters the same problem. In sc, references to F will occur when the result of sc is invoked as a function, which is after sc itself has returned. Unless special care is taken, the binding of the formal parameter F is no longer in force at that point. Deep binding solves the dangling-procedure problem by retaining a pointer in the closure returned by sc that points to sc's referencing environment, which includes the binding for F. Consequently, sc's referencing environment must be retained until all such outstanding pointers are deallocated. The result is that the referencing environments are linked together not as a simple stack, but as a treelike structure, with a new branch formed whenever a closure is created. Initially, the new branch is the same as the current association list, but they diverge as soon as the caller returns, removing the bindings of its lo-

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

cal parameters from the association list. Because cons cells and environment fragments have an indeterminate lifetime, most LISP implementations use garbage collection to reclaim free runtime store.

The alternative to deep binding is shallow binding, in which all nonlocal identifiers are resolved at the point of call. Under shallow binding, functions need never carry any bindings with them, because the only bindings that are used are those in effect when the function is actually evaluated. This simplification allows a simple stack of bindings to be used, but of course, functions such as sc must be implemented differently. One (rather ugly) way to define sc is to explicitly construct a function (using the list function, which makes a list out of its parameters) rather than to simply parameterize it. That is, I might code sc as in Figure 4.30.

Figure 4.30

```
(def sc (lambda (F)                                          1
     (list 'lambda '(x) (list F (list F 'x)))))              2
```

1.7 Identifier Lookup

Shallow and deep binding are also used (unfortunately, ambiguously) to denote two ways of implementing (as opposed to defining) identifier lookup in a dynamically scoped language such as early versions of LISP. I will call them shallow and deep search to avoid any confusion.

In block-structured languages with static scope rules, identifiers are translated to addresses (or offsets within an activation record) at compile time. In dynamically scoped languages like LISP, some runtime overhead to fetch the current binding (that is, value) of a symbol is to be expected, but this cost must be minimized to obtain reasonable performance. As you might expect, linear search through an association list every time an identifier is referenced is too inefficient to be practical.

A key insight is that an atom is actually represented as a pointer to its property list. It is possible to store the value associated with an atom in its property list, allowing fast access to the atom's value.

The question is, what happens when a given atom is re-bound; that is, the same identifier is re-bound as a formal parameter during application of a **lambda** form? A deep-search implementation places the original, or top-level, value of an atom in its property list. Re-bindings are pushed onto a runtime stack when an atom is re-bound. This stack must be searched when the current value of an atom is needed. (The first value found for that atom is the right one.) The name **deep search** is appropriate, since LISP must usually go deep into the stack to find out if an atom has been re-bound. The advantage of deep search is that creating and freeing new bindings is fairly efficient (and somewhat similar to pushing and popping an activation record in a conventional block-structured language).

Shallow search makes lookup faster by storing the most recent binding of an atom in its property list. Lookup is shallow indeed, but there is increased overhead in invoking and returning from functions. In particular, for each local identifier, the current value of that identifier (if there is one) must be saved on the runtime stack before the new binding is stored in the atom's property list. When a function returns, the last bindings pushed on the stack (if any) must be restored.

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

Deciding between deep and shallow search as an implementation technique therefore amounts to choosing whether to optimize identifier lookup or function invocation/return. The trend is toward shallow search, under the assumption that identifiers are referenced more often than functions are invoked and return. Tests show that in most cases shallow search does lead to faster execution.

As a final point, deep binding is compatible with shallow search. When a **function** form is evaluated, rather than copying the entire environment, the implementation copies only the bindings of selected nonlocal identifiers whose bindings it needs to preserve. This idea is similar to **import** statements found in imperative languages such as Modula-2. **Function** then creates a closure comprising the function body and the selected bindings. When a closure is invoked, the selected bindings are reinstated (almost like a second set of parameters), and then local bindings are created. Upon return, both local and deep bindings are removed.

1.8 The Kernel of a LISP Interpreter

It is possible to define a LISP interpreter in terms of a few primitive functions (car, cdr, cons, eq, atom, get, error, null), predefined identifiers (t, nil), forms (**cond**, **def**, **quote**), and metanotions of lambda binding and function application. An interpreter is a compact and exact specification of what any LISP program will compute. Few other languages can boast such a simple and elegant definition.

To simplify things, I will ignore fine points like deep binding, although deep binding can be handled without undue complexity. Whenever I invoke one of the primitive functions in the following functions, I assume that the result defined for that function is immediately computed, perhaps by a call to a library routine. Otherwise, the interpreter would encounter infinite recursion.

The interpreter is a function called Eval, shown in Figure 4.31.

Figure 4.31

```
(def Eval (lambda (List Env) -- evaluate List in Env                1
     (cond                                                          2
          ((null List) nil)                                         3
          ((atom List)                                              4
               (cond                                                5
                    ((get List (quote APVAL))                       6
                         (get List (quote APVAL)))                  7
                    (t (Lookup List Env))))                         8
          ((eq (car List) (quote quote)) (car (cdr List)))          9
          ((eq (car List) (quote cond))                             10
               (EvalCond (cdr List) Env))                           11
          (t (Apply (car List)                                      12
               (EvalList (cdr List) Env) Env)))                     13
))                                                                  14
```

Eval evaluates List in a given environment Env of identifier-value pairs. Values of atoms are looked up in their property lists (lines 6 and 7) or the environment Env (line 8). The forms **quote** (line 9) and **cond** (lines 10–11) are given special treatment. The eq function tests atoms for equality. (We don't

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

need to be concerned about what eq does with nonatoms; distinguishing pointer equality, shallow equality, and deep equality operations. These distinctions are discussed in Chapter 5.) All other lists are evaluated (lines 12–13) by applying the car of the list (a function) to a list of parameters evaluated in the current environment. Apply is defined as in Figure 4.32.

Figure 4.32

```
(def Apply (lambda (Fct Parms Env) -- apply Fct to Parms          1
     (cond                                                        2
          ((atom Fct) (cond                                       3
               ((eq Fct (quote car)) (car (car Parms)))           4
               ((eq Fct (quote cdr)) (cdr (car Parms)))           5
               ((eq Fct (quote cons))                             6
                    (cons (car Parms) (car (cdr Parms))))         7
               ((eq Fct (quote get))                              8
                    (get (car Parms) (car (cdr Parms))))          9
               ((eq Fct (quote atom)) (atom (car Parms)))         10
               ((eq Fct (quote error)) (error (car Parms)))       11
               ((eq Fct (quote eq))                               12
                    (eq (car Parms) (car (cdr Parms))))           13
               (t (cond                                           14
                    ((get Fct (quote EXPR))                       15
                         (Apply (get Fct (quote EXPR))            16
                         Parms Env))                              17
                    (t (Apply (Lookup Fct Env)                    18
                         Parms Env)))))                           19
          ) -- (atom Fct)                                         20
          ((eq (car Fct) (quote lambda))                          21
               (Eval (car (cdr (cdr Fct)))                        22
                    (Update (car (cdr Fct)) Parms Env)))          23
          (t (Apply (Eval Fct Env) Parms Env)))                   24
))                                                                25
```

If Fct is an atom (line 3), Apply first checks for each primitive function. If the atom isn't one of these, Apply checks its property list (lines 15–17), and then its association list Env (lines 18–19). This step can lead to an infinite recursion (that is, an undefined result) if Fct is a symbol bound to itself. If Fct is nonatomic, Apply looks for a **lambda** form (line 21). If it sees one, it binds the actual parameters to the formal **lambda** parameters (using Update), and then evaluates the **lambda** body in the updated environment, which is discarded afterward. If a nonatomic form isn't a **lambda** form, Apply attempts to simplify Fct by evaluating it, and then applying the simplified function to the original parameters (line 24). The remaining procedures, shown in Figure 4.33, are straightforward.

Figure 4.33

```
(def EvalCond (lambda (Conds Env) -- evaluate cond                1
     (cond                                                        2
          ((null Conds) nil) -- could treat as error              3
          ((Eval (car (car Conds)) Env)                           4
               (Eval (car (cdr (car Conds))) Env))                5
          (t (EvalCond (cdr Conds) Env)))                         6
))                                                                7
```

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

```
(def EvalList (lambda (List Env) -- evaluate list                    8
     (cond                                                           9
          ((null List) nil)                                          10
          (t (cons (Eval (car List) Env)                             11
               (EvalList (cdr List) Env))))                          12
))                                                                   13

(def Lookup (lambda (Id Env) -- lookup Id                            14
     (cond                                                           15
          ((null Env) (error (quote UnboundVar)))                    16
          ((eq Id (car (car Env))) (car (cdr (car Env))))            17
          (t (Lookup Id (cdr Env))))                                 18
))                                                                   19

(def Update (lambda (Formals Vals Env) -- bind parameters            20
     (cond                                                           21
          ((null Formals)                                            22
               (cond ((null Vals) Env)                               23
                     (t (error (quote ArgCount)))))                  24
          ((null Vals) (error (quote ArgCount)))                     25
          (t (cons (cons (car Formals)                               26
               (cons (car Vals) nil))                                27
               (Update (cdr Formals) (cdr Vals) Env)))               28
))                                                                   29
```

Many of the above functions assume that their parameters are syntactically well formed. For example, EvalCond (line 1) assumes Conds is a list of pairs. Similarly, most functions assume their parameters are lists, properly terminated by nil. A more careful interpreter would certainly check parameters (as does Update in lines 20–29).

The top level of many LISP implementations is an infinite loop:

```
(loop (Print (Eval (Read))))
```

The built-in Read function returns an expression typed in by the user, Eval derives a value from it, and Print displays that value. If the expression is a new function definition, it is treated as a top-level declaration and is added to the environment, so that later expressions can use it. Functions that are introduced within bodies of other functions are problematic, because if they modify the top-level environment, then function evaluation can have a side effect, which is not appropriate for a functional language. Scheme avoids this problem by forbidding function declarations except at the top level.

Any realistic LISP implementation would surely have more primitives than the above interpreter assumes. Arithmetic, debugging, and I/O functions are obvious omissions. Nevertheless, LISP has a remarkably small framework. To understand LISP one needs to understand lambda binding, function invocation, and a few primitive functions and forms (**cond**, for instance, is required). Everything else can be viewed as a library of useful predefined functions.

Consider how this situation contrasts with even so spartan a language as Pascal, which has a very much larger conceptual framework. Not surpris-

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

ingly, semantic definitions for imperative languages like Pascal are a good deal more complex than for LISP. Chapter 10 discusses the formal semantics of imperative languages.

The Eval function is also known as a **metacircular interpreter**. Such an interpreter goes a long way toward formally defining the semantics of the language. If a question arises about the meaning of a LISP construct, it can be answered by referring to the code of Eval. In a formal sense, however, metacircular interpreters only give one fixed point to the equation

```
Meaning(Program) = Meaning(Interpret(Program))
```

There are other fixed points (for example, that all programs loop forever) that aren't helpful in defining the semantics of a language. We will return to this subject in Chapter 10, which deals with the formal semantics of programming languages.

1.9 Run-time List Evaluation

Not only is Eval expressible in LISP; it is also provided as a predeclared function in every LISP implementation. Programmers can take advantage of the homoiconic nature of LISP to construct programs at runtime and then pass them as parameters to Eval.

For example, say I would like to write a function Interpret that accepts lists in the format of Figure 4.34.

Figure 4.34

```
'(MyAdd (MyAdd 1 5) (MyMult 2 3))
```

Here, MyAdd means "add the two parameters and then double the result," and MyMult means "multiply the two parameters and then subtract one from the result." The input to Interpret may be an arbitrarily nested list. One way to solve this puzzle is to program Interpret recursively. It would check to see if the car of its parameter was an atom, MyAdd, or MyMult, and then apply the appropriate arithmetic rule to the result of recursively interpreting the other parameters. But Figure 4.35 shows a much more straightforward, nonrecursive solution that takes advantage of Eval.

Figure 4.35

```
(def MyAdd (lambda (A B) (* (+ A B) 2)))
(def MyMult (lambda (A B) (- (* A B) 1)))
(def Interpret (lambda (L) (Eval L)))
(Interpret '(MyAdd (MyAdd 1 5) (MyMult 2 3))) -- result is 34
```

The list to be interpreted is treated not as data, but as program, and Eval is capable of executing programs.

1.10 Lazy Evaluation

Normally, a LISP evaluator operates by evaluating and binding actual parameters to formal parameters (first to last) and then evaluating function bodies. If an actual parameter involves a function call, that function is invoked as the parameter is evaluated. This strategy is known as **strict evaluation**. Given

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

the functional nature of LISP programs, other evaluation strategies are possible.

One of the most interesting of these is **lazy evaluation**. As the name suggests, a lazy evaluator only evaluates an expression (typically, an actual parameter) if it is absolutely necessary. Evaluation is performed incrementally, so that only those parts of an expression that are needed are evaluated. For example, if only the `car` of an S-expression is needed, the `cdr` is not yet evaluated.

One form of lazy evaluation that is common even in imperative programming languages is short-circuit semantics for Boolean operators, as discussed in Chapter 1. In imperative languages, short-circuit evaluation can change the meaning of a program, because a subexpression could have a side effect (an assignment hidden in a function call, for example) that is avoided by not evaluating that subexpression. In functional languages, there are no side effects, so there is no danger that short-circuit evaluation will change the semantics. The order of evaluation of expressions (and subexpressions) is irrelevant. This freedom to evaluate in any order makes functional languages particularly fertile ground for generalizing the idea of short-circuit semantics. (The **cond** form requires care to make sure that the textually first successful branch is taken. Although the branches can be evaluated in any order, runtime errors encountered in evaluating conditions textually later than the first successful one need to be suppressed.)

An expression that is not yet evaluated is called a **suspension**. Suspensions are much like closures; they combine a function and a referencing environment in which to invoke that expression. They also include all the unevaluated parameters to that function. When a suspension is evaluated, it is replaced by the computed value, so that future reevaluations are not needed. Often, that computed value itself contains a suspension at the point that evaluation was no longer needed.

Lazy evaluation is of interest primarily because strict evaluation may evaluate more than is really needed. For example, if I want to compute which student scored the highest grade in an exam, I might evaluate `(car (sort Students))`. Strict evaluators will sort the entire list of students, then throw all but the first element away. A lazy evaluator will perform only as much of the sort as is needed to produce the `car` of the list, then stop (because there is no reference to the `cdr` of the sorted list). Now, sorting in order to find the maximum element is an inefficient approach to begin with, and we can't fault strict evaluators for inefficiency when the algorithm itself is so bad. However, lazy evaluation manages to salvage this inefficient (but very clear) approach and make it more efficient.

As a more detailed example, consider trees encoded as lists. For example, `((A B) (C D))` represents a binary tree with two binary subtrees. The frontier (or fringe) of a tree is the list of leaves of the tree (in left-to-right order). The frontier of this particular tree is `(A B C D)`. I want to determine if two trees have the same frontier. An obvious approach is to first flatten each tree into its frontier, then compare the frontiers for equality. I might write the code in Figure 4.36.

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

Figure 4.36

```
(def SameFrontier (lambda (X Y)                                    1
     (EqualList (Flatten X) (Flatten Y))))                         2

(def EqualList (lambda (X Y)                                       3
     (cond                                                         4
          ((null X) (null Y))                                      5
          ((null Y) nil)                                           6
          ((eq (car X) (car Y))                                    7
               (EqualList (cdr X) (cdr Y)))                        8
          (t nil))                                                 9
))                                                                 10

(def Flatten (lambda (List)                                        11
     (cond                                                         12
          ((null List) nil)                                        13
          ((atom List) (MakeList List))                            14
          (t (Append (Flatten (car List))                          15
               (Flatten (cdr List)))))                             16
))                                                                 17
```

Calls to SameFrontier (assuming a strict evaluation mechanism) will flatten both parameters before equality is ever considered. This computation will be particularly inefficient if the trees are large and their frontiers have only a small common prefix.

Lazy evaluation is more appropriate for such a problem. It follows an outermost-first evaluation scheme, postponing parameter evaluation until necessary. That is, in a nested invocation, such as that of Figure 4.37,

Figure 4.37

```
(foo (bar L) (baz (rag L)))
```

foo is invoked before bar or baz, and in fact they may never be invoked at all, if, for example, foo ignores its parameters. If foo needs to evaluate its second parameter, baz is invoked, but not rag, unless baz itself needs it. Furthermore, once a function has been invoked, the result it returns may not be completely computed. For example, the body of bar may indicate that it returns (cons 1 (frob L)). It will return a cons cell (allocated from the heap) with 1 in the car and a suspension in the cdr; the suspension indicates that a frob must be invoked on L in order to achieve a value. This suspension may never be activated.

The algorithm for lazy evaluation is as follows:

1. To evaluate a list, make a suspension out of it (combining the function name, the parameters, which are not to be evaluated yet, and the referencing environment).
2. To evaluate a suspension, make a suspension out of each of its parameters and invoke its function in its referencing environment.
3. To evaluate a cons invocation, create a new cons cell in the heap and initialize its car and cdr to the parameters, which are left as suspensions.
4. To evaluate a primitive Boolean function such as null or eq, evaluate the parameter(s) only as far as needed. Each primitive function has its own lazy evaluation method.

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

Let me trace how a lazy evaluator might evaluate

```
(SameFrontier '((A B) C) '(B C (A D))) .
```

The trace in Figure 4.38 shows the evaluation steps.

Figure 4.38

```
Goal = (SameFrontier S1='((A B) C) S2='(B C (A D)))                          1
=[S body] (EqualList E1=(Flatten S1) E2=(Flatten S2))                        2
=[E body] (cond ((null E1) .. ) .. ) .. )                                    3
|    E1 = (Flatten F1=S1)                                                    4
|    =[F body] (cond ((null S1) .. ) .. )                                    5
|    [S1 is neither null nor an atom]                                        6
|    = (Append A1=(Flatten (car F1))                                         7
|           A2=(Flatten (cdr F1)))                                           8
|    =[A body] (cond ((null A1) .. ) .. )                                    9
|    |    A1 = (Flatten F2=(car F1))                                         10
|    |    =[F body] (cond ((null F2) .. ) .. )                               11
|    |    |    F2 = (car F1) = (car S1) = (car '((A B) C))                   12
|    |    |    = '(A B)                                                      13
|    |    [F2 is neither null nor an atom]                                   14
|    |    A1 = (Append A3=(Flatten (car F2))                                 15
|    |           A4=(Flatten (cdr F2)))                                      16
|    |    =[A body] (cond ((null A3) .. ) .. )                               17
|    |    |    A3 = (Flatten F3=(car F2))                                    18
|    |    |    =[F body] (cond ((null F3) .. ) .. )                          19
|    |    |    |    F3 = (car F2) = (car '(A B)) = 'A                        20
|    |    |    [F3 is not null, but it is an atom]                           21
|    |    |    A3 = (MakeList F3) =[M body] (cons F3 nil)                    22
|    |    |    = '(A)                                                        23
|    |    [A3 is not null]                                                   24
|    |    A1 = (cons (car A3) (Append (cdr A3) A4))                          25
|    [A1 is not null]                                                        26
|    E1 = (cons (car A1) (Append (cdr A1) A2))                               27
[E1 is not null]                                                             28
Goal = (cond ((null E2) .. ) .. )                                            29
|    E2 = (Flatten F4=S2)                                                    30
|    =[F body] (cond ((null F4) .. ) .. )                                    31
|    [F4 is not null or an atom]                                             32
|    = (Append A5=(Flatten (car F4))                                         33
|           A6=(Flatten (cdr F4)))                                           34
|    =[A body] (cond ((null A5) .. ) .. )                                    35
|    |    A5 = (Flatten F5=(car F4))                                         36
|    |    =[F body] (cond ((null F5) .. ) .. )                               37
|    |    |    F5 = (car F4) = (car S2)                                      38
|    |    |    = (car '(B C (A D))) = 'B                                     39
|    |    [F5 is not null, but it is an atom]                                40
|    |    A5 = (MakeList F5) =[M body] (cons 'B nil) = '(B)                  41
|    [A5 is not null]                                                        42
|    E2 = (cons (car A5) (Append (cdr A5) A6))                               43
```

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

```
[E2 is not null]                                                    44
Goal = (cond ((eq (car E1) (car E2)) .. ) .. )                      45
= (cond ((eq (car A1) (car A5)) .. ) .. )                           46
= (cond ((eq (car A3) 'B) .. ) .. )                                 47
= (cond ((eq 'A 'B) .. ) .. )                                       48
= (cond (t nil))                                                    49
= nil -- frontiers are different                                    50
```

The notation is concise, but I hope not too clumsy. I show that a formal parameter is bound to an actual parameter by the notation Formal=Actual, as in S1='((A B) C) in line 1. The names of the formal parameters (here, S) start with the same letter as the name of the function (SameFrontier). I distinguish multiple parameters as well as new instances during recursive calls by numeric suffixes. Simplification steps are marked in various ways. In line 2, =[S body] means expanding a call by inserting the body of a function, in this case, SameFrontier. I have used the declaration of Append from Figure 4.13 (page 110) whenever [A body] is mentioned. The ellipsis (..) shows where evaluation of **cond** pauses in order to evaluate a subexpression. It only evaluates the subexpression to the point that it can answer the condition in question. For example, in line 9, it is necessary to discover if A1 is null. By line 25 A1 has been evaluated enough to answer the question, as reported in line 26. These subordinate evaluations are indented. Line 27 continues the evaluation of E1 started on line 4. It leaves the result in terms of A1 and A2, to be further evaluated in lines 46–48.

Lazy evaluation is more difficult (and costly) to implement than strict evaluation. The example shows that it has much of the flavor of coroutines (see Chapter 2), with control automatically shifting as needed among many computations in order to advance the computation.

An implementation of LISP might allow the programmer to select lazy evaluation when desired; any evaluation strategy will produce the same result so long as the program is written in "pure" LISP. (Some dialects include imperative facilities, which, as you have seen, can make the evaluation order significant.) Automatic determination of the preferable evaluation strategy is an open (and hard) problem.

Lazy evaluation is sometimes called "demand-driven evaluation" because evaluation is triggered by a demand for a value. We conventionally view a computation (very roughly) as first obtaining input values, then computing a result using them, and finally printing that result. Demand-driven evaluation reverses this view. Nothing happens until the evaluator sees a request to write a result. This request initiates computations, which solicit input values. If no demand for output is seen, nothing is computed.[3]

Lazy evaluation also has a declarative, nonprocedural flavor. (Chapter 8 discusses logic programming, which is declarative.) Although LISP is certainly procedural (both imperative and functional languages are in the larger category of procedural languages), lazy evaluation makes evaluation optional.

[3] This is similar to a folk myth concerning the benchmarking of an optimizing FORTRAN compiler. The compiler was presented with a very complex program containing no write statements. It optimized the program by generating no code!

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

That is, an expression is not a command, "Compute this!" but a suggestion as to how to obtain a value if it is needed.

Imperative languages also allow unnecessary computations to be suppressed. For example, optimizing compilers often eliminate "dead code." Since imperative languages are full of side effects, delaying major calculations for extended periods of time is quite difficult. Lazy evaluation is much more attractive in a functional programming-language environment.

1.11 Speculative Evaluation

Another interesting evaluation strategy is **speculative evaluation**. As the name suggests, a speculative evaluator wants to evaluate as much as possible, as soon as possible. This evaluation strategy is best suited for multiprocessors or multicomputers that are able to perform many calculations concurrently. Present multicomputers have hundreds of processors; future machines may have hundreds of thousands or even millions.

A crucial problem in a multicomputer is finding a way to keep a reasonable fraction of the processors busy. Speculative evaluation seeks to evaluate independent subexpressions concurrently. For example, in an invocation of `SameFrontier`, a speculative evaluator could flatten both lists concurrently. Within a function, another source of potential concurrency lies in the evaluation of a **cond** form. Individual guards of a **cond** can be evaluated concurrently, as well as their associated bodies.

Care is required because of the evaluation ordering that is assumed in **cond**'s definition. Evaluation of a subexpression may lead to a runtime error (for example, taking the `car` of an atom), because a speculative evaluator will evaluate an expression that a strict evaluator would never examine. With care, faults can be suppressed until their effect on the overall result is known. Given this caveat, a **cond** form can be a rich source of concurrent evaluations.

Nonetheless, the cost of starting a processor and later receiving its result is often high. If the calculation started speculatively is too small, the overhead will overshadow any advantage provided by the concurrent evaluation. A speculative evaluator for LISP would probably evaluate primitive functions directly and reserve concurrent speculative evaluation for **lambda** forms. Such coarse-grain parallelism is discussed further in Chapter 7.

The ability to evaluate expressions in virtually any order makes speculative evaluation plausible for functional programming languages. In imperative languages, an elaborate analysis of what variables depend on what other variables is required even to consider any form of concurrent evaluation. Once again the von Neumann bottleneck rears its ugly head.

1.12 Strengths and Weaknesses of LISP

Functional programming is in many ways simpler and more elegant than conventional programming styles. Programmers do not need to keep track of potential side effects when a procedure is invoked, so programming is less error-prone. The lack of side effects allows implementations a rich variety of evaluation strategies.

LISP and its descendants have long been the dominant programming languages in artificial intelligence research. It has been widely used for expert systems, natural-language processing, knowledge representation, and vision

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

modeling. Only recently has Prolog, discussed in Chapter 8, attracted a significant following in these areas. LISP is also the foundation of the widely used Emacs text editor. Much of LISP's success is due to its homoiconic nature: A program can construct a data structure that it then executes. The semantics of the core of LISP can be described in just a few pages of a metacircular interpreter.

LISP was the first language to have an extensive program development environment [Teitelman 81]. (Smalltalk, described in Chapter 5, was the second. Such environments are widely available now for Ada, Pascal, and C++.) Programs can be modified and extended by changing one function at a time and then seeing what happens. This facility allows elaborate programs to evolve and supports rapid prototyping, in which a working prototype is used to evaluate the capabilities of a program. Later, the program is fleshed out by completing its implementation and refining critical routines.

The most apparent weakness of the early dialects of LISP is their lack of program and data structures. In LISP 1.5, there are no type-declaration facilities (although some LISP dialects have adopted facilities for data typing). Certainly not everything fits LISP's recursive, list-oriented view of the world. For example, symbol tables are rarely implemented as lists.

Many LISP programmers view type checking as something that ought to be done after a program is developed. In effect, type checking screens a program for inconsistencies that may lead to runtime errors. In LISP, type checking amounts generally to checking for appropriate structures in S-expressions.

Most production LISP dialects (such as Interlisp, Franz LISP, Common LISP, and Scheme) have greatly extended the spartan facilities provided in LISP 1.5, leading to incompatibilities among LISP implementations. Indeed, it is rare to transport large LISP programs between different implementations. This failure inhibits the interchange of software tools and research developments.

It would appear that the corrupting influences of von Neumann programming are so pervasive that even functional languages like LISP can succumb. Most LISP implementations even have a `prog` feature that allows an imperative programming style! In addition, LISP has some decidedly nonfunctional features, such as the `set` function and property lists. In fact, it has been said that "LISP ... is not a functional language at all. [The] success of LISP set back the development of a properly functional style of programming by at least ten years." [Turner 85b]

2 ◆ FP

In comparison to typical block-structured languages, LISP 1.5 stands as a paragon of simplicity. (On the other hand, Common LISP is as big as Ada.) Nonetheless, Backus suggests that even simpler functional programming approaches may be desirable [Backus 78]. He thinks that LISP's parameter-binding and substitution rules are unnecessary and instead proposes a variable-free programming style limited to single-parameter functions. (A parameter may, however, be a sequence, and functions may be curried.) Further, LISP's ability to combine functions in any form (since functions are just S-expressions) is unnecessarily general. He compares this freedom to the

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

unrestricted use of **goto** statements in low-level imperative languages. In contrast, Backus prefers a fixed set of higher-order functions that allow functions to be combined in various ways, analogous to the fixed set of control structures found in modern imperative languages. The result is the FP programming language.

2.1 Definition of an FP Environment

An FP environment comprises the following:

1. A set of objects. An object is either an atom or a sequence, $<x_1, \ldots, x_n>$, whose elements are objects, or $\perp$ ("bottom") representing "error," or "undefined." Included as atoms are ϕ, the empty sequence (roughly equivalent to `nil` in LISP), and `T` and `F`, representing `true` and `false`. Any sequence containing $\perp$ is equivalent to $\perp$. That is, the sequence constructor is bottom-preserving.
2. A set of functions (which are not objects) mapping objects into objects. Functions may be primitive (predefined), defined (represented by a name), or higher-order (a combination of functions and objects using a predefined higher-order function). All functions are bottom-preserving; `f` applied to $\perp$ always yields $\perp$.
3. An application operation that applies a function to an object, yielding an object. Function `f` applied to object `x` is denoted as `f:x`. Here, `x` isn't a variable name (there are no variables!), but rather a placeholder for an expression that will yield an object.
4. A set of higher-order functions used to combine existing functions and objects into new functions. Typical higher-order functions include those shown in Figure 4.39.

Figure 4.39

```
Composition                                                    1
(f ∘ g):x ≡ f:(g:x)                                            2

Construction                                                   3
[f1 , ... , fn ]:x ≡ <f1:x, ... ,fn:x>                         4

Condition                                                      5
(p → f;g):x ≡                                                  6
     if p:x = T then                                           7
          f:x                                                  8
     elsif p:x = F then                                        9
          g:x                                                  10
     else                                                      11
          ⊥                                                    12
```

The conditional form handles nicely a problem that arises with bottom-preserving functions: One or the other branch of a conditional may be undefined (bottom) while the value of the conditional is itself well defined. If one tries to create a conditional function that takes a triple representing the Boolean value, the true-part value and the false-part value, then if any component is $\perp$, so is the entire triple, forcing $\perp$ as the result. Since conditional is a higher-order function, the evaluator doesn't apply the "**then** function" or "**else** function" until the conditional

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

value has been evaluated and tested against T and F.

One of the advantages of restricting higher-order functions is that they form an algebra, which allows forms to be manipulated in well-defined ways. For example, Figure 4.40 is a theorem:

Figure 4.40 $[f_1 , \ldots , f_n] \circ g \equiv [f_1 \circ g, \ldots , f_n \circ g]$

This theorem states that a composed function may be "distributed into" or "factored from" a list of functions. Such algebraic theorems can be viewed as the basis for automatic restructuring of FP programs, potentially allowing sophisticated optimizations.

5. A set of definitions binding functions to identifiers. These identifiers serve merely as abbreviations and placeholders; there is no concept of redefinition or scoping.

2.2 Reduction Semantics

FP environments have a particularly simple semantics called **reduction semantics**. An FP program is composed of a number of functions applied to objects. The meaning of such a program is defined by repeatedly reducing the program by finding a function application and evaluating it. In some cases, function evaluation may be nonterminating. Such functions diverge and are considered undefined (that is, $\perp$). There are only three kinds of valid functions: primitive functions, defined functions, and higher-order functions. Primitive functions are automatically evaluable. Defined functions are reduced by replacing their identifiers with their definitions. Higher-order functions are reduced by substituting their definitions. If a function does not belong to one of these three categories, it is invalid.

Reduction semantics have only a very weak form of identifier binding (defined names map to functions) and employ no changes to hidden states. There is clearly no way to cause side effects, so an evaluator can reduce a function in any order. In fact, early FP languages were called "Red" (reduction) languages.

3 ◆ PERSISTENCE IN FUNCTIONAL LANGUAGES

A value is **persistent** if it is retained after the program that created it has terminated. A database is a good example of persistent values. The conventional way to make values persistent is to write them out to a file. Chapter 3 discusses the type-safety considerations of such values.

If persistent values are to be incorporated into a programming language, we must be able to name such values and to be assured that once created, they do not change. Functional languages can incorporate persistent values in a natural way that avoids explicit input and output [Morrison 90].

Persistent values can be named by reference to a **persistence root**, which is something like the root of a file-system hierarchy. All such values are automatically saved after execution. If a value is structured, its components are also preserved; in particular, other values pointed to by a persistent value are also persistent. Using ML as an example, we might have the code

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

shown in Figure 4.41.

Figure 4.41

```
let                                                    1
     val persist(MyRoot) a.b.c = 3;                    2
     val persist(MyRoot) d.e;                          3
in                                                     4
     a.b.c + d.e                                       5
end;                                                   6
```

Here, the identifier `a.b.c` is introduced as persistent, under root `MyRoot`, and is given the (permanent) value 3. The identifier `d.e` is not given a value; instead, it gets its value from persistent storage, where it should already be defined.

Since functional languages have no side effects, persistent values are immutable, so there is no need to worry about getting consistent copies if two programs access the values at the same time: such values cannot change. It would be a runtime error to reintroduce the same identifier in a persistence hierarchy. The only modification allowed is inserting an object into (and perhaps removing an object from) the persistent store.

The possibility of lazy evaluation in functional programming languages make them even more attractive for persistent store. An incompletely evaluated value, that is, a suspension, can be saved in persistent store so long as the environment on which it depends is also treated as persistent. If later computation resolves, fully or partially, the suspension, it is safe to replace the stored suspension with the resolved value. Future evaluations, either by the same program or by other programs, will see effectively the same values.

One proposal for integrating persistence into a functional language is to build an imperative command outside the language (at the operating-system level, for example) [McNally 91]. The expression in Figure 4.42

Figure 4.42

```
persist ModuleA requires ModuleB, ModuleC
```

means that all identifiers exported from `ModulaA` are to be placed in persistent store. Both `ModuleB` and `ModuleC` must already be in persistent store; values in `ModuleA` may depend on them. If `ModuleA` is already in persistent store, the new copy replaces it, but any pointers to the identifiers of the previous `ModuleA` are still valid. The old `ModuleA` becomes collectable; that is, garbage collection routines may discard and reclaim the storage of any of its values that are no longer pointed to.

4 ◆ LIMITATIONS OF FUNCTIONAL LANGUAGES

The idea that variables are unnecessary is quite attractive. It is often sufficient either to bind values through parameter binding or as constants for the duration of a block. For example, in ML, the **`let`** construct allows an identifier to be bound to a meaning, but there is no assignment as such. There are situations, however, in which the inability to modify an existing value leads to awkward or inefficient programs [Arvind 89; Yuen 91].

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

The first problem involves initializing complex data structures, particularly two-dimensional arrays. For example, I might want an array A with the properties shown in Figure 4.43.

Figure 4.43

```
A[0,j] = A[i, 0] = 1 ∀ 0 ≤ i < n, 0 ≤ j < n                    1
A[i,j] = A[i, j-1] + A[i-1, j] + A[i-1, j-1]                  2
     ∀ 0 < i < n, 0 < j < n                                   3
```

Short of embedding syntax for this elegant mathematical declaration into a programming language, the most straightforward way to accomplish this initialization is with an imperative program that iterates as in Figure 4.44.

Figure 4.44

```
variable                                                      1
     sum, row, col : integer;                                 2
     A : array [0..n-1, 0..n-1] of integer;                   3
begin                                                         4
     for sum := 0 to 2*n-2 do                                 5
          for row := max(0,sum-n+1) to min(n-1,sum) do        6
               col := sum - row;                              7
               if row = 0 or col = 0 then                     8
                    A[row, col] := 1                          9
               else                                           10
                    A[row,col] := A[row, col-1] +             11
                         A[row-1, col] + A[row-1, col-1]      12
               end;                                           13
          end; -- for row                                     14
     end; -- for sum                                          15
end;                                                          16
```

In a functional language, initialization is usually performed by recursion, which returns the value that is to be associated with the identifier. But there is no obvious recursive method that works here, for several reasons. First, unlike lists, arrays are not generally built up by constructors acting on pieces. The entire array is built at once.[4] Second, the natural initialization order, which is by a diagonal wavefront, does not lend itself either to generating rows or columns independently and then combining them to make the array. Third, special-purpose predeclared array constructor functions can only handle simpler cases in which the value at each cell depends only on the cell's indices. For example, to build an array in which each cell has a value computed as the sum of its row and column, we could employ such a function and invoke it as `MakeArray(1, n, 1, n, (fn row, col => row+col))`. That approach fails here, because the value in a cell depends on other values within the array.

One solution to this problem that largely preserves **referential transparency**, that is, that references to an identifier should always produce the same results, is to separate allocation of data from initialization. After the array is built, a language could permit individual cells to be assigned values,

[4] APL, discussed in Chapter 9, allows arrays to be built piecemeal.

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

but only once each. Accesses to values that have not been initialized would be erroneous. (In concurrent programming, discussed in Chapter 7, such accesses would block the thread that tries to access such a cell until the cell is initialized.) Unfortunately, this solution requires that the language have assignment and that there be runtime checks for violations of the single-assignment rule.

Initialized identifiers are not the only problem with functional languages. A different problem arises if I want to summarize information in counters. For example, I may have a function that returns values from 1 to 10, and I want to invoke the function a million times with different parameters. I want to know how often each of the possible return values appears. In an imperative language, it is quite easy to store such results in an array that is initially 0 everywhere and updated after each function invocation. In a functional language, there seems to be no alternative but to enter a new name scope after each function call, getting a new array that is initialized to the old one except for one position, where it is incremented. The only reasonable way to enter a million name scopes is by recursion, and even that seems problematic. A solution to the problem of summarizing information in a functional language is found in the guardians in Post (discussed in Chapter 6) and in multiparadigm languages like G-2.

Finally, functional languages sometimes lose nuances that are essential to efficiency. For example, the Quicksort algorithm can be expressed elegantly in Miranda (Chapter 3) as in Figure 4.45.

Figure 4.45

```
fun                                                          1
     QuickSort [] = []                                       2
     QuickSort (a :: rest) =                                 3
          QuickSort [ b | b <- rest; b <= a ] @              4
          [a] @                                              5
          QuickSort [ b | b <- rest; b > a];                 6
```

However, this representation misses some important details. First, it is inefficient to make two passes through the array to partition it into small and large elements. Second, stack space can be conserved by recursing on the smaller sublist, not always the first sublist (I assume the compiler is smart enough to replace the tail recursion with iteration). Third, Quicksort should sort elements in place; this implementation builds a new array. The first two details can be programmed in the functional model, although perhaps awkwardly. The other is too intricately associated with concepts of swapping values in memory locations.

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

5 ◆ LAMBDA CALCULUS

The mathematician Alonzo Church designed the lambda calculus in the 1930s as a way to express computation [Church 41]. LISP is a direct descendent of this formalism, and ML owes much of its nature to a restricted version called "typed lambda calculus." In one sense, lambda calculus is a set of rules for manipulating symbols; the symbols represent functions, parameters, and invocations. In another sense, lambda calculus is a programming language; it has given rise more or less directly to both LISP and ML.

The underlying ideas of lambda calculus are straightforward. Lambda calculus has only three kinds of terms: identifiers (such as x), abstractions, and applications. **Abstractions** represent functions of a single parameter. They follow the notation shown in Figure 4.46.

Figure 4.46

```
(λ x . (* x 2)) -- Lambda calculus                    1
(lambda (x) (* x 2)) -- LISP                          2
fn x => x * 2 -- ML                                   3
```

In general, an abstraction has the form (λ x . T), where T is any term. **Applications** represent invoking a function with an actual parameter. A function F is invoked with actual parameter P by the notation (F P); both F and P are any terms. Parentheses may be dropped; the precedence rules stipulate that application and abstraction are grouped left to right and that application has a higher precedence than abstraction. Therefore, the terms in Figure 4.47 are equivalent.

Figure 4.47

```
(λ x . ((λ y . q) x) z) -- fully parenthesized        1
λ x . (λ y . q) x z -- minimally parenthesized        2
```

Another notational convenience is that curried functions may be rewritten without currying, as in Figure 4.48.

Figure 4.48

```
(λ x . (λ y . (λ z . T))) = (λ x y z . T)
```

Lambda calculus has a static scope rule. The abstraction (λ x . T) introduces a new binding for the identifier x; the scope of this binding is the term T. In the language of lambda calculus, x is **bound** in (λ x . T). An unbound identifier in a term is called **free** in that term. It is possible to define the concept of free identifiers in a recursive way: An identifier x is free in term T if (1) the term is just the identifier x; (2) the term is an application (F P), and x is free in F or in P; or (3) the term is an abstraction (λ y . T), and x is free in T, and x is not y. Figure 4.49 presents some examples.

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

Figure 4.49

```
(λ x . y) (λ y . z) -- y and z are free; x is bound          1
(λ x . y) (y z) -- y and z are free; x is bound              2
(λ y . (y z)) -- z is free; y is bound                      3
(λ x . (λ y . z)) -- z is free; x and y are bound           4
(λ x . (λ x . z)) -- z is free; x is bound                  5
```

The example in line 5 introduces two different bindings for x. This situation is analogous to an inner declaration that conflicts with an outer declaration. As you expect, the meaning of x within any term is based on the closest enclosing binding. The rules of lambda calculus, which you will see shortly, ensure that this interpretation is followed.

The heart of lambda calculus is the rule of Figure 4.50 that lets you simplify a term.

Figure 4.50

```
(λ x . T) P =β=> {P / x} T
```

This formula says that applying a function (λ x . T) to an actual parameter P yields the body T of the function, with all occurrences of the formal parameter x replaced by the actual parameter P. This simplification is called β **(beta) reduction**, and I denote it by the symbol =β=> . The notation {P / x} T can be read as "P instead of x in T". It is somewhat awkward to define this substitution operator precisely. First, x may have bound occurrences in T that should not be subject to substitution. These are like nested declarations of x, which hide the outer declaration that we are trying to bind to P. Second, P may have unbound instances of identifiers that are bound in T. These identifiers must remain unbound in the substitution. To achieve this goal, such identifiers need to be renamed in T before substitution takes place. Figure 4.51 shows some examples of substitution.

Figure 4.51

```
{a / b} b = a -- no renaming needed
{a / b} a = a -- no free instances of b
{a / b} (λ c . b) = (λ c . a) -- no renaming needed
{a / b} (λ b . b) = (λ z . z) -- b=>z; no free instances left
{a / b} ((λ b . b)(b c)) = (λ z . z)(a c) -- renamed bound b=>z
{(λ x . y) / x} (x y) = ((λ x . y) y)
```

The concept of renaming bound identifiers can be formalized; it is called α **(alpha) conversion** (see Figure 4.52).

Figure 4.52

```
(λ x . T) =α=> (λ y . {y / x} T)
```

Be aware that α conversion requires that y not be free in T.

Figure 4.53 is a fairly complicated example that uses α-conversions and β-reductions.[5]

[5] Modified from [Sethi 89].

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

Figure 4.53

```
(λ a b c . (a c) (b c)) (λ a . a) (λ a . a) =α=>                1
(λ a b c . (a c) (b c)) (λ z . z) (λ y . y) =β=>                2
(λ b c . ((λ z . z) c) (b c)) (λ y . y) =β=>                    3
(λ b c . c (b c)) (λ y . y) =β=>                                4
(λ c . c ((λ y . y) c)) =β=>                                    5
(λ c . c c)                                                    6
```

Line 2 renames bound identifiers in the second and third terms to remove confusion with identifiers in the first term. Line 3 applies β reduction to the first two terms. Line 4 applies β reduction to the inner application. This choice is analogous to evaluating the parameter to the outer application before invoking the function. In other words, it embodies strict evaluation and value-mode parameter passing; in lambda calculus, it is called **applicative-order** evaluation.

Instead, I could have applied β reduction to the outer application first. This embodies lazy evaluation and name-mode parameter passing; in lambda calculus, it is called **normal-order** evaluation. Under normal-order evaluation, I can reduce the same expression as shown in Figure 4.54.

Figure 4.54

```
(λ a b c . (a c) (b c)) (λ a . a) (λ a . a) =α=>                1
(λ a b c . (a c) (b c)) (λ z . z) (λ y . y) =β=>                2
(λ b c . ((λ z . z) c) (b c)) (λ y . y) =β=>                    3
(λ c . ((λ z . z) c) ((λ y . y) c)) =β=>                        4
(λ c . ((λ z . z) c) c)) =β=>                                   5
(λ c . c c)                                                    6
```

The final result is the same under both evaluation orders. A fundamental theorem of lambda calculus, due to Church and Rosser, is that it doesn't matter in what order reductions are applied. If you start with a particular term T and apply β reductions and α conversions, arriving at terms S and R after two different lists of operations, then there is some ultimate result U such that both S and R derive U. All reduction sequences make progress toward the same ultimate result.

If reduction reaches a stage where no β reduction is possible, the result is in **normal form**. Line 6 in Figure 4.54 is in normal form. Surprisingly, not every term can be reduced to normal form; some reductions continue forever, as in Figure 4.55.

Figure 4.55

```
(λ x . (x x)) (λ x . (x x)) =α=>                               1
(λ x . (x x)) (λ y . (y y)) =β=>                               2
(λ y . (y y)) (λ y . (y y)) =α=>                               3
(λ x . (x x)) (λ x . (x x)) =α=>                               4
```

Line 4 is the same as line 1; the β conversion did not simplify matters at all.

Another example that I will use later is the term Y, defined as shown in Figure 4.56.

Figure 4.56

```
Y = (λ f . (λ x . f (x x)) (λ x . f (x x)))
```

This term has no free identifiers; such terms are called **combinators**. Fig-

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

ure 4.57 shows that Y has an interesting property.

Figure 4.57

```
Y g = (λ f . (λ x . f (x x)) (λ x . f (x x))) g =β=>          1
(λ x . g (x x)) (λ x . g (x x)) =β=>                          2
g ((λ x . g (x x)) (λ x . g (x x))) =                         3
g (Y g)                                                       4
```

Line 4 is surprising; it comes from noticing the similarity between lines 2 and 3. If we continue this "reduction," we move from Y g to g (Y g) to g (g (Y g)) and so forth, expanding the result each time. Combinators like Y with the property that Y g = g (Y g) are called **fixed-point operators**. I will use Y later to define recursive functions.

A last simplification rule, shown in Figure 4.58, is called η **(eta) conversion**.

Figure 4.58

```
(λ x . F x) =η=> F
```

We can only apply η conversion when F has no free occurrences of x. Figure 4.59 shows several η conversions.

Figure 4.59

```
(λ a b  . (+ a b)) =                                          1
(λ a . (λ b . (+ a b))) =                                     2
(λ a . (λ b . (+ a) b))) =η=>                                 3
(λ a . (+ a)) =η=>                                            4
+                                                             5
```

To make a programming language from the lambda calculus requires very little additional machinery. It is necessary to introduce predeclared identifiers, such as `true` and **`if`**, which are called "constants." The set of predeclared constants and their meanings distinguish one lambda calculus from another. The meanings of constants are expressed by reduction rules, such as

```
if false T F => F
```

Here, **`if`** is a curried function of three parameters. This lambda calculus can now be translated directly into ML:

Lambda calculus	**ML**
F P	F P
λ x . T	**fn** x => T
if B T F	**if** B **then** T **else** F
{A / x} T	**let val** x = A **in** T **end**

Recursive function definitions require the combinator Y defined in Figure 4.56 (page 133). Consider the ML declaration in Figure 4.60.

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

Figure 4.60

```
let val rec Parity = fn x =>                    1
     if x = 0 then 0                            2
     else if x = 1 then 1                       3
     else Parity (x - 2)                        4
in                                              5
     Parity 3                                   6
end;                                            7
```

It is not hard to express the body of Parity as a lambda term B, as shown in Figure 4.61.

Figure 4.61

```
B = if (= x 0) 0 (if (= x 1) 1 (r (- x 2))
```

The form of this lambda term bears a strong resemblance to LISP's parenthesized syntax. I have introduced some new constants, such as the nullary operators 0 and 1 (that is, classical constants), and the binary operators = and - .[6] The identifier r is free in this expression; I use it to refer to a recursive call to the function itself. I now define Parity as shown in Figure 4.62, using the fixed-point operator Y.

Figure 4.62

```
Parity = Y (λ r x . B)
```

To show that this definition makes sense, I need to perform some reductions; see Figure 4.63.

Figure 4.63

```
Parity = Y (λ r x . B) =                                    1
(λ r x . B) Y (λ r x . B) =                                 2
(λ r x . B) Parity =β=>                                     3
(λ x . if (= x 0) 0 (if (= x 1) 1 (Parity (- x 2))          4
```

Line 1 represents the definition of Parity. Line 2 comes from the fixed-point nature of Y. Line 3 substitutes the definition of Parity back into the result. Line 4 performs a single β reduction, using the definition of B. Together, these lines show that Parity is in effect defined recursively.

The last step in turning lambda calculus into a programming language is to introduce the concept of types. The constant 0 is meant to be used differently from the constant **if**; the former is nullary, and the latter takes three parameters. In Figure 4.64, following the notation of ML, I can show the types of the constants introduced so far.

[6] The fact that I now have - means that I must have an integer type as well.

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

Figure 4.64

```
0: int
1: int
+: int*int -> int
-: int*int -> int
=: int*int -> bool
if: bool*'a*'a -> 'a
```

Complicated types may be parenthesized, but parentheses may be dropped. The precedence rules stipulate that * is grouped left-to-right and has high precedence, whereas -> is grouped right-to-left and has low precedence. The typed lambda calculus requires that abstractions include type information.

Figure 4.65

```
(λ x : t . T)
```

In Figure 4.65, t is some type indicator. Now it is possible to reject some malformed expressions that were acceptable, but meaningless, before, as shown in Figure 4.66.

Figure 4.66

```
λ x : int . x y                                    1
if 1 (x y) (y x)                                   2
if (= x y) 2 (= x z)                               3
```

Line 1 is unacceptable because x is used in the body of the application as a function, but has type int. Line 2 is invalid because 1 is not of type bool. Line 3 is rejected because both branches of the **if** must have the same type, but one is int and the other is bool. At this point, we have almost built the ML programming language. All that is lacking is some syntactic elegance (such as patterns), data types (lists are very useful for functional programming), and the many parts of ML that I have not discussed at all.

Lambda calculus is valuable for several reasons. First, it gives a purely mathematical, formal basis to the concept of programming, assigning a set of rules that determine the meaning of a program. This ability to mathematically define the semantics of a programming language is investigated in more detail in Chapter 10. Because all chains of reductions give rise to equivalent results, the semantics are not affected by the order of evaluation. Second, it introduces the concept of higher-order functions as a natural building block for programming. Lambda abstraction builds an anonymous function that can be applied or returned as the result of a function. Third, it gives rise to the functional style of programming, because it has no need for variables. The languages that are derived from lambda calculus, particularly LISP and ML, have been quite successful. Insofar as a purely functional subset of these languages is used, they lend themselves to lazy and speculative evaluation. ML takes the concept of typed lambda calculus and infers types in order to enforce strong typing.

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

EXERCISES

Review Exercises

4.1 Why is it natural for a language that has no variables to provide no iterative control constructs?

4.2 If a language treats functions as first-class values, does the language support higher-order functions?

4.3 In Figure 4.3 (page 106), I show that `(cons (cons 'A (cons 'B nil)) (cons nil (cons 11 nil)))` is the same as `(('A 'B) () 11)`. What is the value of the following expression?

```
(cons (cons (cons 'A (cons 'B nil)) nil) (cons 11 nil))
```

4.4 In LISP, is parameter passing by value mode? If not, by what mode?

4.5 In Figure 4.25 (page 113), why introduce the function `Extend`?

4.6 Convert the `Double` function of Figure 4.20 (page 112) into ML.

4.7 Generalize the `Double` function of Figure 4.20 (page 112) so that it doubles recursively within sublists as well as at the top level.

4.8 Generalize the answer to problem 4.7 to make a `Multiple` function that accepts two parameters: a list `L` and a multiplier `M`, so that if `M` is 2, the effect is like `Double`, but higher and lower integer multipliers also work.

4.9 Under what circumstances does it make a difference in what order the parameters to a function are evaluated?

4.10 Reduce the following lambda expression to normal form.

```
(λ y . (λ z . x z (y z))) (λ a . (a b))
```

4.11 Reduce the lambda expressions given in Figure 4.67.

Figure 4.67

```
{a / b}(λ a . b)
{a / b}(λ b . a)
{a / b}(λ c . b)
{a / b}(λ b . c)
{a / b}(λ a . a)
{a / b}(λ b . b)
{a / b}(λ c . c)
```

Challenge Exercises

4.12 What does it mean for something to be a first-class value in a purely functional language?

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.

4.13 As suggested in the text (page 111), show how to use an ML datatype to implement heterogeneous lists. You may assume that atoms are always integers or `nil`. Implement both `Reverse` and `ReverseAll`. If you use `::` as a constructor for your list datatype, ML automatically overloads the `[]` syntax for you.

4.14 What sort of runtime storage organization is appropriate for ML? Restrict your answer to the purely functional part of ML.

4.15 On page 114, I claim that building a closure is easy in statically scoped languages in which procedures are not first-class values. Is it harder if procedures are first-class values?

4.16 Does LISP really need garbage collection? Wouldn't reference counts suffice?

4.17 On page 116, I suggest that the implementation copy only the bindings of selected nonlocal identifiers whose bindings it needs to preserve. How does it know which ones?

4.18 On page 117, I say that the updated environment is discarded after a **`lambda`** body is evaluated. But the example shows no explicit discard. Explain.

4.19 How must the LISP interpreter be enhanced to deal with deep binding?

4.20 The trace of lazy evaluation in Figure 4.38 (starting on page 122) happened not to need to return to a partially evaluated result. Trace the more interesting example

```
(SameFrontier '(A B C) '(A C D)),
```

which will need to do so.

4.21 In FP, the sequence constructor is bottom-preserving. Show how this requirement precludes lazy evaluation.

4.22 To introduce persistence into a functional language, I have used an imperative command. Is an imperative style necessary?

4.23 Use the combinator `Y` to build a lambda-calculus definition of integer multiplication that translates the ML program of Figure 4.68.

Figure 4.68

```
val rec Multiply = fn (x,y) =>                          1
      if x = 0 then 0 else Multiply(x-1,y) + y;         2
```

4.24 What is the type of `Y` in Figure 4.56 (page 133)?

4.25 Write a lambda term that grows longer after each β reduction.

Copyright © Addison-Wesley. Reproduction fee $.02 per page, per copy.