11 IMPLEMENTATION OF RECURSIVE LIST-PROCESSORS: LISP

11.1 RECURSIVE INTERPRETERS

A LISP Interpreter Can Be Written in LISP

In Chapter 9, Section 9.1, we said that the first LISP interpreter resulted from writing a *universal function* for LISP. A universal function is a function that can interpret any other function. In Section 11.1 we develop a universal function for LISP because it is an example of a *recursive interpreter*, one of two major classes of interpreters. In Chapter 1 we saw an example of the other major class, *iterative interpreters.*

The recursive interpreter is written in LISP, although it could be written in any language with recursive procedures and the ability to implement linked lists. Since it is written in LISP, it makes use of the facilities of LISP, such as the list-processing operations `car`, `cdr`, and `cons`. In particular, these operations are used to interpret `car`, `cdr`, and `cons` operations in the program that we are interpreting. This might seem circular and pointless. In fact it is exactly analogous to the way floating-point operations were implemented in the pseudo-code interpreter in Chapter 1 (Section 1.3). There, floating-point operations in the implementation language (say, Pascal) were used to implement floating-point operations in the pseudo-code. Of course, if our implementation language had not had a floating-point capability, then these operations would have had to be implemented in terms of more basic operations. Similarly, if the implementation language for a LISP interpreter does not have list manipulation operations, it is necessary to implement these in terms of more basic operations. However, since we are using LISP as the implementation language, we can use list manipulation operations directly.

The LISP universal function is conventionally called `eval` since it evaluates a LISP expression. In addition to the expression to be evaluated, `eval` must have a second parameter, which is a data structure (a list of some sort) representing the context in which the evaluation is to be done. Recall that it is incomplete just to ask for the value of an expression; *it is also necessary to specify the context of the evaluation* (see Section 2.5, p. 78, and Section 6.1, p. 212). Hence, if E is any LISP expression (written in the S-expression notation) and A is a list representing a context, then

(eval '$E\,A$) = V

where *V* is the value of *E* in that context. In other words, the result of evaluating `(eval '`*E* *A*`)` is the same as the result of evaluating *E* in the context represented by *A*. Consider the following application (where we have assumed `nil` represents the empty context):

```
(eval '(cons (quote A) (quote (B C D)) ) nil)
   (A B C D)
```

The result agrees with the result of evaluating `(cons 'A '(B C D))` in the empty (or any other) context, which is `(A B C D)`.

The Interpreter Is Arranged by Cases

If we were to evaluate a LISP expression by hand, our first step would be to classify it, that is, to decide the sort of LISP expression with which we are dealing. This is exactly what we will do in `eval`, and the first step is to classify LISP expressions.

First, we have atoms, such as `2` and `val`. Numeric atoms, such as `2`, represent themselves; nonnumeric atoms, such as `val`, represent the value to which they are bound.

All other LISP expressions are represented by nonatoms, that is, lists. These are all some form of *application.* We have *primitive* applications such as `(car x)` and `(cons x y)`. We also have applications of *user-defined* functions such as `(make-table text nil)`. Finally, we have *special* applications, such as `(quote x)` and `(if p e f)`.[1] The kinds of LISP expressions are summarized in Table 11.1.

The `eval` function will break its input down into cases as shown in Table 11.1. The LISP mechanism for handling cases is `if`, so `eval` will take the form of a large `if`:

```
(defun eval (e a)
   (if
      (atom e)
      Handle atoms
      Handle lists))
```

TABLE 11.1 Types of LISP Expressions

Type	Example
Numeric atom	`2`
Nonnumeric atom	`val`
Quotation	`(quote (B C D))`
Conditional	`(if (null x) nil 1)`
Primitive	`(cons x y)`
User defined	`(make-table text nil)`

[1] Recall that `'`*X* is just an abbreviation for `(quote `*X*`)`.

There are two kinds of atoms—numeric and nonnumeric—so the "Handle atoms" procedure is

```
(if (numberp e)
      Handle numeric atoms
      Handle nonnumeric atoms)
```

(Numberp is a LISP predicate that returns `t` *if its atomic argument is a number.)*

Applications fall into two broad categories: (1) the special functions (`quote` and `if`) that do not evaluate their arguments and (2) the normal applications (primitives and user defined) that do. We can distinguish these two cases by looking at the name of the function [given by `(car e)`] to see if it is special:

```
(cond
   ((eq (car e) 'quote) Handle quotations)
   ((eq (car e) 'if      Handle conditionals)
   (t                    Handle normal applications))
```

By combining all of the above cases, we get the following structure for the `eval` function:

```
(defun eval (e a)
   (if
      (atom e)
      (if (numberp e)
            Handle numeric atoms
            Handle nonnumeric atoms)
      (cond ((eq (car e) 'quote) Handle quotations)
             ((eq (car e) 'if      Handle conditionals)
             (t                    Handle normal applications))
   ))
```

With only five distinct cases, it is really unnecessary to have three nested conditionals; the cases are more obvious if we "flatten" the structure:

```
(defun eval (e a)
   (cond
     ((and (atom e) (numberp e)) Handle numeric atoms)
     ((atom e)                   Handle nonnumeric atoms)
     ((eq (car e) 'quote)        Handle quotations)
     ((eq (car e) 'if)           Handle conditionals)
     (t                          Handle normal applications))  )
```

We will address each of these cases in the following sections.

The Value of a Numeric Atom Is That Atom

First we will consider the evaluation of numeric atoms. The value of 2 is the atom 2; therefore,

```
(eval 2 a) = 2
```

and the case for handling numeric atoms is

```
((and (atom e) (numberp e)) e)
```

Nonnumeric Atoms Must Be Looked Up in the Environment

Next we will consider nonnumeric atoms, such as `val` or `text`. These are assumed to be bound to values, and the result of evaluating them is the value to which they are bound. The value to which an atom is bound is determined by the *environment* in which the atom is evaluated. This is the purpose of the second parameter to `eval`; `(eval` *E A*`)` calls for the evaluation of expression *E* in environment represented by the list *A*. For example, if the environment `A` binds `val` to 2 and `text` to `(to be or not to be)`, then

```
(eval 'val A)
   2
(eval 'text A)
   (to be or not to be)
```

There are many ways in which environments can be represented in LISP; one of the simplest is an *association list* (Chapters 9, Section 9.3). An environment that binds `val` and `text` as specified above could be represented by this association list:

```
( (val 2) (text (to be or not to be)) )
```

The evaluation of `val` in this environment is

```
(eval 'val '((val 2) (text (to be or not to be)) ))
   2
```

We can see now how a nonnumeric atom must be evaluated: We have to look up its value in the association list representing the current environment. Looking something up in an association list is accomplished with the `assoc` function that we described earlier (Chapter 9, Section 9.3). Thus, if *e* is a nonnumeric atom

```
(eval e a) = (assoc e a)
```

The first two cases of the `eval` function are now defined as follows:

```
(defun eval (e a)
   (cond
     ((and (atom e) (numberp e)) e)
     ((atom e) (assoc e a))
     ... ))
```

A common programming error is to attempt to use an undefined variable. The code above does not check for this error condition; it assumes that the name *e* is bound in the environment *a*. Of course, a good interpreter would check for this error condition; we will omit this and most other checking to keep the presentation simple. Try to identify all the places in the interpreter where error checking should be done.

▩ ***Exercise 11-1:*** Revise the handling of nonnumeric atoms to issue a diagnostic on an attempt to evaluate an unbound name. Assume a function `(error m)` is available that prints the value of *m* and terminates evaluation.

Quote Suppresses Evaluation

Next we will consider the evaluation of the various applications, beginning with the special applications `quote` and `if`. The whole purpose of `quote` is to *suppress* or *prevent* the evaluation of its argument. In other words,

```
(eval '(quote X) A) = X
```

no matter what *X* might be. Therefore, the case for handling `quote` is trivial:

```
((eq (car e) 'quote) (cadr e))
```

Note that `(cadr e)` is the first (in this case, only) argument of the application `e`.

The Conditional Delays Evaluation of Its Arguments

The conditional expression is different from other built-in functions in that it delays evaluation of its arguments. That is, an argument (consequent or alternate) is not evaluated until its value is needed—in effect it is passed by name. This *lenient* evaluation strategy is necessary, since the conditional may return a value even though some of its arguments are undefined (recall our discussion in Section 10.1 of the conditional interpretation of the logical connectives). Now consider a conditional:

```
(if P T F)
```

Its evaluation proceeds as follows: First evaluate *P*. If it is `t`, then evaluate *T*; if it is `nil`, then evaluate *F*. How can this process be programmed in LISP?

If `e` is a conditional `(if P T F)`, then `(cadr e)` is *P*, the condition of the conditional, `(caddr e)` is *T*, the *consequent* (then-part), and `(cadddr e)` is *F*, the *alternate* (else-part). Thus, to check the condition we must evaluate `(cadr e)` in the context `a` of the `if`; if the result is `t`, we evaluate `(caddr e)` in this same context, otherwise we evaluate `(cadddr e)` in it. These evaluations are accomplished by a recursive call of `eval`, which ensures that any legal LISP expression can be used as the condition, consequent, or alternate of an `if`. This is one of the sources of LISP's generality.

It is now easy to write the checking process in LISP:

```
(if (eval (cadr e) a)
    (eval (caddr e) a)
    (eval (cadddr e) a))
```

Note that, as promised at the beginning of Section 11.1, we are using `if` to interpret `if`.

■ ***Exercise 11-2:*** Use pass by name parameters in Algol-60 to program a function `if` defined so the `if(`P, T, F`)` returns the value of T if P is **true** and the value of F if it is **false**. The `if` function should evaluate a parameter only if it is necessary to do so. Thus, the following invocation should execute correctly:

```
x := if (y = 0, 1, x/y)
```

■ ***Exercise 11-3:*** Extend `eval` to implement `cond`. (*Hint:* You will want to implement an auxiliary function to handle the list of condition-consequent pairs.)

Arguments Are Recursively Evaluated

The only cases remaining are the applications of primitive and user-defined functions. In both of these cases, the arguments of the application must be evaluated. This will be accomplished by a function called `evargs`, which we must define. Notice that if `e` is the application

$(f\ x_1\ x_2\ \ldots\ x_n)$

then `(car e)` is f, the function to be applied, and `(cdr e)` is the list of arguments. Therefore,

```
(evargs (cdr e) a)
```

will be the list of argument values.

How is `evargs` to be defined? We need to construct a list, the ith element of which is the result of evaluating `(eval` x_i `a)`. This is clearly an application for `mapcar` since we want to perform an operation—evaluation in the environment `a`—on each element of the list. That is, we want to apply to each element of the list the function

```
(bu (rev 'eval) a)
```

since

```
((bu (rev 'eval) a) xᵢ)
  = ((rev 'eval) a xᵢ)
  = (eval xᵢ a)
```

The definition of `evargs` follows immediately:

```
(defun evargs (x a) (mapcar (bu (rev 'eval) a) x))
```

The function to be applied and the evaluated list of arguments can then be passed to a function `apply` that performs the application. The `apply` function has three arguments:

`(apply` $f\ x\ a$`)`

The first, f, is the function to be applied, which is computed by `(car e)` in this case; the second, x, is the list of evaluated actual parameters, which is computed by `(evargs (cdr`

e) a) in this case; the third, a, is the environment of the caller, which is a in this case. Therefore, the case in eval to handle all normal applications is

```
(t (apply (car e) (evargs (cdr e) a) a) )
```

■ ***Exercise 11-4:*** Program evargs recursively, that is, without using mapcar, bu, or rev.

Primitive Operations Are Performed Directly

There are two types of normal applications: primitive functions and user-defined functions. We will now consider the primitive functions.

Since there are a number of primitive operations, the natural structure for the apply function is a cond that handles the different primitive operations. That is,

```
(defun apply (f x a)
  (cond
   ((eq f 'plus)    Handle a plus)
   ((eq f 'car)     Handle a car)
   ((eq f 'cdr)     Handle a cdr)
   ((eq f 'cons)    Handle a cons)
   and so forth ))
```

We will consider a typical function, plus. Suppose we are evaluating the application

```
(plus 1 (times 2 3))
```

the arguments passed to apply will be

```
f = plus
x = (1 6)
a = the environment of the caller
```

Notice that x is a *list* containing the *values* of the actual parameters. What should the result of apply be? In this case, we must add the first argument to the second argument. The first argument is (car x) and the second argument is (cadr x), so, if f is plus, we should return

```
(plus (car x) (cadr x))
```

in summary, the case for handling plus is

```
((eq f 'plus) (plus (car x) (cadr x)) )
```

Notice that, as promised at the beginning of this section (p. 375), we are using plus to implement plus. This is correct since we are using LISP as both the implementation language and the interpreted language. Similarly, the list-processing operations (car, cdr, etc.) will be implemented in terms of themselves. If we were using a language such as Pascal as an implementation language, it would be necessary to define these operations in terms of more basic pointer manipulation operations, such as those described in Chapter 9, Section 9.3.

The other primitive functions are similar; we show here an `apply` function that handles the primitive list manipulation functions. Other primitive operations (arithmetic, predicates, etc.) are implemented in the same way.

```
(defun apply (f x a)
   (cond
    ((eq f 'plus)   (plus (car x) (cadr x)) )
    ((eq f 'car)    (car (car x)) )
    ((eq f 'cdr)    (cdr (car x)) )
    ((eq f 'atom)   (atom (car x)) )
    ((eq f 'null)   (null (car x)) )
    ((eq f 'cons)   (cons (car x) (cadr x)) )
    ((eq f 'eq)     (eq (car x) (cadr x)) )
                    .
                    .
                    .
               other primitives
                    .
                    .
                    .
    (t    Handle user-defined functions )) )
```

▩ ***Exercise 11-5:*** Add the other arithmetic functions (`difference`, `times`, `quotient`) and the predicates (`minusp`, `lessp`) to the definition of `apply`.

▩ ***Exercise 11-6*:*** Add error checking to `apply`. For example, the arithmetic operations should be applicable only to numbers and the list operators only to lists of the appropriate type (e.g., `car` only applies to nonnull lists).

User-Defined Function Applications Require Four Steps

All that is required to complete our LISP interpreter is to handle the application of user-defined functions. The steps required to do this are very similar to the steps required to invoke a procedure in other languages.

It is easy to see the necessity for these steps. The goal of a function application is the evaluation of the body of the function. However, as we have seen before, an expression has meaning only in some *context* or *environment*. This environment must bind all of the names used in the expression.

Where does this environment come from? It is composed of *locals* and *non-locals*, the locals being the formal parameters of the function. Since LISP uses dynamic scoping, the nonlocals come from the environment of the caller. Therefore, to construct the environment of evaluation for the function's body, it is necessary to add the formal-actual bindings to the environment of the caller.

We get the formal-actual bindings by binding each formal to the *value* of the corresponding actual. In summary, the following four steps are required to apply a user-defined function:

1. Evaluate the actual parameters.
2. Bind the formal parameters to the actual parameters.
3. Add these new bindings to the environment of evaluation.
4. Evaluate the body of the function in this environment.

We have already accomplished the first step; `evargs` is used to evaluate the actuals before they are passed to `apply`. Therefore, we will address the remaining three steps.

Constructing the Environment of Evaluation

Suppose that we are evaluating the application

```
(consval text)
```

in an environment that binds `text` to `(to be or not to be)`, binds `val` to `whether`, and binds the function name `consval` to

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

Notice that `consval` is bound to a lambda expression since this is how a function value is represented.

Since the actual parameters have been evaluated, the arguments to `apply` are

```
f = consval
x = ( (to be or not to be) )
a = ( ... (val whether) ... (consval (lambda (x) ...)) ... )
```

Notice that the actual parameter list `x` is a one-element list containing `(to be or not to be)`, the value of the only actual parameter. Before we can do any of the remaining three steps, it is necessary to determine to which function `consval` is bound. In other words, it is necessary to interpret `consval` *in context.* This is accomplished by evaluating the function's name since this will cause it to be looked up in the current environment. In this case, `(eval f a)` returns

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

which is the value of `consval`. Let's call this *L*, which stands for "lambda expression."

We can now proceed with constructing the environment of evaluation. The first step is to bind the formals to the actuals. In general, *L* will be a lambda expression:

L = `(lambda` $(v_1\ v_2 \ldots v_n)$ B`)`

where the v_i are the formal parameters and B is the body. The actual parameters, x_i, are a corresponding list:

$$x = (x_1\ x_2 \ldots x_n)$$

Since environments are represented by association lists, binding the formals to the actuals is accomplished by pairing up the formals and actuals in an association list:

$$LE = ((v_1\ x_1)\ (v_2\ x_2)\ ...\ (v_n\ x_n))$$

(*LE* stands for "local environment.") It should now be clear that *LE* can be constructed by mapping the `list` function across the list of formals and *x*. Since the list of formals is just `(cadr` *L*`)`, we have

```
LE = (mapcar 'list (cadr L) x)
```

The next step is to add these new bindings to the environment of the caller, thereby forming the environment of evaluation. Since the `assoc` function, which is used for variable accessing, searches from the beginning of the association list, if the local environment is appended to the beginning of the nonlocal, the new bindings will supersede any previous bindings of the same names. The environment of evaluation, *EE*, is constructed by

```
EE = (append LE a)
```

since `a` is the current environment, that is, the environment of the caller.

The last step is to evaluate the body of the lambda expression in *EE*. Since `(caddr` *L*`)` is the body of the lambda expression, this is accomplished by

```
(eval (caddr L) EE)
```

These pieces can all be assembled into the following LISP code for applying a user-defined function:

```
(let ((L (eval f a) ))
   (let ((LE (mapcar 'list (cadr L) x) ))
        (eval (caddr L) (append LE a)) ))
```

The complete LISP universal function, or interpreter, is shown in Figure 11.1; it is about 20 lines long.[2] The fact that a LISP interpreter can be written in LISP in so few lines is a testament to LISP's simplicity and power.

■ ***Exercise 11-7:*** The LISP universal function is inefficient in one place: The `append` function copies the entire list *LE*, even though it has just been created by `(mapcar 'list ...)`. Show that this can be avoided by writing a new function `pairlis` that performs the pairing and appending operations at the same time. That is,

```
EE = (pairlis (cadr L) x a)
```

Modify `eval` to use this new function.

■ ***Exercise 11-8**:*** The LISP universal function we have described is not a complete LISP interpreter. In particular, it does not look at the property lists of atoms when evaluating them (i.e., it assumes atoms are bound to values only via association lists). As a

[2] We have omitted most of the primitive operations, since they are a routine addition to `apply`.

```
(defun eval (e a)
  (cond
   ((and (atom e) (numberp e)) e)
   ((atom e) (assoc e a))
   ((eq (car e) 'quote) (cadr e))
   ((eq (car e) 'if (if (eval (cadr e) a)
                        (eval (caddr e) a)
                        (eval (cadddr e) a)) )
   (t  (apply (car e) (evargs (cdr e) a) a) )) )

(defun evargs (x a) (mapcar (bu (rev 'eval) a) x))

(defun apply (f x a)
  (cond
   ((eq f 'car)    (car (car x)) )
   ((eq f 'cdr)    (cdr (car x)) )
   ((eq f 'atom)   (atom (car x)) )
   ((eq f 'null)   (null (car x)) )
   ((eq f 'cons)   (cons (car x) (cadr x)) )
   ((eq f 'eq)     (eq (car x) (cadr x)) )
   (t (let  (( L   (eval f a) ))
              (let (( LE (mapcar 'list (cadr L) x) ))
                   (eval (caddr L) (append LE a)) )) )) )
```

Figure 11.1 LISP Universal Function in LISP

consequence, it does not implement property-modifying pseudo-functions such as `defun` and `set`. Add these facilities to `eval`.

Closures Implement Static Scoping

In Chapter 10, Section 10.2, we discussed the `function` construct, which is used to bind a lambda expression to its environment of definition. That is, when we write

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

we mean that the lambda expression should always be called in the environment in which `function` is applied. This preserves the context of the lambda expression's body by simulating static scoping.

How is the `function` construct implemented? Its purpose is to inform the interpreter that the environment of definition of the lambda expression must be remembered so that it can be used when the lambda expression is applied. This is exactly the purpose of a *closure*, which we have discussed many times in the past. A closure has two parts:

1. An *ip*, or *instruction part*, that points to a piece of program
2. An *ep*, or *environment part*, that points to the environment in which that piece of program must be evaluated

In this case, the instruction part is the lambda expression and the environment part is the environment of definition.

Thus, the result of `function` is a closure containing the lambda expression and its environment. How should a closure be represented? The simplest approach is to use a list, the first element of which is the atom `closure`,[3] to identify the list as a closure. For example,

(closure *ip ep*)

This is the data structure that must be constructed by `function`.

The actual implementation of `function` is quite simple. Since it is a special function (because it does not evaluate its argument), it is handled in the main body of `eval` in the same way as `quote` and `cond`. The modified `eval` is

```
(defun eval (e a)
  (cond
   ((and (atom e) (numberp e)) e)
   ((atom e) (assoc e a))
   ((eq (car e) 'quote) (cadr e))
   ((eq (car e) 'if)
    (if (eval (cadr e) a)
        (eval (caddr e) a)
        (eval (cadddr e) a)) )
   ((eq (car e) 'function) (list 'closure (cadr e) a))
   (t  (apply (car e) (evargs (cdr e) a) a) )) )
```

since `(cadr e)` is the argument of `e` (i.e., the lambda expression).

The above modification handles just the *construction* of a closure; it does not *use* the closure. The other place where we must deal with closures is `apply` since it handles function calls and, hence, must know whether to call in the environment of definition or in the environment of the caller. The result of evaluating the function `(eval f a)` may be either a lambda expression or a closure. The case of a lambda expression is the same as in Figure 11.2. Let's consider what must be done if it is a closure:

L = (closure (lambda ($x_1 \dots x_n$) *B*) *ED*)

The local environment is constructed just as before, except now it must be appended to the environment of definition, *ED*, rather than the caller's environment, `a`. Hence,

LE = (mapcar 'list (cadadr *L*) x)
EE = (append *LE* (caddr *L*))

since (caddr *L*) is *ED*—the environment of definition—and (cadadr *L*) is the list of formal parameters.

[3] In most LISP systems the atom `funarg`, meaning "functional argument," is used instead of `closure`.

```
(defun eval (e a)
  (cond
   ((and (atom e) (numberp e)) e)
   ((atom e) (assoc e a))
   ((eq (car e) 'quote) (cadr e))
   ((eq (car e) 'if) (if (eval (cadr e) a)
                          (eval (caddr e) a)
                          (eval (cadddr e a)) )
   ((eq (car e) 'lambda) (list 'closure e a))
   (t (apply (car e) (evargs (cdr e) a) a) )) )

(defun evargs (x a) (mapcar (bu (rev 'eval) a) x))

(defun apply (f x a)
  (cond
   ((eq f 'car)    (car (car x)) )
   ((eq f 'cdr)    (cdr (car x)) )
   ((eq f 'atom)   (atom (car x)) )
   ((eq f 'null)   (null (car x)) )
   ((eq f 'cons)   (cons (car x) (cadr x)) )
   ((eq f 'eq)     (eq (car x) (cadr x)) )
   (t (let (( L (eval f a) ))
          (let (( LE (mapcar 'list (cadadr L) x) ))
               (eval (caddadr L) (append LE (caddr L)) )) )) ))
```

Figure 11.2 Statically Scoped LISP Interpreter in LISP

Since both dynamically scoped lambda expressions and statically scoped closures must be accommodated, the handling of user-defined functions is a little more complicated.

```
(defun apply (f x a)
  (cond
   ((eq f 'car) (car (car x)) )
   ... etc. ...
   (t (let ((L (eval f a) ))
          (if
              (eq (car f) 'closure)
              (let (( LE (mapcar 'list (cadadr L) x) ))
                   (eval (caddadr L) (append LE (caddr L)) ))

              (let (( LE (mapcar 'list (cadr L) x) ))
                   (eval (caddr L) (append LE a)) )) )) ))
```

[Note that `(caddadr L) = (caddr (cadr L))` = the body of the lambda expression, which is the *ip* of the closure *L*.]

■ ***Exercise 11-19:*** Show that `caddr` accesses the environment of definition from a closure.

■ ***Exercise 11-10:*** Show that `cadadr` accesses the formal parameter list from a closure.

■ ***Exercise 11-11:*** Show that `caddadr` accesses the function body from a closure.

Lexical Scoping Uses Closures Uniformly

We are now trying to make our interpreter accommodate two incompatible scope rules. As we discussed in Chapter 10, Section 10.2, recent LISP dialects (such as Scheme) have adopted a uniform static scope rule. This simplifies the interpreter. The most obvious simplification is that the `apply` function has to handle only one kind of function, that represented by closures. Furthermore, since *all* lambda expressions are bound to their environments of definition, there is no longer any need for `function`. That is,

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

is considered equivalent to

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

A closure is constructed for all lambda expressions. The interpreter that results from these alterations is shown in Figure 11.2; it is as simple as the dynamic scoping interpreter.

Common LISP, in an attempt to gain the advantages of static scoping while remaining reasonably compatible with older LISP dialects, has adopted a combination of static and dynamic scoping. For example, formal parameters and variables (established by `sets`, `lets`, and the like) are normally statically scoped. On the other hand, variables can be given dynamic scope by declaring them "special."[4] Function definitions established by `defun` are global; therefore, static and dynamic scoping amount to the same thing for them.

■ ***Exercise 11-12*:*** Extend the LISP universal function that we have developed to implement `let` (Section 10.2).

11.2 STORAGE RECLAMATION

Explicit Erasure Complicates Programming

In our discussion of the representation of lists (Chapter 9, Section 9.3), we saw that the `cons` operation calls for the allocation of a new cons-cell. These cons-cells are allocated from a free storage area similar to that used by the `new` operation in Pascal. Since conses are so

[4] Certain other constructs, such as "catchers" (analogous to Ada exception handlers) also have dynamic scope. The justification for dynamic scoping of exception handlers was discussed in Section 8.1.

frequent in LISP, it is clear that this free storage area will be exhausted by most programs. Unless there is some way of returning cells to the free area, the program will have to be aborted.

One solution to this problem—the one adopted by Pascal and many other languages—is *explicit erasure*. This means that when the program no longer needs a certain cell, the programmer must explicitly return it to the free storage area. In Pascal this is accomplished by the `dispose` procedure. The invocation `dispose(`*p*`)` returns to free storage the cell pointed to by *p*. This is normally accomplished by linking the cell onto a *free-list*, where the storage allocator can find it to satisfy a later storage allocation (`new`) request.

Explicit erasure has several problems. First, it requires programmers to work harder; they must keep careful track of each cell and of all of the lists in which it participates (remember, sublists can be shared) and determine when the cell can be released. Some people have claimed that "good" programmers will understand their programs well enough to be able to do this. You may recognize this as the same argument that was used against early assemblers and compilers (such as FORTRAN): "Good" programmers do not need symbolic names since they know the absolute addresses of all the variables. It is the more dystopian view of programming languages, since it focuses on the negative aspects of greater distance from the machine (Section 1.4). Now we know that it is much better to have a computer take care of bookkeeping details such as these since it allows programmers to concentrate on more important issues of program structure and organization. Automatic erasure is advantageous if it can be accomplished transparently.

Another problem with explicit erasure is that it violates the *security* of the programming system. Suppose that a cell is returned to free storage but is still referenced by several other lists. These other lists will now have *dangling pointers*, that is, pointers that do not point to an allocated cell. This situation is illustrated in Figure 11.3. When the storage allocator reuses this cell, these dangling references will likely cause mysterious errors. In fact, since unallocated cells may still be accessible from the program (via dangling references), it is even possible to corrupt the storage allocator's data structures (e.g., the free-list).

Garbage collection represents an application of the Responsible Design Principle, since it provides a better solution to the problem (dynamic storage allocation and deallocation) than is often requested by programmers (operations for explicit allocation and deallocation). Although some languages, such as C and its offspring, have gone back to explicit operations,

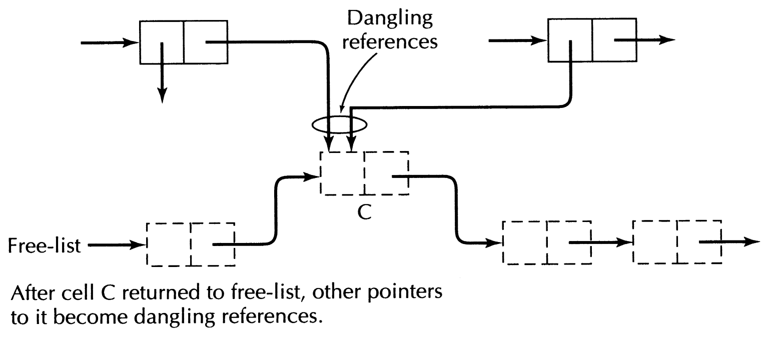
Dangling
references
C
Free-list
After cell C returned to free-list, other pointers
to it become dangling references.

Figure 11.3 Example of Dangling References

there has been a rediscovery of garbage collection in languages concerned with programming security (e.g., Java).

References Counts Identify Inaccessible Lists

We have seen that explicit erasure is low level and error-prone. What is the alternative? The goal is to return a cell to free storage when it is no longer needed, that is, when it is no longer accessible from the program. Therefore, an automatic erasure system must somehow keep track of the accessibility of each cell. In LISP a cell is accessible only if it is *referenced* (i.e., pointed at) by some other accessible cell. The only cells that are *directly accessible* are those used by the interpreter, such as those containing the program being interpreted and the association lists representing environments still in use.

Note that when we are considering whether a cell can be reclaimed, it is not important *which* cells point to this particular cell; all that is relevant is the *number of accessible cells* referencing the cell of concern. This observation leads to a technique of storage reclamation called *reference counts* since it keeps track of the number of accessible references to each cell. One of many[5] ways to do this is to include a *reference count field* in each cell (see Figure 11.4).

Reference counts must be maintained correctly. Whenever an additional reference to a cell is made, that cell's reference count must be incremented. Similarly, whenever a reference to a cell is destroyed, that cell's reference count must be decremented. There are two ways a reference can be destroyed. A pointer can be overwritten (such as happens in a `rplaca`, `rplacd`, Section 9.3, or assignment operation) or the cell containing the pointer can itself become inaccessible.

When a cell's reference count becomes zero, it means that the cell is inaccessible and can be returned to the free-list. Notice, however, that each cell contains pointers to two other cells, that is, the pointers in its `left` and `right` fields. Therefore, whenever a cell is released, the reference counts of the two cells to which it points must also be decremented

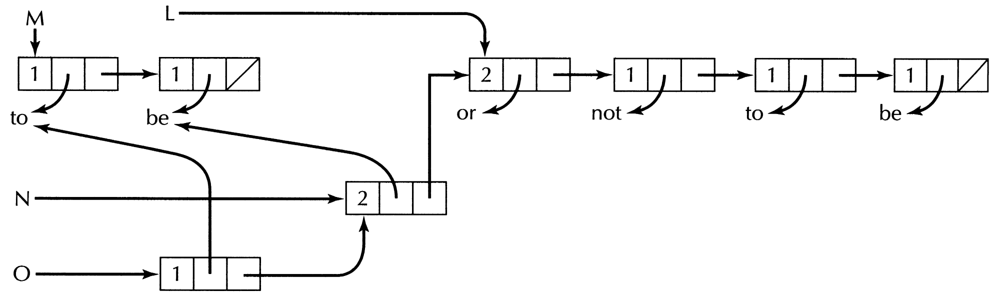

Figure 11.4 Example of Reference Counts

[5] There are a great many techniques for automatic storage reclamation; in this section we can touch on only the simplest. The interested reader should consult the data structures literature for more information. See especially Cohen (1981).

since there is now one less *accessible* reference to each. This decrementing may cause the reference counts of either of these cells to go to zero, which means that they also must be freed. Hence, the process of decrementing a reference count may be recursive. Here is the procedure for decrementing the reference count of cell C:

```
decrement (C):
          reference count (C) := reference count (C) - 1;
          if reference count (C) = 0 then
             decrement (C↑.left);
             decrement (C↑.right);
             return C to free-list;
          end if.
```

Notice that this approach reclaims a cell *as soon as* it becomes available. Therefore, if the free-list is ever exhausted, it means that there is *no more* storage available, and the program must be aborted. Further, studies have shown that almost all (95%) reference counts are 1, so at least one cell is usually freed whenever a pointer is destroyed.

Cyclic Structures Are Not Reclaimed

Consider the list structure shown in Figure 11.5. It is a *cyclic structure* that could have been constructed by using `rplacd` to make the right field of C point to A. Suppose that the only references to these fields are those shown in the figure; then the reference counts will be as shown. Notice that this structure is accessible from only one place—the pointer entering from the left. Now suppose that this pointer is destroyed, making the structure inaccessible; this will cause A's reference count to be decremented to 1. Since neither A, B, nor C has a zero reference count, no cells will be returned to free storage *even though none of these cells is accessible!* This situation will occur wherever there is a *cycle* in the list structure, that is, wherever there is a path from a cell back to itself. In a reference-counted system, these cells will be lost forever.

Several solutions to this problem have been proposed. One is to disallow cyclic structures altogether. Since they are produced only by using the `rplaca` and `rplacd` pseudo-functions, eliminating these will eliminate cycles.[6] Some computer scientists have argued for this solution on the basis that these pseudo-functions have side effects and do

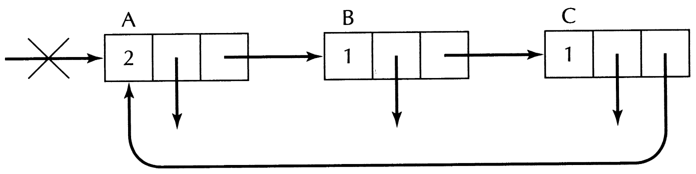

Figure 11.5 Cyclic List Structure

[6] Under some circumstances `putprop`, `set`, `defun`, and similar pseudo-functions can produce cycles through property lists.

not belong in an applicative programming language. They claim that cycles are error-prone and difficult to understand. Other computer scientists have argued that cyclic structures are necessary for some problems. Whether cyclic structures should or should not be a part of LISP or whether they could be included in a tamed form is an open research problem.

▩ ***Exercise 11-13*:*** Discuss the pros and cons of cyclic list structures. Give at least one example of the usefulness of cyclic structures. Explain how you could do without them in that situation.

▩ ***Exercise 11-14**:*** Develop a reference counting scheme that is not fooled by cycles.

Garbage Collection Reclaims Cyclic Structures

There is an alternative approach to automatic storage reclamation that can handle cyclic structures; it is called *garbage collection*. With reference counts, cells are released as soon as they become inaccessible. Thus, cells that are inaccessible but not available for reuse ("garbage") never accumulate. Garbage collection adopts a different philosophy. Inaccessible cells are simply abandoned; no attempt is made to return them to free storage. When the supply of free cells is exhausted, the system enters a *garbage collection phase* in which it identifies all of the unused cells and returns them to free storage. One way to view garbage collection is that it ignores the storage reclamation problem until a crisis develops, and then it deals with the crisis.

There are many techniques for garbage collection and many analyses of garbage collection under varying circumstances. In this section we will deal briefly with the simplest kind of *mark-sweep* garbage collector, which operates in two phases. In the *mark* phase, the garbage collector identifies all of those cells that are accessible, that is, that are not "garbage." In the *sweep* phase, the garbage collector places all of the inaccessible cells in the free storage area, often by placing them on the free-list.

First, we will investigate the mark phase. We noted earlier that all accessible cells must ultimately be reachable from some number of *roots* known to the interpreter, such as the program structure and the association lists representing the active environments. In the mark phase, the garbage collector starts at the roots and follows the pointers, marking each cell it reaches as accessible. For this purpose each cell requires an associated mark bit that indicates whether or not it is accessible. These mark bits may be part of the cell or may be stored in a separate area of memory; we will draw them as though they are part of the cells. At the beginning of the garbage collection process, all cells are unmarked. Whenever the marking process encounters a cell that is already marked, it knows that it need not continue down that path (since it has already marked it). In particular, this guarantees that cycles in the lists will not put the garbage collector into an infinite loop. Marking is depicted in Figure 11.6; the dotted arrows indicate the path traced by the garbage collector.

The algorithm for the mark phase is thus

```
mark phase:
        for each root R, mark (R).
```

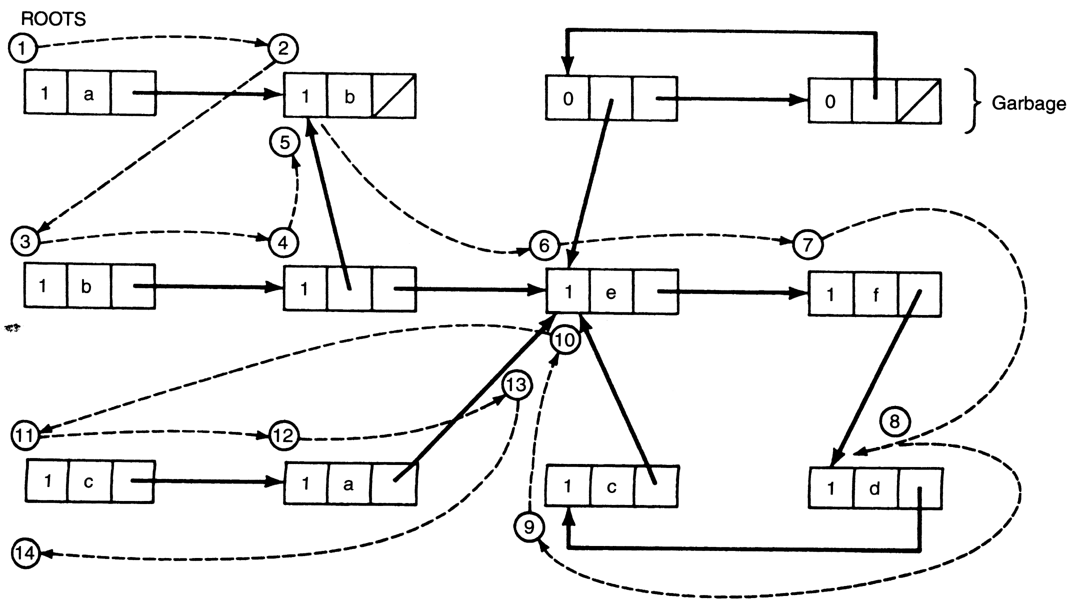

Figure 11.6 Example of Mark Phase of Garbage Collection

```
mark (R):
        if R is not marked then:
           set mark bit of R;
           mark (R↑.left);
           mark (R↑.right);
        endif.
```

Notice that this is a recursive algorithm, and, like all recursive algorithms, it will require space for the activation records of the recursive procedure. You might wonder how the garbage collector could ever run since it is only called in a storage allocation crisis when no free storage is available. There are many solutions to this problem. One is to invoke the garbage collector before the last cell is allocated and when there is still enough space for the garbage collector's stack. Another solution is to encode the stack in some clever way, such as by reversing the links in the marked nodes. (If you are interested in these issues, consult the literature about garbage collectors.)

The second phase of garbage collection is the sweep, when all of the inaccessible cells are returned to free storage. By the end of the mark phase, all accessible cells have been marked. Therefore, in the sweep phase we can visit each cell in order (that is, *sweeping* through memory). If a cell is unmarked, then it is inaccessible, and it can be linked onto the free-list. If it is marked, then it is accessible; we reset its mark bit (in preparation for the next garbage collection) and go on to the next cell. This procedure can be summarized:

```
sweep phase:
         for each cell C:
            if C is marked then reset C's mark bit,
            else link C onto the free-list.
```

■ ***Exercise 11-15*:*** Design a garbage collection algorithm that avoids the necessity for a stack by reversing the links in the list structures.

Garbage Collection Results in Nonuniform Response Time

A serious problem results from garbage collection's crisis approach to storage reclamation. In a large address space, a garbage collection can be quite expensive since it necessitates tracing down all of the lists and visiting every cell in memory. The result is that a program runs along quite nicely until it runs out of storage; then it grinds to a halt for a garbage collection. After the garbage collection (and assuming an adequate amount of storage was reclaimed), it resumes high-speed execution until the next garbage collection. This is quite apparent in an interactive system since, periodically, the system will stop (often for minutes) while a garbage collection is in progress. You can probably imagine how annoying this can be to the users. Under these conditions, garbage collection is anything but transparent; we find ourselves looking *at* the garbage collector rather than *through* it. It is a particularly serious problem in *real time* situations where the program must be guaranteed to respond in a certain amount of time. If a garbage collection happens at the wrong time, it can lead to disaster. (Consider a program responsible for the control surfaces of the Space Shuttle!) This is not as much of a problem with reference counts since the time required for storage reclamation is distributed across the execution of the program. There are also other solutions to this problem, such as parallel garbage collection. With this approach, garbage collection takes place continuously and in parallel with normal program execution. (Once again, see the garbage collection literature if you are interested in these problems.)

11.3 EVALUATION AND EPILOG

LISP Has Been Very Successful in Artificial Intelligence

LISP has become the most commonly used language for artificial intelligence applications. One of the reasons is the same one that motivated its original design: its ability to represent and manipulate complex interrelationships among symbolic data. There are several other reasons that have become apparent through the use of LISP.

LISP Is Suited to Ill-Specified Problems

It is a characteristic of artificial intelligence applications that the problem is not well understood. Indeed, often one goal of the research is to understand the problem better. This characteristic also applies to research and advanced development projects in areas other than artificial intelligence. LISP is very suited to this kind of problem.

One of the characteristics that suits LISP to ill-defined problems is its dynamic type system and flexible data structures. In previous chapters we have seen that one of the recent trends in programming methodology and programming languages has been the specification of *abstract data types*. That is, the programmer first decides what data structures and data types are required and then specifies the necessary operations in terms of their inputs and outputs. Finally, the representation of the data structures and operators is fixed (i.e., the abstract data type is implemented).

On ill-defined problems this methodology does not work so well. Often it is not known exactly what operations on a data type will be required. It is difficult to set down ironclad input-output specifications when the ultimate requirements that the data structure must satisfy are not clear. Rather, a more experimental approach is valuable. Just as in a laboratory, where it is necessary to be able to connect equipment together in a wide variety of ways, in an experimental software environment, flexibility of interconnection is also important. A language such as LISP, with few restrictions on invocation of procedures and passing of parameters, is well suited to experimentation.

LISP Is Easy to Extend, Preprocess, and Generate

The representation of LISP programs as LISP lists, although originally an accident, has turned out to be one of LISP's greatest assets. It has meant that it is very convenient to manipulate LISP programs using other LISP programs. We saw an example of this in the `eval` interpreter. In practice it has influenced the *focus* and *actions* (Section 1.4) of LISP programmers, who have been encouraged to write many programming tools in LISP. These include programs to transform LISP programs into other LISP programs and to generate LISP programs from other notations. It has simplified writing programs that process LISP programs, such as compilers, cross referencers, and optimizers. It has encouraged special-purpose extensions to LISP for pattern matching, text processing, editing, type checking, and so on. Of course, all of these tools could be (and in some cases, have been) provided for conventional languages such as Pascal. However, they tend to be large and complicated because they must deal with a conventional language's complex, character-oriented syntax. Modifying a Pascal or Ada compiler can be a Herculean undertaking; as a result it is usually done only by systems programmers and only when absolutely necessary. In contrast, it is trivial for one LISP program to access the parts of another because LISP has a *simple, structured* syntax. As a result, programmers are encouraged to experiment with sophisticated programming tools.

LISP Programming Environments Developed

The ease with which LISP programs can manipulate other LISP programs has led to the development of a wide variety of LISP programming tools. From these libraries of tools there developed the idea of a *programming environment.* This is a system that supports all phases of programming, including design, program entry, debugging, documentation, and maintenance. It is significant that the first, and richest, programming environments developed around LISP systems.

The Interlisp Programming Environment Grew from an Early LISP System

As an example of an integrated programming environment for LISP we will consider the Interlisp system.[7] The origin of Interlisp was a 1966 implementation of LISP by BBN (Bolt Beraneck and Newman); in 1972 it became a joint project with Xerox PARC (Palo Alto Research Center), and was renamed Interlisp. The system grew and evolved during the 1970s in response to the needs and ideas of its implementors and users. The characteristics of the LISP language combined with an experimental approach to tool development to produce a highly integrated, but loosely coupled and flexible programming environment. In many cases it is a good example of a tool that transparently extends the programmer's reach (recall Section 1.4). Here we can, at best, only touch on its facilities.

"Do What I Mean" Attempts Intelligent Error-Correction

Certainly one of the more notable features of Interlisp is DWIM, the "Do What I Mean" facility. DWIM is invoked by a component of the Interlisp system (e.g., the LISP interpreter, the keyboard command interpreter, or the editor) whenever it detects an error. DWIM attempts to diagnose the cause of the error *in the context in which it occurred* and to make an appropriate correction. In all cases, DWIM attempts to do what intelligent human assistants would do if you made a simple mistake communicating with them.

For example, if the user types an illegal editor command, DWIM will attempt to determine if the cause was a typing error. To this end it has a function for measuring the "closeness" of words as they are typed. For example, words will be close if one could result from the other (1) by replacing a letter by another that is close to it on the keyboard, (2) by exchanging two letters, or (3) by omitting or repeating a letter (as occurs with "key bounce"). If DWIM finds a unique correction (a legal editor command, in this case) that is "sufficiently close," it will correct it automatically; otherwise it will seek confirmation from the user before making the replacement. (All of this is under the control of the user; in practice DWIM makes good corrections, and users come to trust it.)

It is important to understand that DWIM is more than a simple spelling corrector applied to keyboard input; it may be called by any program when it detects an error of any kind. The invoking program is responsible for providing the context to be used for correction. For example, DWIM is invoked by the mail utility, so if the user mistypes a mail address, DWIM will make a context-sensitive correction by consulting its history of e-mail addresses used by that person.

As another example, the LISP interpreter invokes DWIM whenever it detects an error. If it is interpreting a program that invokes an undefined function, DWIM will attempt to determine if the cause is a misspelling of a built-in or user-defined function. Thus `cnos` would be replaced by `cons` and `mak-table` by `make-table`. Again, DWIM is aware of the

[7] A brief discussion of the Interlisp system can be found in "The Interlisp Programming Environment" (*Computer*, April 1981, pp. 25–33) by Warren Teitelman and Larry Masinter. See also Eric Sandewall's "Programming in an Interactive Environment: The LISP Experience" (*ACM Comp. Surveys 10*, 1, March 1978, pp. 33–71).

sorts of parenthesis errors commonly made by programmers (e.g., in writing a `cond`) and will rearrange them appropriately. Notice that these corrections are not purely syntactic; an undefined function is essentially a semantic error. DWIM can be invoked in response to any kind of error. So, for example, if a function is passed an illegal argument, it can invoke DWIM to try to correct it (by doing an implicit type conversion, for example). When DWIM finds a correction to a program, it will change the program in memory (subject to user confirmation, if necessary), since the program is just a LISP data structure, and inform the file system to automatically update the source-code file.

The Programmer's Assistant Saves Work

Another useful component of the Interlisp system, the Programmer's Assistant, continues the theme that the programming environment should be an intelligent assistant to the user. The Programmer's Assistant maintains a "history list" of all commands typed by the user, and permits the user to reuse previous commands in various ways. For example, using the system's natural language-oriented syntax, the user could type

```
use print for read
```

to reexecute the most recent command, but with the atom `read` replaced by `print`. (This is a LISP sublist replacement, not a string replacement.) For more complex modifications, the `fix` command invokes the editor on a previous command.

It is not so unusual now to find command interfaces that provide a history list with some ability to edit the commands (although they are not usually so general, easy to use, or intelligent as the Programmer's Assistant), but other features are still rare. One of these is `undo`, which reverses the effect of any command (not just the most recent) on the history list without undoing subsequent commands. This is remarkable, since a single LISP invocation can have wide-ranging effects, including destructive changes to list structures (such as occur from editing). (Imagine a `mapcar` that performs destructive operations on every element of a long list.)

The `undo` feature is implemented by having the most basic destructive operations keep track of the changes they make and how to reverse them. In the simplest case there are just two destructive operations in LISP: `rplaca` and `rplacd`; all other destructive operations invoke these. Therefore, `rplaca` must record the previous contents of the `left` field, and `rplacd` the previous contents of the `right` field. This information is kept in an global change-list. Whenever a command is executed, the Programmer's Assistant keeps track of the change-list elements contributed by that command. To undo the effect of the command, we reverse the changes recorded in each of those elements. Such a change-list may occupy a lot of (virtual) memory space, but it permits a very reliable implementation of a general `undo` facility. Arbitrary programs can make use of the `undo` facility, provided all their destructive operations are implemented in terms of LISP's (`rplaca`, `rplacd`), or provided their most basic destructive operations are modified to update the change-list.

The Programmer's Assistant is a general-purpose tool that can be invoked from any program that provides a command interface. For example, the editor invokes the Programmer's Assistant, and so `redo`, `undo`, and the other Programmer's Assistant facilities can be used in the editor. Similarly, user programs can invoke the Programmer's Assistant for their user interfaces.

Masterscope Helps Maintain Large Systems

Interlisp was designed for experimental program development, in which it is often unknown, in advance of program implementation, what can be done or even what should be done. In such cases it is often impossible to decide in advance what the interfaces should be. Therefore, in an experimental programming environment, more than in a production environment, it is useful to be able to gather information about a large, complex software system. This capability is provided by Masterscope, which analyzes LISP programs and compiles a database of cross-reference information, which is automatically updated whenever a function is modified. This database can be queried by natural language-like commands, such as

```
who sets count
```

which will tell the user which functions set the variable `count`. Masterscope can also be used to "drive" other programs, such as the editor. For example, we can type

```
edit where any function uses the record symtab
```

to invoke the editor on every function that uses the record `symtab`. This is much more intelligent than a simple search for the string '`symtab`'. Similarly, we could invoke the editor on every function that writes a certain variable, that calls a certain function, etc.

The File System Maintains Consistency among Representations

Among the other facilities in the Interlisp system are a compatible compiler and interpreter, a debugger, and a syntax extension mechanism. The file system ensures consistency between various representations of each data structure. For example, when a function is edited as a list structure in memory (e.g., by the user or as a result of DWIM), a note is made to dump to disk automatically a "pretty printed" definition of the function. Transparency is preserved.

Interlisp Is Integrated But Extensible

Teitelman and Masinter remark that Interlisp is not unique in being "friendly, cooperative and forgiving"; its uniqueness comes from being highly but loosely integrated and in being extensible. Integration results from any tool being able to make use of any other, when it makes sense to do so; it is not necessary to exit one tool to use another. Masterscope can call the editor, the editor can call the LISP interpreter, the interpreter can call the debugger, which can call the editor, and so forth; everything makes use of DWIM and the Programmer's Assistant. Although the tools are integrated, they are loosely coupled, because they are autonomous programs that can be developed and evolve independently.

The evolution and flexibility of Interlisp are enhanced by several different dimensions of flexibility. First, a number of the tools permit users to define new commands in terms of existing commands. In the simplest case these are "substitution macros"; that is, the macro's arguments (which are LISP lists) are substituted for formal parameters in the macro's definition (also a list), which is passed as a command sequence to the tool. More complex extensions can be implemented by "computed macros," which use the full power of LISP to compute the sequence of commands to be forwarded to the tool.

The Interlisp system as a whole is extensible because users are "implicated in the conspiracy," since their programs are allowed to make use of the components of Interlisp. For example, user programs can make use of the Programmer's Assistant and DWIM, call or be called from the editor, etc. Thus Interlisp as a whole grows through the loose integration of additional tools. This is facilitated by the Interlisp philosophy of providing users with "enabling capabilities," which are features a user needs to implement higher level capabilities. For example, when the LISP interpreter encounters an error, it invokes a function called `FaultEval`; that is the enabling capability. Using it, higher level capabilities such as DWIM (and others not yet invented) can be implemented. A more tightly integrated system would not be so easy to extend.

In summary, Interslip demonstrates how the LISP language has facilitated the development and evolution of a comprehensive, integrated, extensible, and flexible programming environment, which remains an inspiring example. In the next chapter we will look at the Smalltalk system, which has many similar positive characteristics. It is probably no coincidence that it too was developed at Xerox PARC.

LISP's Inefficiency Has Discouraged Its Use

With all that we have said, you might ask, "Why doesn't everyone use LISP?" But "all technology is nonneutral." For example, early LISP implementations were quite inefficient. There were several sources of this inefficiency.

First, most LISP systems are interpreted, and interpreters often run two orders of magnitude slower than the code produced by a compiler. This is one reason that many LISP systems now provide compilers and optimizers.

Another reason is that LISP depends heavily on the use of recursion, which (as we saw in Chapter 6) can be fairly complicated to implement. As a result, recursion was inefficient on most machines of the 1960s. More recent machines have included hardware to assist procedure invocation and activation record manipulation for block-structured languages. This improves the performance of implementations of LISP, particularly statically scoped dialects of LISP. In addition, techniques have been discovered that allow many LISP function applications to be replaced by machine branches. Therefore, a loop expressed recursively may be no less efficient than one written using a **goto**.[8]

Certainly, dynamic storage allocation is one of the more expensive aspects of LISP, but it is also one of the most valuable. For this reason, there has been a great deal of work on methods of speeding up both storage allocation and reclamation. One way of doing this has been to include assistance for garbage collection in the hardware. More recently this has led to the development of "LISP machines," that is, computers specially designed for running LISP. Several of these are commercially available.

LISP Continues to Evolve

As already mentioned, a standard Common LISP has been developed. Like many other languages (including Ada and C), LISP has been extended to include facilities for *object-oriented programming* (discussed in the next chapter), which are included in the standard.

[8] See, for example, MacLennan (1990), Section 3.9, for a discussion of these optimizations.

Despite some extensions such as this, the design of Common LISP was restrained by the desire to maintain some compatibility with earlier dialects. Therefore it has retained a number of undesirable characteristics from its ancestors (e.g., dynamic scoping). For this reason attempts have been made to "do LISP right," that is, to design a LISP-like language without the burden of its history. One such attempt was the never-completed LISP-2. A more recent, successful attempt is the Scheme language, which was designed in the 1975–1980 period and evolved over the following decade; it became an ANSI standard in 1991. It is a better vehicle than LISP for functional programming, and is used at some universities (e.g., MIT) to teach beginning programming (see, for example, Sussman and Abelson's *Structure and Interpretation of Computer Programs*).

LISP Shows What Can Be Done with an Applicative Language

LISP is the closest thing to an applicative language in widespread use. Therefore, the experience gained with LISP has been very valuable in evaluating applicative programming languages and applicative programming styles. This experience finds direct application in the development of functional programming languages. We have seen how many ideas, including lists and functionals, have been taken from LISP by functional programming. Again, the LISP experience is evidence in favor of the practicality of functional programming languages. In the future, we can expect the development of new LISP dialects, new functional and applicative languages, and new LISP programming tools and environments.

Characteristics of Function-Oriented Programming Languages

Although LISP is not a purely applicative language, it illustrates many of the characteristics of applicative (function-oriented) programming. We have seen that there is an emphasis on the use of pure functions and minimal use of assignment operations (none in a *purely* applicative language). This leads to the use of recursion as a method of iteration, and Polish (prefix) notation as the basic *syntactic structure.* Applicative languages usually have the list as their principal *data structure*. Use of lists is supported by dynamic (but strong) typing and automatic storage management. The basic *control structures* of applicative programming are the conditional expression and recursion, although many of these languages (especially the specifically *functional* languages) provide higher-level control structures, such as mapping and reduction operators.

Function-oriented languages have many desirable properties. First, their high level of abstraction, especially when functionals are used, suppresses many of the details of programming (in conformance with the Abstraction Principle) and thus removes the possibility of committing many classes of errors. Second, their lack of dependence on assignment operations allows function-oriented programs to be evaluated in many different orders. This evaluation order independence makes function-oriented languages good candidates for programming massively parallel computers. Third, the absence of assignment operations makes function-oriented programs much more amenable to mathematical proof and analysis than are imperative programs. It is difficult to reason about things that change in time. Since function-oriented programming takes place in the timeless realm of mathematics, it avoids the difficulties of temporal reasoning. On the other hand, some applications do not seem to fit easily into this timeless framework. In the next chapter we investigate a very different fifth-generation programming paradigm—one that confronts directly the problem of time by modeling the way objects change in time.

EXERCISES

1*. Implement the `eval` interpreter in a conventional language such as Pascal or Ada.

2*. Write in LISP a recursive interpreter `value` for fully parenthesized infix arithmetic expressions. For example,

```
(value '(3 + ((8 - 2) * 4)) )
27
```

3. Write a function (body *f*) that returns the body of the lambda expression that is the value of the `expr` property of the atom *f*.

4. Using the `body` function defined in the previous exercise, write a LISP program to compile a table giving the frequency of occurrence of each of the functions `car`, `cdr`, and `cons` in a given function.

5*. In Exercise 6 at the end of Chapter 10 you designed a more readable syntax for LISP. Write an `eval` function for programs in this form or write a program to translate programs in this form into the usual LISP notation.

6*. (Difficult) Write a list structure editor in LISP. This editor can also be used for editing LISP functions. The editor should comprise the following procedures:

(edit *L*)	- edit list *L*; focuses attention on the top level of *L*
(down *n*)	- focus attention on *n*th element of *CL* (current list)
(up)	- focus attention on list containing *CL*
(delete *n*)	- delete *n*th element of *CL*
(insert *n x*)	- insert *x* after *n*th element of *CL*

Each command should echo *CL* after making the modification. To keep the echoing brief, only the top three levels of *CL* should be echoed (lower-level lists are printed as the atom <list>). For example,

```
(set 'GP (body 'append))
   (if (null L)
        M
        (cons (car L) (append (cadr L) M)))
(edit GP)
   (if (null L) M (cons <list> M))
(down 4)
   (cons (car L) (append (cadr L) M))
(down 3)
   (append (cadr L) M)
```

```
(delete 2)
   (append M)
(insert 1 '(cdr L))
   (append (cdr L) M)
```

7*. Extend the `eval` interpreter to handle `set` and `defun`. This will require you to implement property lists.

8. Show how `set` could lead to a cyclic list structure.

9*. Write a reference counting storage manager in a conventional language such as C, Pascal, or Ada.

10*. Write a garbage collecting storage manager in a conventional language such as Pascal or Ada. *Hint:* To be able to implement garbage collection, you will have to have an array containing all the list cells and use indices into this array instead of list cell pointers.

11*. Based on your experience in the previous exercise, discuss the problems of writing system software such as storage managers in languages like Pascal and Ada. Would writing a garbage collector be easier in FORTRAN or C?

12*. In the exercises following Chapter 9, you studied `cdr` encoding. Describe a garbage collector for `cdr` encoded lists. Compare its performance with the garbage collector for the usual representation of lists.

13*. Obtain the documentation for a commercial "LISP machine." Evaluate it as an environment for program development.

14*. Read and critique "The INTERLISP Programming Environment" by W. Teitelman and L. Masinter (*Computer 14*, 4, April 1981).

15*. Read and critique "Design of a LISP-Based Microprocessor" by G. L. Steele, Jr., and G. J. Sussman (*Commun. ACM 23*, 11, November 1980).

16*. It has been 20 years since Sandewall's article was written, and nearly that long for Teitelman and Masinter's article (see footnote 7 for citations). Read one or the other of these articles and discuss them relative to current programming practice and contemporary programming environments. Which ideas are still valuable? Which are now obsolete?