9 LIST PROCESSING: LISP

HISTORY AND MOTIVATION

The Fifth Generation Comprises Three Paradigms

We come now to the fifth generation of programming languages, which comprises three overlapping programming paradigms: *functional programming*, *object-oriented programming*, and *logic programming*. We will consider each in turn, beginning with functional programming, which can be illustrated by LISP.

The Desire for an Algebraic List-Processing Language

LISP developed in the late 1950s out of the needs of artificial intelligence programming.[1] In these applications complex interrelationships among data must be represented. The result is that the pointer and, in particular, linked list structures are natural data-structuring methods. In the 1950s Newell, Shaw, and Simon (at Carnegie Institute of Technology and the Rand Corporation) developed many of the ideas of list processing in the IPL family of programming languages. These ideas included the linked representation of list structures and the use of a stack (specifically, a push-down list) to implement recursion.

In the summer of 1956, the first major workshop on artificial intelligence was held at Dartmouth. At this workshop John McCarthy, then at MIT, heard a description of the IPL 2 programming language, which had a low-level pseudo-code, or assembly-language-like syntax. McCarthy realized that an algebraic list-processing language, on the style of the recently announced FORTRAN I system, would be very useful.

[1] The historical information in this section is from McCarthy (1978).

FLPL Was Based on FORTRAN

That summer Gerlernter and Gerberich of IBM were working, with the advice of McCarthy, on a geometry program. As a tool they developed FLPL, the FORTRAN List-Processing Language, by writing a set of list-processing subprograms for use with FORTRAN programs. One result of this work was the development of the basic list-processing primitives that were later incorporated into LISP.

McCarthy Developed the Central Ideas of LISP

Use of FLPL resulted in an important contribution to control structures—the conditional *expression*. Recall that FORTRAN I had only one conditional construct—the arithmetic `IF`-statement. This construct was very inconvenient for list processing, which led McCarthy, in 1957, to write an `IF` function with three arguments. Here is an example invocation:

```
X = IF (N .EQ. 0, ICAR(Y), ICDR(Y))
```

The value of this function was either the second or third argument, depending on whether the first argument was true or false. In the above case, if `N` were zero, then `X` would be assigned the value of `ICAR(Y)`, otherwise it would be assigned the value of `ICDR(Y)`. An important consequence of this invention was that it made it feasible to *compose* `IF` functions and list-processing functions to achieve more complicated actions. We have seen the same *combinatorial power* in Algol's *conditional expression*, for example,

```
x := 0.5 × sqrt (if val < 0 then -val else val)
```

Algol's conditional expression was suggested by McCarthy when he was a member of the Algol committee.

In 1958 McCarthy began using recursion in conjunction with conditional expressions in his definition of list-processing functions. This is a very important idea in LISP, as we will see in Chapter 10, Section 10.1. During the summer of 1958, McCarthy became convinced of the power of the combination of these two constructs. Since FORTRAN does not permit recursive definitions, it became apparent that a new language was needed.

The LISP List-Handling Routines Were Developed First

In the fall of 1958, implementation of a LISP system began. One important component of this was a set of primitive list-handling subroutines for the LISP run-time environment. These were the first parts of the system that were implemented. The original intention was to develop a compiler like FORTRAN. Therefore, to gain experience in code generation, a number of LISP programs were hand compiled into assembly language.

A LISP Universal Function Resulted in an Interpreter

McCarthy became convinced that recursive list-processing functions with conditional expressions formed an easier-to-understand basis for the theory of computation than did other

formalisms such as Turing machines. In his 1960 paper, "Recursive Functions of Symbolic Expressions and Their Computation by Machine," he presented his ideas.

In the theory of computation, it is often important to investigate *universal functions*. For example, a *universal Turing machine* is a Turing machine that can simulate any other Turing machine when given a description of the latter machine. Given a universal Turing machine, it becomes possible to prove that certain properties hold of all Turing machines by proving that they hold of the universal Turing machine.

McCarthy did just that with LISP. He defined a universal LISP function that could interpret any other LISP function. That is, he wrote a LISP interpreter in LISP. We will see this interpreter in Chapter 11.

Since LISP manipulates only lists, writing a universal function required developing a way of representing LISP programs as list structures. For example, the function call

```
f [x+y; u*z]
```

would be represented by a list whose first element is '`f`' and whose second and third elements are the lists representing '`x+y`' and '`u*z`'. In LISP this list is written

```
(f (plus x y) (times u z))
```

The Algol-like notation (e.g., `f[x+y; u*z]`) is called *M-expressions* (*M* for meta-language), and the list notation is called *S-expressions* (*S* for symbolic language).

Once the list representation was designed and the universal function was written, one of the project members realized that the group had, in effect, an interpreter. Therefore, he translated the universal function into assembly language and linked it with the list-handling subroutines. The result was the first working LISP system.

This system required programs to be written in the *S*-expression notation, but this was seen as a temporary inconvenience. The Algol-like LISP 2 system then being designed would permit the use of the *M*-expression notation. This system, however, was never completed, and LISP programmers still write their programs in *S*-expressions. Although this was unintentional, we will see later that it is now recognized as one of the main advantages of LISP.

LISP Became Widely Used in Artificial Intelligence

The first implementation of LISP was on the IBM 704 (the same machine that hosted the first implementation of FORTRAN). A prototype interactive LISP system was demonstrated in 1960 and was one of the earliest examples of interactive computing. LISP systems rapidly spread to other computers, and they now exist on virtually all machines, including microcomputers. LISP has become the most widely used programming language for artificial intelligence and other symbolic applications. McCarthy claims that LISP is second only to FORTRAN in being the oldest programming language still in widespread use.

LISP Was Standardized after a Period of Divergent Evolution

As LISP use spread throughout the artificial intelligence community, its flexibility encouraged many groups to extend the interpreter to include new features. The result was a plethora of LISP dialects belonging to two loose families, often called "East Coast LISP," of which

MacLisp (developed at MIT) was the best known member, and "West Coast LISP," of which Interlisp (developed by BBN and Xerox in California) was the best known. In 1981 ARPA sponsored a "LISP Community Meeting" to define the future direction of LISP, and out of it came the plan to develop a "Common LISP" dialect to reconcile the differences between the major dialects. The language design was completed in 1982 and 1983, and has since become the most widely available LISP dialect. Standardization efforts began in 1985 and resulted in 1992 in a draft ANSI standard over 1000 pages long. Needless to say, although we will devote three chapters to Common LISP, we will consider only its most important features, especially those that illustrate recursive list processing and functional programming.

9.2 DESIGN: STRUCTURAL ORGANIZATION

An Example LISP Program

Figure 9.1 shows an example LISP program in the *S*-expression notation. The purpose of this program is to generate a "frequency table" that records the number of times a particular word appears in a given list of words. Three functions are defined: `make-table` and its auxiliary function `update-entry` construct the frequency table; `lookup` uses the resulting frequency table to determine the number of occurrences of a given word. The session shown also defines `text` to be a particular list of words (viz., `'to be or not to be'`) and `Freq` to be the frequency table computed from this list.

```
(defun make-table (text table)
  (if (null text)
      table
      (make-table (cdr text)
                  (update-entry table (car text)) )) )

(defun update-entry (table word)
  (cond ((null table) (list (list word 1)) )
        ((eq word (caar table))
        (cons (list word (add1 (cadar table)))
              (cdr table)))
        (t (cons (car table)
                 (update-entry (cdr table) word))) ))

(defun lookup (table word)
  (cond ((null table) 0)
        ((eq word (caar table)) (cadar table))
        (t (lookup (cdr table) word)) ))

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

(set 'Freq (make-table text nil))
```

Figure 9.1 Example of LISP Program

(Do not expect to be able to read this program unless you have had previous LISP experience. It is included here so that you can see the general appearance of LISP programs.)

Function Application Is the Central Idea

Programming languages are often divided into two classes. *Imperative languages*, which include all of the languages we have discussed so far, depend heavily on an assignment statement and a changeable memory for accomplishing a programming task. In previous chapters we have pointed out that most programming languages are basically collections of mechanisms for routing control from one assignment statement to another.

In an *applicative language*, the central idea is function application, that is, applying a function to its arguments. Previous chapters have shown some of the power of function application. For example, with it we can eliminate the need for control structures, as we did when we defined the `Sum` procedure using Jensen's device (Chapter 3, Section 3.5). Also, in Chapter 8 we saw how even the built-in operators can be considered function applications. LISP takes this approach to the extreme; almost everything is a function application.

To see this, it is necessary to know that a function application in LISP is written as follows:

$(f\ a_1\ a_2\ \ldots\ a_n)$

where f is the function and $a_1, a_2, \ldots, a_n$ are the arguments. This notation is called *Cambridge Polish* because it is a particular variety of Polish notation developed at MIT (in Cambridge, MA). *Polish notation* is named after the Polish logician Jan Łukasiewicz.

The distinctive characteristic of Polish notation is that it writes an operator before its operands. This is also sometimes called *prefix notation* because the operation is written before the operands (pre = before). For example, to compute 2 + 3 we would type[2]

```
(plus 2 3)
```

to an interactive LISP system, and it would respond

```
5
```

LISP's notation is a little more flexible than the usual infix notation since one `plus` can sum more than two numbers. We can write

```
(plus 10 8 5 64)
```

for the sum 10 + 8 + 5 + 64. Also, since LISP is *fully parenthesized*, there is no need for the complicated precedence rules found in most programming languages.

Notice that the example in Figure 9.1 consists of all function applications (which is why there are so many parentheses). For example,

```
(set 'Freq (make-table text nil))
```

[2] Some LISP dialects, including Common LISP, permit typing `(+ 2 3)`.

is a nested function application: `set` is applied to two arguments: (1) `Freq` and (2) the result of applying `make-table` to the arguments `text` and `nil`. (The function of the quote before `Freq` is discussed on p. 315.) In an Algol-like language this would be written

```
set (Freq, make-table (text, nil))
```

Many constructs that have a special syntax in conventional languages are just function applications in LISP. For example, the conditional expression is written as an application of the `cond` function. That is,

```
(cond
   ((null x) 0)
   ((eq x y) (f x))
   (t (g y)) )
```

evaluates `(null x)`; if it is true, then `0` is returned. Otherwise we test `(eq x y)`; if this is true, then the value of `(f x)` is returned. If neither of the above is true, then the value of `(g y)` is the result. This would be written in an Algol-like language as follows:

```
if     null(x) then 0
elsif  x = y then f(x)
else   g(y) endif
```

We can see that even function definition is accomplished by calling a function, `defun`, with three arguments: the name of the function, its formal parameter list, and its body.

Why is everything a function application in LISP? There are a number of reasons that will be discussed later, but we will mention one, the Simplicity Principle, here. If there is only one basic mechanism in a language, the language is (other things being equal) easier to learn, understand, and implement.

The List Is the Primary Data Structure Constructor

We have said that one of LISP's goals was to allow computation with *symbolic data.* This is accomplished by allowing the programmer to manipulate *lists* of data. An example of this is the application

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

The second argument to `set` is the list

```
(to be or not to be)
```

(Ignore the quote mark for the time being; it will be explained on p. 315.) LISP manipulates lists just like other languages manipulate numbers; they can be compared, passed to functions, put together, and taken apart. In Section 9.3 (Data Structures), we will discuss in detail the ways that lists can be manipulated.

The list above is composed of four distinct *atoms*:

```
to be or not
```

arranged in the list in the order

```
(to be or not to be).
```

LISP provides operations for putting atoms together to make a list and for extracting atoms out of a list.

LISP also allows lists to be constructed from other lists. For example, the list

```
((to be or not to be) (that is the question))
```

has two *sublists*; the first is `(to be or not to be)` and the second is `(that is the question)`.

The list is the *only* data structure constructor originally provided by LISP, another example of the *Simplicity Principle. If there is only one data structure in our language, then* there is only one about which to learn and only one to choose when programming our application. Although the list is not always the best data structure for an application, it is remarkably versatile. More recent dialects of LISP, including Common LISP, provide a variety of data structures, including arrays and structures (records).

Programs Are Represented as Lists

Function applications and lists look the same. That is, the *S*-expression

```
(make-table text nil)
```

could either be a three-element list whose elements are the atoms `make-table`, `text`, and `nil`; or it could be an application of the function `make-table` to the arguments named `text` and `nil`. Which is it?

The answer is that it is both because a LISP program is itself a list. Under most circumstances an *S*-expression is interpreted as a function application, which means that the arguments are *evaluated* and the function is invoked. However, if the list is *quoted*, then it is treated as data; that is, it is *unevaluated.* The function of the prefixed quote mark in

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

is to indicate that the list `(to be or not to be)` is treated as data, not as a function application. If it had been omitted, as in

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

then the LISP interpreter would have attempted to call a function named '`to`' with the arguments named '`be`', '`or`', '`not`', '`to`', and '`be`'. This would, of course, be an error if, as in this case, these names were undefined. In any case, it is not what we intended; the list is supposed to represent *data* not *program.*

The fact that LISP represents both programs and data in the same way is of the utmost importance (and almost unique among programming languages). As we will see, it makes it very easy to write a LISP interpreter in LISP. More important, it makes it convenient to have one LISP program generate and call for the execution of another LISP program. It also simplifies writing LISP programs that transform and manipulate other LISP programs. These ca-

pabilities are important in artificial intelligence and other advanced program-development environments. Nevertheless, these *amplifications* come with a *reduction* of readability (cf. Section 1.4).

LISP Is Often Interpreted

Most LISP systems provide interactive interpreters; in fact, they were some of the earliest interactive systems. We interact with the LISP interpreter by typing in function applications. The LISP system then interprets them and prints out the result. For example, if we type

```
(plus 2 3)
```

the system will respond

```
5
```

because 2 + 3 = 5. Similarly, if we type

```
(eq (plus 2 3) (difference 9 4))
```

the system will respond

```
t
```

(meaning *true*) because 2 + 3 = 9 − 4.

Functions like `eq` and `plus` are called *pure functions* (or simply *functions*) because they have no effect other than the computation of a value. Pure functions obey the Manifest Interface Principle because their interfaces (i.e., their inputs and outputs) are apparent (*manifest*).

The Manifest Interface Principle

All interfaces should be apparent (manifest) in the syntax.

Some functions in LISP are *pseudo-functions* (or *procedures*). These are functions that have a *side effect* on the state of the computer in addition to computing a result. A simple example is `set`, which binds a name to a value. The application `(set 'n x)` binds the name (atom) `n` to the value of `x` and, almost incidently, returns this value. Thus, if we type

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

the LISP system will print the result:

```
(to be or not to be)
```

More important, the name `text` is bound to this list, which we can see by typing

```
text
```

LISP will then respond

```
(to be or not to be)
```

The atom `text` can now be used as a name for this list in any expression, for instance

```
(set 'Freq (make-table text nil))
```

Another important pseudo-function is `defun`, which defines a function. The application

(defun *f* (*n*$_1$ *n*$_2$ ··· *n*$_m$) *b*)

defines a function (or pseudo-function) with the name *f*; formal parameters n_1, n_2, ... , n_m; and body *b*. It is thus analogous to the Algol declaration

procedure *f* (*n*$_1$, *n*$_2$, ... , *n*$_m$); *b*;

(although in LISP the binding process is dynamic, that is, it takes place at run-time).

This is a good place to mention that LISP dialects differ from each other in many small ways. In particular, the function application used to define functions is different in many dialects, although you should have no trouble relating them to the form used here. (In this book we use the Common LISP dialect; see Steele, 1984.)

Most LISP programs (such as our example in Figure 9.1) take the form of a collection of function definitions and `set`s. Users can then apply the defined functions from their terminals. For example, the application

```
(lookup Freq 'be)
```

results in the value

```
2
```

the number of occurrences of `be` in the list.

9.3 DESIGN: DATA STRUCTURES

The Primitives Include Numeric Atoms

As we have done on all the languages we have investigated, we classify LISP's data structures into *primitives* and *constructors.* The principal *constructor* is the list; it permits more complicated structures to be built from simpler structures. The *primitive* data structures are the starting points for this building process. Thus, the primitive data structures are those data structures that are not built from any others; they have no parts. It is for this reason that they are called *atoms* ('atom' in Greek = indivisible thing).

There are at least two types of atoms in all LISP systems. We have already seen examples of *numeric* atoms, which are atoms having the syntax of numbers (i.e., all digits with possibly one decimal point). Various arithmetic operations can be applied to numeric atoms. For example,

```
(plus 2 3)
```

applies the function `plus` to the atoms `2` and `3` and yields the atom `5`.

LISP provides a very large set of primitive functions for manipulating numeric atoms. These include the arithmetic operations (`plus`, `difference`, etc.), predecessor and suc-

cessor functions (`sub1`, `add1`), maximum and minimum functions, relational tests (`equal`, `lessp`, `greaterp`), and predicates (i.e., tests, such as `zerop`, `onep`, `minusp`). All of these functions take both integer and floating-point (and, in some systems, multiple-precision) arguments and return results of the appropriate type.

LISP's use of Cambridge Polish limits its numerical applications. For example, the expression

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

must be written

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

Common LISP (and many other dialects) permits symbolic operations, but it is not much of an improvement:

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

On the other hand, it is fairly easy to write a LISP function to translate conventional infix notation into LISP's prefix notation. If we defined such a function and called it `infix`, we could write

```
(infix '((- b + sqrt (b ↑ 2 - 4 * a * c)) / (2 * a)))
```

Exercise 9-1*: LISP basically provides one numeric type and converts between integer and floating-point (and possibly multiple-precision) representations, as necessary, at run-time. This is very different from the other languages we've discussed, wherein numeric representations are fixed at compile-time. Discuss the advantages and disadvantages of LISP's approach to numbers.

Exercise 9-2: Translate the following expressions into LISP:

1. $^1/_2\sqrt{4r^2 - l^2}$
2. $\dfrac{abc}{4\sqrt{s(s-a)(s-b)(s-c)}}$
3. $\dfrac{n!}{r!(n-r)!}$, use `(fac n)` for $n!$
4. $\dfrac{\pi R^2 E}{180}$

Nonnumeric Atoms Are Also Provided

The other kind of primitive data structure in LISP is the *nonnumeric atom.* These atoms are strings of characters that were originally intended to represent words or symbols. We saw

nonnumeric atoms in the list `(to be or not to be)`. With few exceptions, the only operations that can be performed on nonnumeric atoms are comparisons for equality and inequality. This is done with the function `eq`:

```
(eq x y)
```

returns `t` (an atom meaning true) if x and y are the same atom, and `nil` (an atom meaning, among other things, false) if they are not the same.

The atom `nil` has many uses in LISP; we will see more of them later. One very common operation in LISP is testing something to see if it is `nil`; this operation is often used as a base for recursive definitions. Although this test can be written

```
(eq x nil)
```

it is so frequent that a special predicate has been provided:

```
(null x)
```

Notice that `nil` is the noun and `null` is the corresponding adjective.

Some LISP systems provide additional types of atoms, such as strings. In these cases special operations for manipulating these values are also provided. Recall that an abstract data type is a set of data values together with a set of operations on those values.

The Principal Constructor Is the List

The characteristic method of data structuring provided by LISP is called the *list*. Lists are written in the *S*-expression notation by surrounding with parentheses the list's elements, which are separated by blanks. Lists can have none, one, or more elements, so they satisfy the Zero-One-Infinity Principle. The elements of lists can themselves be lists, so the Zero-One-Infinity Principle is also satisfied by the nesting level of lists.

For a historical reason relating to the first LISP implementation, the empty list, `()`, is considered equivalent to the atom `nil`. That is,

```
(eq '() nil)
(null '() )
```

are both true (i.e., return `t`). For this reason, the empty list is often called the *null list*. Except for the null list, all lists are *nonatomic* (i.e., not atoms); they are sometimes called *composite* data values. We can find out whether or not something is an atom by using the `atom` predicate; for example.

```
(atom 'to)
  t
(atom 5)
  t
(atom (plus 2 3) )
  t
(atom nil)
  t
```

```
(atom '() )
  t
(atom '(to be) )
  nil
(atom '(()) )
  nil
```

(We indent the responses of the LISP system so we can differentiate them from what the user types.)

Notice that the null list is a definite object and must be distinguished from "nothing." In particular, neither `(())` nor `(nil)` is the null list; rather, each is a one-element list containing the null list. If you are familiar with set theory, you will see that this is analogous to the difference between $\varnothing$, the null set, and $\{\varnothing\}$, a nonnull set containing one element, the null set.

■ ***Exercise 9-3:*** Explain the result returned by each of the applications of `atom` shown above.

Car and Cdr Access the Parts of Lists

We have described the kind of data values that lists are. We have seen in previous chapters, however, that there's much more to a data type than just data values. An *abstract data type* is a set of data values *together* with a set of operations on those data values. What are the primitive list-processing operations?

A complete set of operations for a composite data type, such as lists, requires operations for building the structures and operations for taking them apart. Operations that build a structure are called *constructors*,[3] and those that extract their parts are called *selectors*. LISP has one constructor—*cons*—and two selectors—*car* and *cdr*.

The first element of a list is selected by the `car` function.[4] For example,

```
(car '(to be or not to be) )
```

returns the atom `to`. The first element of a list can be either an atom or a list, and `car` returns it, whichever it is. For example, since `Freq` is the list

```
( (to 2) (be 2) (or 1) (not 1) )
```

the application

```
(car Freq)
```

returns the list

```
(to 2)
```

---

[3] This is not a new meaning for the term "constructor." We have said that constructors are used to build structures from the primitives or from simpler structures. This applies to structures of all sorts: name structures, data structures, control structures, and now list structures.

[4] This is pronounced "cahr." The historical reasons for this name are discussed later in this chapter.

Notice that the argument to `car` is always a nonnull list (otherwise it cannot have a first element) and that `car` may return either an atom or a list, depending on what its argument's first element is.

Since there are only two selector functions and since a list can have any number of elements, any of which we might want to select, it is clear that `cdr` must provide access to the rest of the elements of the list (after the first).

The `cdr`[5] function returns all of a list *except* its first element. Therefore,

```
(cdr '(to be or not to be) )
```

returns the list

```
(be or not to be)
```

Similarly, `(cdr Freq)` returns

```
( (be 2) (or 1) (not 1) )
```

Notice that, like `car`, `cdr` requires a nonnull list for its argument (otherwise we cannot remove its first element). Unlike `car`, `cdr` *always* returns a list. This could be the null list; for example, `(cdr '(1))` returns `()`.

It is important to realize that both `car` and `cdr` are *pure functions*, that is, they do not modify their argument list. The easiest way to think of the way they work is that they make a new copy of the list. For example, `cdr` does not delete the first element of its argument; rather, it returns a new list exactly like its argument except without the first element. We will see later when we discuss the implementation of LISP lists that this copying does not actually have to be done; however, it is a good cognitive model when first learning about LISP.

`Car` and `cdr` can be used in combination to access the components of a list. Suppose `DS` is a list representing a personnel record for Don Smith:

```
(set 'DS '( (Don Smith) 45 30000 (August 25 1980) ) )
```

The list `DS` contains Don Smith's name, age, salary, and hire date. To extract the first component of this list, his name, we can write `(car DS)`, which returns `(Don Smith)`. How can we access Don Smith's age? Notice that the `cdr` operation deletes the first element of the list, so that the second element of the original list is the first element of the result of `cdr`. That is, `(cdr DS)` returns

```
(45 30000 (August 25 1980))
```

so that `(car (cdr DS))` is `45`, Don Smith's age. We can now see the general pattern: To access an element of the list, use `cdr` to delete all of the preceding elements and then use `car` to pick out the desired element. Therefore, `(car (cdr (cdr DS)))` is Don Smith's salary, and

```
(car (cdr (cdr (cdr DS))))
```

---

[5] `Cdr` is pronounced "could-er."

is his hire date. We can see this from the following (by 'cdr →' we mean "applying cdr returns"):

```
( (Don Smith) 45 30000 (August 25 1980) )
cdr → (45 30000 (August 25 1980) )
cdr → (30000 (August 25 1980) )
cdr → ( (August 25 1980) )
car → (August 25 1980)
```

In general, the *n*th element of a list can be accessed by $n - 1$ cdrs followed by a car. Since (car DS) is this person's name (Don Smith), his first name is

```
(car (car DS))
  Don
```

and his last name is

```
(car (cdr (car DS)))
  Smith
```

We can see that any part of a list structure, no matter how complicated, can be extracted by appropriate combinations of car and cdr. This is part of the simplicity of LISP; just these two selector functions are adequate for accessing the components of any list structure. This can, of course, lead to some large compositions of cars and cdrs, so LISP provides an abbreviation. For example, an expression such as

```
(car (cdr (cdr (cdr DS))))
```

can be abbreviated

```
(cadddr DS)
```

The composition of cars and cdrs is represented by the sequence of 'a's and 'd's between the initial 'c' and the final 'r'. By reading the sequence of 'a's and 'd's in reverse order, we can use them to "walk" through the data structure. For example, caddr accesses the salary:

```
((Don Smith) 45 30000 (August 25 1980))
d → (45 30000 (August 25 1980))
d → (30000 (August 25 1980))
a → 30000
```

Also, cadar accesses the last-name component of the list:

```
((Don Smith) 45 30000 (August 25 1980))
a → (Don Smith)
d → (Smith)
a → Smith
```

This can be seen more clearly if the list is written as a linked data structure; then a 'd' moves to the right and an 'a' moves down (see Figure 9.2). This shows that (caddadddr DS)

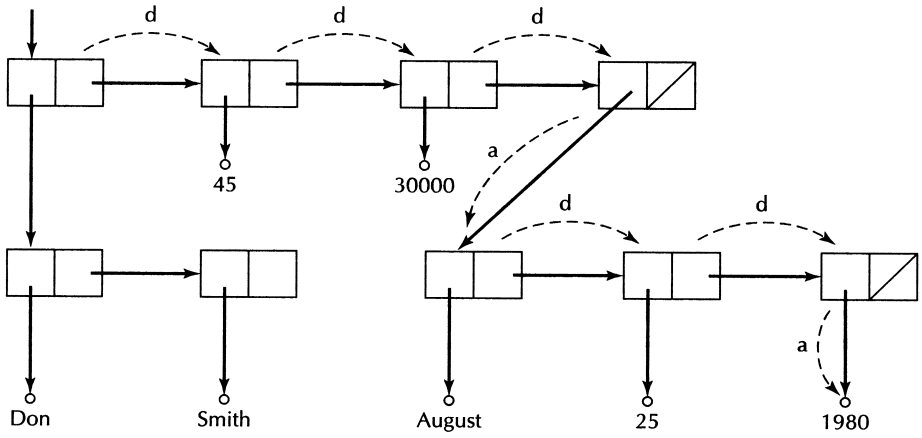


Figure 9.2 Walking Down a List Structure

accesses the year Don Smith was hired. Clearly, these sequences of 'a's and 'd's can become quite complicated to read. Writing them is also error-prone. One solution to this is to write a set of special-purpose functions for accessing the parts of a record. For example, a function for accessing the hire date could be defined as

```
(defun hire-date (r) (cadddr r))
```

Then `(hire-date DS)` returns Don Smith's hire date. Similarly, we could define `year` to give the year part of a date:

```
(defun year (d) (caddr d))
```

Then Don Smith's year of hire can be written:

```
(year (hire-date DS))
```

which is certainly more readable than `(caddadddr DS)`. It is also more maintainable since if we later change the format of this list (say, by adding a new piece of information), it will be necessary to change only the accessing functions. In other words, we should think *of these personnel records as an abstract data type that can only be accessed through the provided accessing functions.*[6]

■ ***Exercise 9-4:*** Define the functions `name`, `age`, `salary`, and `hire-date` for acessing the parts of a personnel record; the functions `firstn` and `lastn` for accessing the parts of a name; and the functions `month`, `day`, and `year` for accessing the parts of a date. Write expressions for accessing Don Smith's last name, salary, and the month in which he was hired.

[6] In fact, there is no need for the programmer to define records in this manual way. Common LISP provides a powerful *structure* facility for defining record types. Our goal here is to illustrate list processing in LISP.

Information Can Be Represented by Property Lists

A personnel record would probably not be represented in LISP in the way we have just described: It is too inflexible. Since each property of Don Smith is assigned a specific location in the list, it becomes difficult to change the properties associated with a person. A better arrangement is to precede each piece of information with an *indicator* identifying the property. For example,

```
(name (Don Smith) age 45 salary 30000 hire-date (August 25 1980))
```

This method of representing information is called a *property list* or *p-list*. Its general form is

$(p_1\ v_1\ p_2\ v_2, \ldots\ p_n\ v_n)$

in which each p_i is the indicator for a property and each v_i is the corresponding property value.

The advantage of property lists is their flexibility; as long as properties are accessed by their indicators, programs will be independent of the particular arrangement of the data. For example, the *p*-list above represents the same information as this one:

```
(age 45 salary 30000 name (Don Smith) hire-date (August 25 1980))
```

This flexibility is important in an experimental software environment in which all of the relevant properties may not be known at design time.

Information is selected from a simple list by various compositions of `car` and `cdr`. How can the properties of a *p*-list be accessed? We can attack this problem by considering how a person might solve it. Asked Don Smith's age, a person would probably begin searching from the left of the list for the indicator `age`. The following element of the list is Don Smith's age. Let's consider this process in more detail: Exactly how do we search a list from the left? We begin by looking at the first element of the list; if it is `age`, then the second element of the list is Don Smith's age, so we return it and we are done. If the first element of the list is not `age`, then we must skip the first two elements (the first property and its value) and repeat the process by checking the new first element.

Let's begin to express this in LISP notation. Suppose p is the property for which we are looking and x is the object that we are searching. We will write a `getprop` function such that `(getprop` p x`)` is the value of the p property in property list x. First, we want to see if the first element of x is p; we can do this by `(eq (car` x`)` p`)`. If the first element of x is p, then the value of `(getprop` p x`)` is the second element of x, that is, `(cadr` x`)`. If the first element of x is not p, then we want to skip over the first property of x and its value and continue looking for p. Notice that `(getprop` p `(cddr` x`))` will look for p beginning with the third element of x. We can summarize our algorithm as follows:

```
(getprop p x) =
  if (eq (car x) p)
    then return (cadr x)
    else return (getprop p (cddr x))
```

It remains only to translate this into LISP notation.

The simplest LISP conditional expression is written `(if` C T F`)`. The condition C is

evaluated. If it is true (i.e., returns any value except `nil`), then the value of the expression is the value of *T*, otherwise its value is the value of *F*. Thus LISP's `if` is essentially the same as Algol-60's conditional expression (p. 127). The `getprop` function is

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

This definition, like most LISP definitions, is recursive. To find out properties of `DS`, we can now use `getprop`:

```
(getprop 'name DS)
   (Don Smith)
(getprop 'age DS)
   45
(getprop 'hire-date DS)
   (August 25 1980)
(year (getprop 'hire-date DS))
   1980
```

Notice that the name of the property is quoted; we want the actual atom `age`, not some value to which this atom might be bound.

What will this function do if we ask for a property that is not in the property list? By tracing the execution of the function we will see that eventually we will have skipped all the properties and will be asking if the first element of the null list is the indicator we are seeking. Specifically, we will attempt to take the `car` of `nil`. Since this is illegal on most LISP *systems, we will get an error:*

```
(getprop 'weight DS)
   Error: Car of nil.
```

Often we do not know exactly what properties an object has so it would be more convenient if `getprop` were more forgiving. One way to do this is to have `getprop` return a distinguished value if the property does not exist. An obvious choice is `nil`, but this would not be a good choice since it would then be impossible to distinguish an undefined property from one that is defined but whose value is `nil` (e.g., meaning false). `Nil` is too common a value in LISP for this to be a good choice. A better decision is to pick some atom, such as `undefined-property`, which is unlikely to be used for any other purpose. The `getprop` function can be modified to return this atom when it gets to a null list in its search process:

```
(getprop 'weight DS)
   undefined-property
```

■ ***Exercise 9-5:*** Modify the `getprop` procedure so that it returns `undefined-property` when the requested property is not defined in the property list.

■ ***Exercise 9-6:*** The `getprop` procedure provided by some LISP systems is not exactly

like the one we have described. Rather, it is defined to return all of the property list *after* the indicator. For example,

```
(getprop 'age DS)
   (45 salary 30000 hire-date (August 25 1980))
(getprop 'salary DS)
   (30000 hire-date (August 25 1980))
```

Thus, the value of the property is the `car` of the result of this `getprop`. This approach permits a different solution to the undefined property problem; the `getprop` procedure returns `nil` if the property is not in the list. That is,

```
(getprop 'weight DS)
   nil
```

Define this version of the `getprop` procedure.

Information Can Be Represented in Association Lists

The property list data structure works best when exactly one value is to be associated with each property. That is, a property list has the form

$(p_1\ v_1\ p_2\ v_2\ \ldots\ p_n\ v_n)$

This is sometimes inconvenient; for example, some properties are *flags* that have no associated value—their presence or absence on the property lists conveys all of the information. In our personnel record example, this might be the `retired` flag, whose membership in the property list indicates that the employee has retired. Since property indicators and values must alternate in property lists, it is necessary to associate *some* value with the `retired` indicator, even though it has no meaning.

An analogous problem arises if a property has several associated values. For example, the `manages` property might be associated with the names of everyone managed by Don Smith. Because of the required alternation of indicators and values in property lists, it will be necessary to group these names together into a subsidiary list.

These problems are solved by another common LISP data structure—the *association list*, or *a-list*. Just as we can associate two pieces of information in our minds, an association list allows information in list structures to be associated. An *a*-list is a list of pairs,[7] with each pair associating two pieces of information. The *a*-list representation of the properties of Don Smith is

```
( (name (Don Smith))
  (age 45)
  (salary 30000)
  (hire-date (August 25 1980)) )
```

[7] Actually, an *a*-list is normally defined to be a list of *dotted* pairs. We will not address this detail until later.

The general form of an *a*-list is a list of attribute-value pairs:

$((a_1\ v_1)\ (a_2\ v_2)\ \ldots\ (a_n\ v_n))$

As for property lists, the ordering of information in an *a*-list is immaterial. Information is accessed *associatively;* that is, given the indicator `hire-date`, the associated information `(August 25 1980)` can be found. It is also quite easy to go in the other direction: Given the "answer" `(August 25 1980)`, find the "question," that is, `hire-date`. The function that does the forward association is normally called `assoc`. For example,

```
(set 'DS '((name (Don Smith) (age 45) ...))
  ((name (Don Smith)) (age 45) ...)
(assoc 'hire-date DS)
  (August 25 1980)
(assoc 'salary DS)
  30000
```

- ***Exercise 9-7:*** Write the `assoc` function in LISP. You will have to decide how to handle the case where the requested attribute is not associated by the *a*-list. Justify your solution.

- ***Exercise 9-8:*** Write the function `rassoc` that performs "reverse association"; that is, given the attribute's value it returns the attribute's indicator. For example,

```
(rassoc 45 DS)
  age
(rassoc '(August 25 1980))
  hire-date
```

  How will you deal with the fact that several attributes might have the same value? Justify your solution.

- ***Exercise 9-9:*** Write a function `length` such that `(length L)` is the number of (top-level) components in the list *L*. For example,

```
(length '(to be or not to be))
  6
(length Freq)
  4
(length '() )
  0
```

Cons Constructs Lists

We have seen that the `car` and `cdr` functions can be used to extract the parts of a list. How are lists constructed? When we design an abstract data type, we should make sure that the constructors and selectors work together smoothly. This is necessary if the data type is to be

easy to learn and easy to use. In particular, the data type will be more *regular* if the constructors and selectors are *inverses*; that is, the selectors undo what the constructors do, and vice versa. In the case of lists, notice that the `car` and `cdr` functions operate at the *beginnings* of lists; `car` selects the first element of a list and `cdr` removes the first element of a list. It is natural then to pick a constructor function that operates at the beginning of a list and reverses the selectors. LISP's only constructor, `cons`,[8] adds a new element to the beginning of a list. For example,

```
(cons 'to '(be or not to be))
```

returns the list `(to be or not to be)`. Notice that `cons` is the inverse of `car` and `cdr`:

```
(car '(to be or not to be))          = to
(cdr '(to be or not to be))          = (be or not to be)
(cons 'to '(be or not to be))        = (to be or not to be)
```

Therefore, any list that we can construct we can also take apart, and any list that we can take apart can be reassembled from its parts.

What is the meaning of `(cons '(a b) '(c d))`? You might think it is `(a b c d)`. But if we consider the inverse relation between `cons` and the selectors, we can see that the correct answer is `((a b) c d)`. Observe:

```
(car '((a b) c d) ) = (a b)
(cdr '((a b) c d) ) = (c d)
```

because `car` returns the first element of `((a b) c d)`, which is the list `(a b)`. It then follows that

```
(cons '(a b) '(c d)) = ((a b) c d)
```

These relationships among `car`, `cdr`, and `cons` can be summarized in the equations in Figure 9.3. Notice that the second argument of `cons` must be a list, although the first argument can be either an atom or a list.

Like `car` and `cdr`, `cons` is a *pure function.* That means that it does not actually add a new element to the beginning of its second argument. Rather, it acts as though it has constructed a completely new list, whose first element is the first argument to `cons` and the remainder of whose elements are copied from its second argument. We will see later that the actual implementation of `cons` is much more efficient than suggested by this description.

Lists Are Usually Constructed Recursively

We saw that the value of `(cons '(a b) '(c d))` was *not* `(a b c d)`. Suppose, however, that we want to concatenate two lists, and that `(a b c d)` is the required result from `(a b)` and `(c d)`. How can this be accomplished? Investigating the program-

[8] `Cons` is pronounced "konss," like the first syllable of "construct."

```
(cons (car L) (cdr L)) = L,  for nonnull L
(car (cons x L)) = x
(cdr (cons x L)) = L
```
where `L` is a list and `x` is an atom or list.

Figure 9.3 Equations for the List Data Type

ming of a function to do this will furnish a good example of the use of recursion to construct lists. Our goal will be to develop a function `append` such that

```
(append '(a b) '(c d)) = (a b c d)
```

In general, `(append` *L M*`)` will return the concatenation of the lists *L* and *M*.

A good way to start with a problem like this is, first, to identify those cases that are easy to solve, and then to try to reduce the other cases to the easy-to-solve cases. In other words, we should ask ourselves which cases can be solved with the functions already available. In this situation, we can see that if either of the lists to be appended is null, then the result is the other list. That is,

```
(append '() L) = L
(append L '()) = L
```

These are the easily solved cases. In LISP the null case is frequently the easy-to-solve case.

We want to proceed by reducing the unsolved cases to the solved cases. That is, we want to work the nonnull lists toward null lists. More specifically, if we can reduce the problem of appending a list of length n to the problem of appending a list of length $n - 1$, then we can continue this process until we are appending a list of length 0, which is the null list and already solved. This process is very much like an inductive proof in mathematics.

Let's consider the specific case of appending the three-element list `(a b c)` to the three-element list `(d e f)`. Suppose the problem is already solved for two-element lists, for example,

```
(append '(b c) '(d e f)) = (b c d e f)
```

How can we get from this solution to the solution of the three-element case? The result required is `(a b c d e f)`, which is given by

```
(cons 'a '(b c d e f))
```

The above can be expressed in terms of the length = 2 solution:

```
(cons 'a (append '(b c) '(d e f)) )
```

Notice, now, that 'a' is the `car` of the original list `(a b c)` and that `(b c)` is the `cdr` of this list. This leads us to the general form of the reduction: If *L* is a list of length $n > 0$, then

```
(append L M) = (cons (car L) (append (cdr L) M) )
```

Notice that `(cdr` *L*`)` is of length $n - 1$, so we have successfully reduced the length = n problem to the length = $n - 1$ problem. As we continue to reduce the length of this list, we are guaranteed to reach the null list eventually, which we know how to solve, so the process *must* terminate. It is now easy to see the program. To compute `(append` *L M*`)`, if *L* is null,

then return *M*; otherwise return the result of consing `(car` *L*`)` onto the result of `(append cdr` *L*`)` *M*`)`. In LISP this is

```
(defun append (L M)
  (if (null L)
      M
      (cons (car L) (append (cdr L) M)) ))
```

Notice that if we had decided to reduce *M* to the null list, it would have been much more difficult, because the LISP selector functions have been designed to work at the *beginnings* of lists.

■ ***Exercise 9-10*:*** Program the `append` function so that it reduces its second rather than its first argument to the null list. Accomplishing this will require you to define one or more auxiliary functions.

■ ***Exercise 9-11:*** Write a function `delprop` that removes a property and its value from a property list. For example,

```
DS
   (name (Don Smith) age 45 salary 30000
         hire-date (August 25 1980))
(delprop DS 'age)
   (name (Don Smith) salary 30000 hire-date (August 25 1980))
(delprop DS 'name)
   (age 45 salary 30000 hire-date (August 25 1980))
(delprop DS 'hire-date)
   (name (Don Smith) age 45 salary 30000)
```

■ ***Exercise 9-12:*** Write a function `remassoc` that removes an association from an association list. For example,

```
DS
   ((name (Don Smith)) (age 45) (salary 30000)
     (hire-date (August 25 1980)))
(remassoc 'salary DS)
   ((name (Don Smith)) (age 45) (hire-date (August 25 1980)))
```

■ ***Exercise 9-13:*** Write a function `addprop` that adds a property to a property list if it is not there or alters its value if it is. For example,

```
(addprop DS 'male 'sex)
   (name (Don Smith) age 45 salary 30000
         hire-date (August 25 1980) sex male)
(addprop DS 34500 'salary)
   (name (Don Smith) age 45 salary 34500
         hire-date (August 25 1980))
```

■ ***Exercise 9-14:*** Write a function analogous to `addprop` for *a*-lists.

Atoms Have Properties

LISP was originally developed for artificial intelligence applications. These applications often must deal with the properties of objects and the relationships among objects. Suppose we were writing a LISP program to manipulate European countries. Each country has a number of properties and each is related to a number of other countries. For example, each country has a name, a capital, a population, an area, and various countries on which it borders. How can these be handled in LISP?

In LISP, objects are represented by atoms, and each atom has an associated *p*-list that represents the properties of the atom and the relations in which it participates. We can represent some of the properties of England and France by Figure 9.4. The small circles represent objects (atoms), and the lines show their properties. Notice that the edges are labeled with the *indicators* of the properties and that the values of the properties are often themselves objects (i.e., atoms). How are atoms created and how are they given properties?

Atoms are created in LISP by simply mentioning them. For instance, if we type

```
(England France Spain Germany Portugal)
```

then the atoms `England`, `France`, and so on, will have been created. Each of these objects comes complete with a property, its *print name*, which is a character string tagged by `pname`[9] (see Figure 9.5). Every atom has a print name; it is the means by which we denote the atom and it is the way the atom is represented when it is displayed.

Suppose that we define `Europe` to be a list of the European countries:

```
(set 'Europe '(England France Spain Germany Portugal ...))
  (England France Spain Germany Portugal ...)
```

Now if we request that some atom be displayed, it is its print name that we will see. For example,

```
(car Europe)
  England
```

Similarly, whenever we write a name such as `England`, it will refer to that unique atom whose print name is `England`, that is, the LISP system will look for an atom whose `pname`

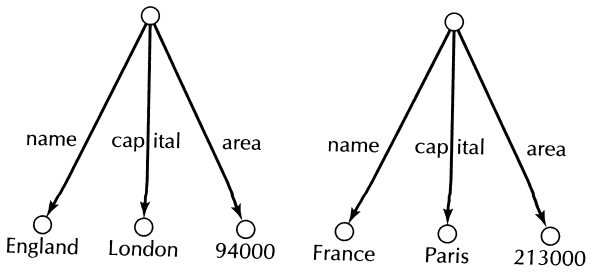


Figure 9.4 Examples of Objects and Their Properties

[9] This is the case in the LISP 1.5 system, which is the basis for all later LISP implementations. Other LISP systems may differ in implementation details. For example, several, including Common LISP, use "print-name cells" instead of the `pname` property.

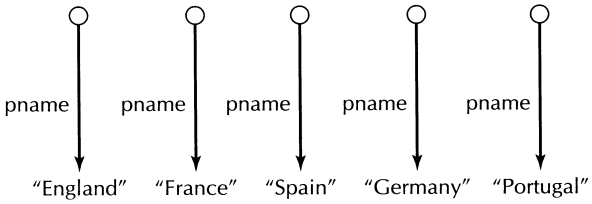


Figure 9.5 Examples of Objects and Their Print Names

is `"England"`. We can see this by using the `eq` function, which tells us if two atoms are the same:

```
(eq 'England (car Europe))
  t
(eq 'France (car Europe))
  nil
```

Thus, we can see that the print name of an atom is analogous to a proper name in English; it uniquely denotes an object.

We have seen that all atoms come with one property—their print name. How are other properties attached to an atom? Several procedures are provided by LISP for accessing the properties of an atom. For example, to define the capital of France as Paris, we can write

```
(putprop 'France 'Paris 'capital)
  Paris
```

which alters the properties of the object France as shown in Figure 9.6. We can find out the value of a property with the `get` function:

```
(get 'France 'capital)
     Paris
(get 'France 'pname)
     "France"
```

Notice that the `putprop` and `get` procedures are reminiscent of the `addprop` and `getprop` procedures we saw on pp. 324–330. This is not coincidental, as we will see when we discuss the implementation of atoms.

There are several other important properties that many objects have. One of these is the `apval` of an atom.[10] When an atom is used in a `set`, it is bound to some value. For example,

```
(set 'Europe '(England France ...))
```

binds the atom `Europe` to the list `(England France ...)`. The property of being bound to a value is denoted by the `apval` indicator (which stands for "applied value"). Af-

[10] Here again we refer to LISP 1.5. Some other LISP dialects use a "value cell" instead of the `apval` property.

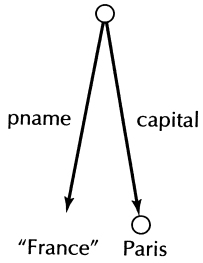


Figure 9.6 Example of Putting a New Property on an Object

ter the above `set` is executed, the atom `Europe` will have the properties shown in Figure 9.7. We can see this by typing

```
(get 'Europe 'apval)
   (England France Spain ...)
```

In fact, `set` is just an abbreviation for a particular application of `putprop`. Thus `(set 'Europe '(England France ...))` is exactly equivalent[11] to

```
(putprop 'Europe '(England France ...) 'apval)
```

There are several other built-in properties that we will mention briefly. Consider a function definition such as

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

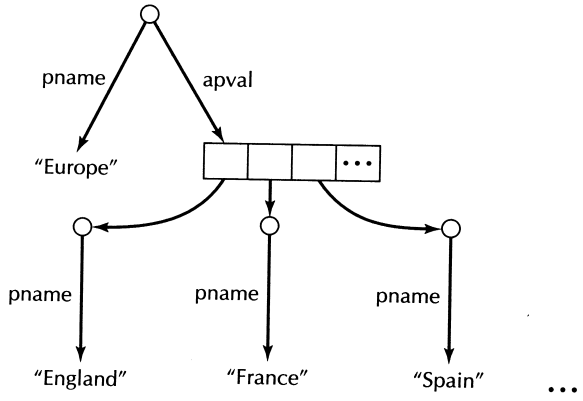


Figure 9.7 Example of the Applied Value of an Object

[11] Although, as noted before, some LISP implementations make a special case of `apval` for the sake of efficiency.

This is another example of binding a name to a value; in this case, the name `getprop` is bound to a function. Binding of atoms to functions is represented by the `expr` property.[12] After the above application of `defun` the value of the `expr` property of `getprop` can be found by

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

(The atom `lambda` indicates a function value, which is discussed in Chapter 10.) Thus, the above `defun` is equivalent to

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

There are a number of other properties used by the LISP system that indicate compiled functions, functions that do not evaluate their arguments, and so forth.

Some LISP systems allow the entire property list of an atom to be accessed.[13] For example,

```
(symbol-plist 'France)
   (pname "France" capital Paris)
(symbol-plist 'Europe)
   (pname "Europe" apval (England France Spain ...))
(symbol-plist 'getprop)
   (pname "getprop" expr (lambda (p x)
                                 (if (eq (car x) p)
                                     (cadr x)
                                     (getprop p (cddr x)) )) )
```

We can see that the property lists of atoms are just like the property lists we discussed previously.

Lists Have a Simple Representation

Recall that LISP developed out of a desire for an algebraic language for linked list processing. Therefore, although there are many ways that lists can be represented, it is not surpris-

[12] In LISP 1.5. Other LISP dialects use "function cells" instead of the `expr` property.

[13] The function to accomplish this is known variously as `symbol-plist`, `plist`, and `getproplist`.

ing to learn that LISP lists are usually implemented as linked structures. For instance, the list

```
(to be or not to be)
```

is represented as a sequence of six (not necessarily contiguous) cells in memory, each containing a pointer to the next. Each cell also contains a pointer to the element of the list it represents. This is shown in Figure 9.8. (We have labeled atoms with their print names; their exact representation is described later.) The last cell points to `nil`, representing the end of the list. We will call the two parts of a cell the *left* part and the *right* part. The final null pointer will often be drawn as a slash through the right half of the last cell.

Lists containing other lists are represented in the same way. For example, the list

```
( (to 2) (be 2) (or 1) (not 1) )
```

would be represented in storage as shown in Figure 9.9. Since the left part of a cell points to an element of a list, it can point either to an atom or a list (which is in turn represented by a cell). Since the right part of a cell normally points to the rest of the list, it will normally point to another cell or `nil`. The null list is simply a pointer to the atom `nil`.

- ***Exercise 9-15:*** Draw the list structures for both the *p*-list and *a*-list representations of the personnel record for Don Smith.

- ***Exercise 9-16:*** Since programs are themselves written as lists, programs can be represented as the same list structures as data. Draw the list structure corresponding to the definition of `make-table` in Figure 9.1, which is the first four lines of that figure.

- ***Exercise 9-17:*** Draw the list structure corresponding to the following expression:

```
(quotient (plus (minus B)
                (expt (difference (expt b 2)
                                  (times 4 a c))
                      0.5))
          (times 2 a))
```

  A structure such as you have drawn is often called an *expression tree.*

- ***Exercise 9-18*:*** Discuss LISP's strategy of dynamic typing and compare it with statically typed languages discussed in previous chapters. Discuss the advantages and disadvantages of each, paying particular attention to flexibility, security, and efficiency.

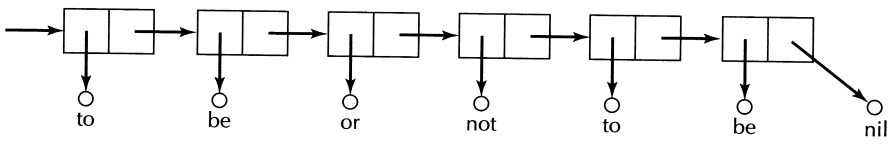


Figure 9.8 Representation of `(to be or not to be)`

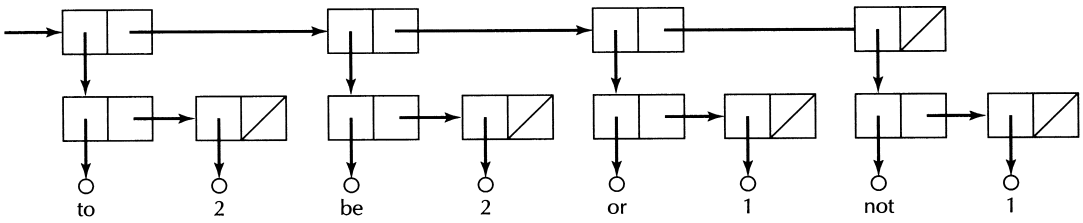


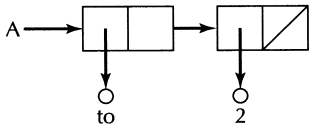
Figure 9.9 Representation of a List of Lists

The List Primitives Are Simple and Efficient

Let's consider the implementation of the list-processing functions `car`, `cdr`, and `cons`. Suppose we have a pointer, `L`, to the beginning of the list in Figure 9.8 and we want to get a pointer `A` to the `car` of this list. We can see that this pointer is in the left half of the cell pointed to by `L`. Therefore, in Pascal notation,

```
A := L↑.left;
```

In other words, follow the `L` pointer to a cell in memory and return its left half. This is an efficient operation; it works regardless of whether the first element of the list is an atom or another list. Consider the same operation on the list in Figure 9.9. The pointer `A` will be



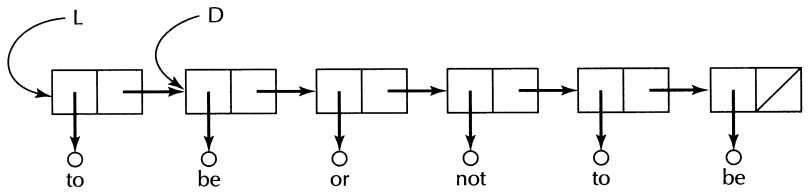
In other words,

```
(car '((to 2) (be 2) (or 1) (not 1)) ) = (to 2)
```

The `cdr` function is exactly analogous: Follow the pointer and extract the *right* half of the word:

```
D := L↑.right;
```

In the case of the list in Figure 9.8:



Notice that `D` actually points to a *sublist* of list `L`; we will discuss the implications of this on pp. 338–340. Also notice that repeated applications of `cdr` will "walk" from each element of the list to the next by following its right pointers.

John McCarthy has explained the origin of the cryptic names `car` and `cdr`. On the IBM 704 computer, each word had two fields large enough to hold a pointer; these were called the "address" field and the "decrement" field. Thus, if `L` were a pointer to a list, then `(car L)` would return the "Contents of the Address part of Register `L`" (memory locations were called "registers"). Similarly, `(cdr L)` meant "Contents of the Decrement part of Register `L`." Over the years many alternative names for these functions have been proposed, including `first/rest`, `head/tail`, `first/final`, and `Hd/Tl`. However, none of these has been able to supplant `car` and `cdr`. One reason may be that none of these other names is amenable to the construction of compound selectors (such as `caddar`).[14]

Next we will consider the implementation of the `cons` function. Since `cons` is the inverse of `car` and `cdr`, it is already clear what its effect must be; this is shown in Figure 9.10. The result of consing two lists pointed to by `A` and `D` must be a pointer `L` to a cell whose left half is `A` and whose right half is `D`. In other words, all we have to do is put `A` and `D` in the left and right halves of a cell. But which cell? Clearly, it is necessary for `L` to be a pointer to a cell that is not in use. Therefore, the `cons` operation requires a *storage allocation* step; a new cell must be allocated from a *heap* or *free storage area*. The mechanism required to do this will be discussed in Chapter 11, Section 11.2; for now we will assume that a procedure `new` is available that returns a pointer to a freshly allocated cell. This is the case in Pascal: `new(L)` stores into `L` a pointer to a new memory cell. Therefore, the steps required to do a `cons` are

```
new(L);
L↑.left  := A;
L↑.right := D;
```

With the possible exception of memory allocation, which we have not discussed, we can see that `cons` is also quite efficient.

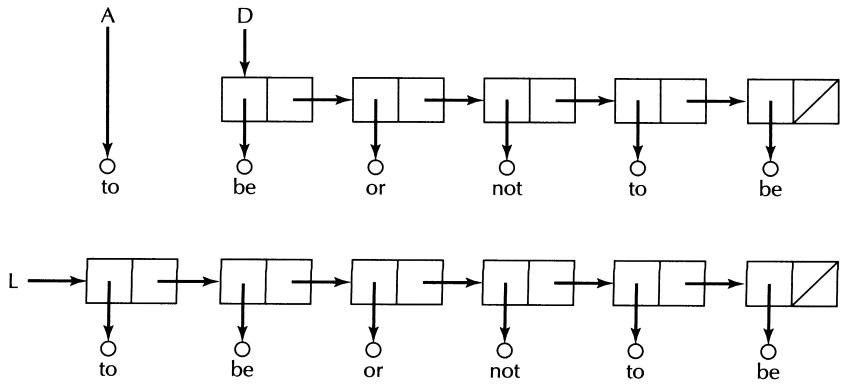


Figure 9.10 Implementation of Cons

[14] Common LISP does permit `first` and `rest` as synonyms for `car` and `cdr`.

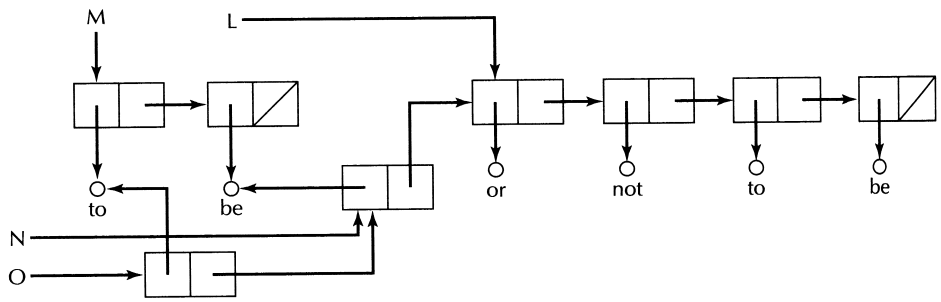
```
(set 'L '(or not to be))
  (or not to be)
(set 'M '(to be))
  (to be)
(car M)
  be
(set 'N (cons (cadr M) L))
  (be or not to be)
(car M)
  to
(set 'O (cons (car M) N))
  (to be or not to be)
(cons (car M) (cons (cadr M) L))
  (to be or not to be)
```

Figure 9.11 Illustration of Shared Sublists

Sublists Can Be Shared

Consider the LISP session in Figure 9.11.

The result of these operations is the following list structure:



Trace through each of the lists `M`, `L`, `N`, and `O` to be sure that you see that all the elements are there in the right order. We can see that a lot of the substructure of the lists is shared. This economizes storage. In this case, eight cons-cells are used rather than the 21 that would be required if each list were independent.

In previous chapters we have discussed the danger of *aliasing*, that is, of having more than one path to a memory location. The danger is that a variable can have its value changed without being directly assigned to because it shares its storage with a different variable that has been assigned. This makes programs less predictable and much harder to understand. We might be led to expect that the extensive sharing of sublists in LISP programs makes these exceptionally hard to understand, but this is not the case.

The reason is that aliasing, as well as sharing of data structures, is a problem only when combined with the ability to update data structures. Note that `car`, `cdr`, and `cons` are all *pure functions*, that is, they have no side effects on lists already created. As we have noted before, `cdr` does not *delete* an element from a list; the original list still exists with all of its

members. Similarly, `cons` does not *add* an element to a list; rather, it computes a new list with the correct elements. The result is that this sharing of sublists is *transparent;* it increases the efficiency of the program without changing its meaning. We will see below, however, that there are some circumstances in which the programmer can alter list structures. In these cases, it makes a difference as to whether lists are being shared, and there is a loss of transparency.

List Structures Can Be Modified

In the beginning of this chapter, we contrasted an *applicative* language, which is based on the application of pure functions to their arguments, with an *imperative* language, which is based on assignments to changeable memory locations. Although LISP is predominantly an applicative language, it does have a few imperative features. We have seen the pseudo-functions (or procedures) `set` and `defun`, which are used for binding names to objects. Now we will discuss two pseudo-functions for altering list structures.

LISP lists are very simple; they are constructed from a number of instances of identical components, cons-cells, all having exactly two parts, their left and right halves. Thus, if we want to alter a LISP list, there are really only two things we can do: alter the left half of a cell or alter the right half of a cell. The two pseudo-functions that LISP provides for this are called `rplaca` and `rplacd` (meaning "replace address part" and "replace decrement part").

The implementation of these pseudo-functions is simple—`(rplaca L A)` simply assigns the pointer `A` to the left part of the cell pointed to by `L`:

```
L↑.left := A;
```

Similarly, `(rplacd L D)` assigns the pointer `D` to the right part. This operation is illustrated in Figure 9.12. The dotted lines show the new pointer established by `rplacd`.

The interaction of sharing and assignment lead to all of the bad effects we have come to expect from aliasing. Consider this example:

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

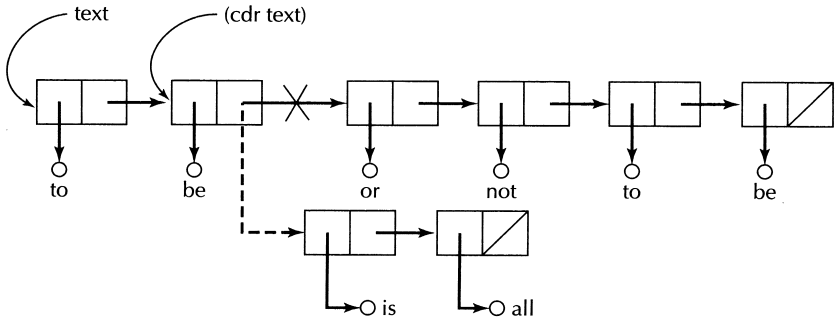


Figure 9.12 Execution of `(rplacd (cdr text) '(is all))`

```
(rplacd x '(is all))
  (be is all)
text
  (to be is all)
```

Notice that the value of `text` has been changed even though it was not even mentioned in the `rplacd` operation! In real programs (which are much larger than this example), a single `rplaca` or `rplacd` can change the values of seemingly unrelated lists throughout the program. It can even change the program since the program itself is represented as a list. Clearly, these operations are a trap for the unwary and must be used with caution!

- ***Exercise 9-19:*** The `rplaca` and `rplacd` pseudo-functions are usually used to increase the performance of LISP programs. For example, the `addprop` function we programmed in a previous exercise changes the value of a property by recopying the entire list up to and including the value to be changed. This could be quite inefficient if the value to be changed were near the end of a long list. Write a new `addprop` pseudo-function that uses `rplaca` and `rplacd` to alter an element of a property list without copying the rest of the list.

- ***Exercise 9-20:*** Write an analogous pseudo-function for *a*-lists.

- ***Exercise 9-21:*** Use the `symbol-plist` and `addprop` functions to define `putprop`.

- ***Exercise 9-22:*** Write a pseudo-function `remprop` that deletes a property's indicator and value from an atom's property list.

- ***Exercise 9-23:*** The following commands will cause many LISP systems to go into an infinite loop printing `A`'s and `B`'s:

```
(set 'x '(A B))
  (A B)
(rplacd (cdr x) x)
  (B A B A B A B A B A B A B A B ....
```

Explain why.

Atoms Are Just Pointers to Their Property Lists

We have seen that the cons-cells from which lists are constructed have two parts, each of which can point to either an atom or a list. In the case of a list, the cons-cell points to another cons-cell, namely, the one containing the `car` and `cdr` of the element list. What does the cons-cell point to when it points to an atom? In other words, how are atoms represented in memory?

To answer this question, we can begin by asking what it is that makes one atom different from another. In other words, what constitutes the *identity* of an atom? One obvious answer is the print name; two atoms are different if they have different print names, and they are the same if they have the same print name. This is not the complete solution, however. Recall that atoms, like the real-world objects they are often used to model, have many prop-

erties. In the real world, we distinguish different objects by their properties: Two objects are different if they differ in at least one property, and they are the same (i.e., indistinguishable) if they agree in all of their properties. Of course, many different objects agree in *some* of their properties; for example, they might have the same name or the same shape. Computers, however, are finite: It is not possible to model the unlimited number of properties that characterize real-world objects. Rather, a finite subset of these is selected that is relevant to the situation being modeled. Thus, we might find that two distinct atoms, modeling two distinct objects, have exactly the same properties. We know that they are distinct atoms because they participate in different list structures. Since list structures really are just structures of pointers, we find that two atoms are the same if they are represented by the same pointer, and they are different if they are represented by different pointers. In implementation terms, an atom is equivalent to its location in memory; any structure pointing at that location is pointing at the same atom, and any structure pointing at a different location is pointing at a different atom. Hence, the small circles representing atoms in Figures 9.8 and 9.9 are really memory locations.

What is stored in the memory location representing an atom? There need not be anything since we are really just using the *address* of the location as a tag that uniquely identifies the atom. In fact, it would even be possible to use *illegal addresses*, which do not represent a location in memory at all, to stand for atoms. Some LISP systems make use of these memory locations by using them to hold a pointer to the atom's property list. For example, the original LISP system used regular cons-cells to represent atoms. A special value that was not a legal address (say, -1) was placed in the left field to indicate that the cell represented an atom rather than a regular list. The right field pointed to the property list of the atom. Figure 9.13 gives an example of this.

You will see in the following exercises that the operations on atoms are simple and efficient to implement.

- ***Exercise 9-24:*** Write a Pascal-like expression for determining whether a pointer to a cons-cell is a pointer to a list or a pointer to an atom. This is, in essence, the `atom` predicate.
- ***Exercise 9-25:*** Write a Pascal-like expression to access the property list of the atom represented by the pointer `A`. This is, in essence, the `symbol-plist` function.
- ***Exercise 9-26:*** How is the `eq` predicate implemented?

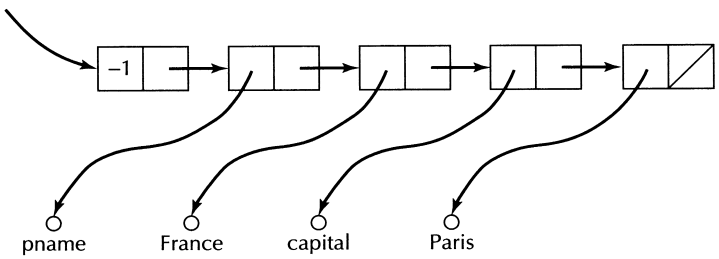


Figure 9.13 Representation of the Atom `France`

EXERCISES

1. `Car`, `Cdr`, and `cons` all work on the *first* elements of lists. Define the meaning and describe the implementation of the analogous functions `last`, `butlast`, and `suffix` that work on the last elements of lists. Define `append` and `getprop` using these functions.

2. Suppose that arrays are to be represented by linked lists. Write a function `(elt` *A n*`)` that returns the *n*th element of a list *A* and an "indexed substitution" function `(indsubst` *x n A*`)` that returns a list like *A* except that the *n*th element has been replaced by *x*.

3. Write a LISP function to convert a property list into the corresponding association list. Write a LISP function to convert an association list into the corresponding property list.

4. Write a LISP function to reverse a list.

5.** Write the list primitives (`car`, `cdr`, `cons`, `eq`, `atom`, `null`) in a conventional programming language such as Pascal or Ada. Write a `readlist` procedure that reads a list in the *S*-expression notation and constructs the corresponding list structure. Write a `printlist` procedure that takes a list structure and prints it in the *S*-expression notation. Can you think of a simple way of indenting the list so that it is readable?

6.** It has been observed that many lists occupy contiguous memory locations. Therefore, the right pointers in most of the cells are redundant, since they just point to the following memory location. One alternative representation for lists is called *cdr-encoding*, in which there are two different kinds of cells. One has left and right halves, as described in this chapter; the other just has a left field, since the right field is assumed to point to the following cell. This halves the storage required for many lists. Develop the method of cdr-encoding: Design the formats for list cells; describe the algorithms for performing `car`, `cdr`, and `cons`; describe what you will do about shared sublists; and estimate the differences in space and time consumed by cdr-encoding and the usual representation.

7.** Design a completely different representation for LISP lists. For example, represent all lists in consecutive memory locations in exactly the order in which they are written on paper, including open and close parentheses. Analyze your new list representation, including space and time performance estimates and other advantages and disadvantages.

8. Consider the following LISP commands:

```
(defun message (x) (cons x '(is dull)) )
(message 'That)
(rplacd (cdr (message 'Nothing)) '(a surprise))
(message 'This)
```

The first call of `message` returns `(That is dull)`; the second call of `message` (vice `rplacd`) returns `(Nothing is dull)`. What does the third call return? What conclusions can you draw from this example?

9*. Write a LISP function that determines whether a list is circular. To accomplish this you need to know that `(eq` *x y*`)` compares the *pointers* to *x* and to *y*.