

CHAPTER 3

LISTS

In Chapter 2, we discussed one type of LISP object—numbers and subtypes of numbers. In this chapter, we will begin discussing the most important type of Lisp object, the *list*—*what* LISP was named for. We will start discussing the evaluation of lists in the next chapter. In this chapter, we will discuss the printed representation of lists; that is, list S-expressions. We can define the list S-expression as follows:

A left parenthesis followed by zero or more S-expressions followed by a right parenthesis is a list S-expression.

According to this definition, (1 2 3 . 3 4) is a list S-expression, since 1, 2, 3 . 3, and 4 are S-expressions (denoting numbers). Also () is a list S-expression, since it is a left parenthesis followed by zero S-expressions followed by a right parenthesis. We refer to the (zero or more) S-expressions in a list as *elements* or *members* of the list.¹ So, the first list has four members, which are 1, 2, 3 . 3, and 4, and the second list has no members (we call it *the empty list*). Since a list S-expression is itself an S-expression, a list can be a member of a list. For example, (1 (2 3 . 3) 4) is a list with three members, the second of which is the list (2 3 . 3). Notice that the parentheses are part of the list; they are

¹ In this chapter, we will say “list” instead of “list S-expression” and say “list object” when we mean the object.

not merely grouping brackets as they are in algebra. For example, if you remove the inner set of parentheses from the three-member list `(1 (2 3.3) 4)`, you get the four-member list `(1 2 3.3 4)`, and the list `((1 2 3.3 4))` is different yet, because it is a list with one member, which is a list. Even `()` and `(())` are different lists. The first is the empty list; the second is a list with one member, which happens to be the empty list.

If we typed a list to a Lisp listener, it would read it, construct the list object it represents, and try to evaluate the list object. As I said above, we will discuss the evaluation of list objects in the next chapter, so at this stage, we are only interested in having Lisp print the list back to us. You can prevent Lisp from evaluating an object by *quoting* the S-expression you enter. You quote an S-expression by typing a *quote mark* in front of it. The quote mark is the single quote mark on your keyboard that you might use for an apostrophe. In this text, it will look like this: `'`. Notice that there is another single quote mark on your keyboard that points in the other direction. It is called a *back quote* and will look in this text like this: ```. If you type a quoted list to Lisp, it will type the list back to you:

```
> '(1 2 3.3 4)
(1 2 3.3 4)
> '(1 (2 3.3) 4)
(1 (2 3.3) 4)
> '((1 2 3.3 4))
((1 2 3.3 4))
```

Actually, the Lisp listener is still going through its normal read-eval-print cycle, but when it reads a quoted S-expression, it constructs a quoted object, and *the value of a quoted object is the object itself*, so then it prints the list you typed in, using a printed representation it chooses. We will discuss quoted objects more completely in Chapter 9. The printed representation Lisp chooses for a list is not always the same as the one you use. For example, Lisp has its own peculiar way of printing the empty list:

```
> '()
NIL
> '(() )
(NIL)
```

We will discuss `NIL` more in Chapter 6. For now, just think of it as a possible printed representation of the empty list.

The other way that LISP's printed representation of lists may differ from yours is in matters of spacing and line breaking. Anywhere within a list that one blank makes sense, you may have several blanks and even a carriage return, and blanks are optional on either side of a parenthesis. You may type spacing to please your own eye; LISP may choose different spacing. For example:

```
> '(1(2 3.3)
   ( 4 )5)
(1 (2 3.3) (4) 5)
```

It is important to remember that a list is one `S-expression` regardless of how many members it has. So the LISP listener will read one top-level list at a time. That is, after printing a value or upon initial entry to LISP, LISP prints a prompt. You now type a left parenthesis, perhaps preceded by a quote mark. You are typing a top-level list until the number of right parentheses you type equals the number of left parentheses you have typed. Your list may extend over several lines. Some LISPs will type a prompt at the beginning of each line. Others won't. When you finally type a right parenthesis to match that first left parenthesis and then type a carriage return, LISP will type the value of the list object whose printed representation you entered.

Miscounting parentheses can lead to a common, but very frustrating, experience. You have typed in too few right parentheses. You think you have entered an entire list and hit the carriage return. LISP, however, just types a prompt (or doesn't even do that), and you both just sit there staring at each other. LISP is waiting for you to finish your list. If you are too confused to finish it properly, it often works to just type several right parentheses—more than enough to do the job—and then the final carriage return. Some LISPs don't require you to type a carriage return after a list. They recognize when one is finished, automatically go to the next line, and output the value. These LISPs avoid the confusion discussed in this paragraph. Many modern LISP development environments also make typing lists easier by blinking the cursor on the matching left parenthesis just after you type each right parenthesis. If you are using such an environment, it pays to watch that cursor.

An easy way to count parentheses is to count 1 at the first left parenthesis, increase the count by 1 at each subsequent left parenthesis, and decrease the count by 1 at each subsequent right parenthesis. When you reach zero again, you are at the right parenthesis that matches the first left parenthesis and your list is finished. The list below has the appropriate numbers written below each parenthesis.

```
(1 ( ) (2(3) 4) 5 ( ( (6) ) 7) )
1 2 1 2 3 2 1 2 3 4 3 2 1 0
```

Our hierarchy of COMMON LISP types is now

```
list
number
  integer
  float
    short-float
    single-float
    double-float
    long-float
```

Exercises

- 3.1 (*r*) Try all the interactions of this chapter for yourself.
- 3.2 (*i*) Enter some short unquoted lists and note the error messages. We will discuss these errors later.
- 3.3 (*i*) Type a line containing just a right parenthesis. How does LISP respond?
- 3.4 (*i*) Enter a quoted list with too many right parentheses. How does your LISP respond?
- 3.5 (*i*) Enter some quoted lists that, by including carriage returns, extend over several lines. Carefully observe how LISP behaves until the list is finally finished. Note the format in which LISP prints lists.

- 3.6 (i) Start entering a list that, by including carriage returns, extends over several lines, but stop before finishing the list. If you now decide that you don't want to enter this list after all, how do you erase it? Your character erase key may erase even into previous lines. Try that. If you are using some kind of LISP Machine, you may have a CLEAR INPUT key. Press it. Otherwise, try pressing the interrupt character to get into the debugger (see Exercise 1.6); then get back to top-level LISP (see Exercise 1.8). Write your delete-current-list sequence here:_____ You should now be able to delete the last character, delete the current line, or delete the current list.
- 3.7 (d) Experiment with lists with different spacing. Try spaces between the quote mark and the first open parenthesis. In each case, compare LISP's printed representation with yours.