6

List Data Structures

6.1 INTRODUCTION

This chapter presents more list-manipulation functions, and shows how lists are used to implement such other data structures as sets, tables, and trees. Common Lisp offers many built-in functions that support these data structures. This is one of the strengths of Lisp compared to other languages. A Lisp programmer can immediately concentrate on the problem he or she wants to solve. A Pascal or C programmer faced with the same problem must first go off and implement parts of a Lisp-like system, such as linked list primitives, symbolic data structures, a storage allocator, and so on, before getting to work on the real problem.

The approach we take to lists in this chapter is somewhat more sophisticated than in Chapter 2. We will discuss not only what various Lisp primitive functions do, but also how they work inside. In preparation for this, you may want to review the discussion of dotted-pair notation in section 2.17. If you haven't been reading the Advanced Topics sections, that's okay; just go back and read section 2.17 now.

6.2 PARENTHESIS NOTATION VS. CONS CELL NOTATION

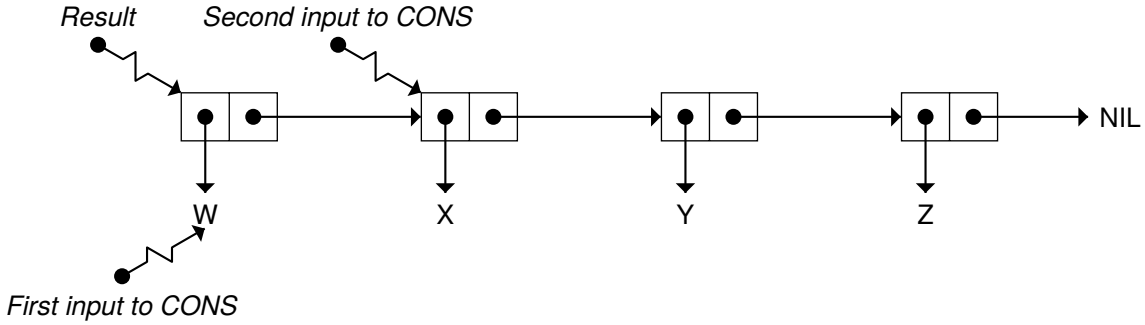
Writing lists in parenthesis notation is convenient, but it can be misleading. Lists in parenthesis notation appear symmetric: They begin with a left parenthesis and they end with a right one. One might therefore expect the CONS function to treat its arguments symmetrically. If CONS can add a symbol to the front of a list like so:

```
(cons 'w '(x y z))  ⇒  (w x y z)
```

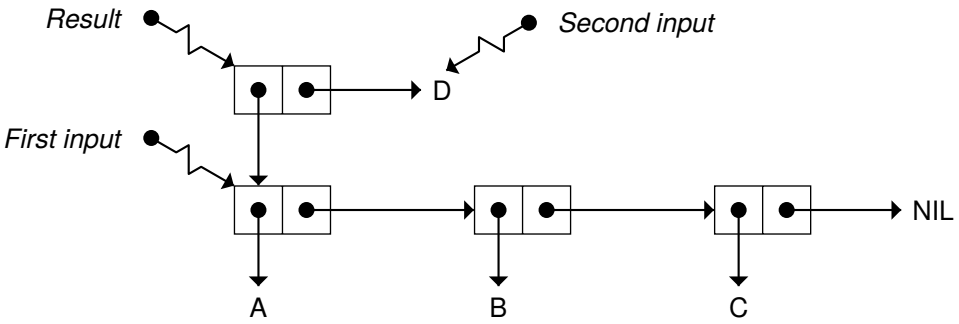
why can't it add a symbol to the end of a list? Beginners who try this are surprised by the result:

```
(cons '(a b c) 'd)  ⇒  ((a b c) . d)
```

There is no reason to view the left end of a list as fundamentally different from the right end if we stick to parenthesis notation. But switching to cons cell notation reveals the crucial difference: Lists are one-way chains of pointers. It is easy to add an element to the *front* of a list because what we're really doing is creating a new cons cell whose cdr points to the existing list. If the inputs to CONS are W and (X Y Z), the result will be a new cell whose car points to W and whose cdr points to the old chain (X Y Z), as shown below. Although we usually display the result as (W X Y Z), we can also write it in dot notation as (W . (X Y Z)).



When we cons (A B C) onto D, it's the car of the new cell that points to the old list (A B C); the cdr points to the symbol D. The result is normally written ((A B C) . D), which looks decidedly odd in parenthesis notation. The dot is necessary because the cons cell chain ends in an atom other than NIL. In cons cell notation the structure looks like this:



There is no direct way to add an element to the end of a list simply by creating a new cons cell, because the end of the original list already points to NIL. More sophisticated techniques must be used. One of these is demonstrated in the next section.

6.3 THE APPEND FUNCTION

APPEND takes two lists as input; it returns a list containing all the elements of the first list followed by all the elements of the second.*

```
> (append '(friends romans) '(and countrymen))
(FRIENDS ROMANS AND COUNTRYMEN)

> (append '(l m n o) '(p q r))
(L M N O P Q R)
```

If one of the inputs to APPEND is the empty list, the result will be equal to the other input. Appending NIL to a list is like adding zero to a number.

```
> (append '(april showers) nil)
(APRIL SHOWERS)

> (append nil '(bring may flowers))
(BRING MAY FLOWERS)

> (append nil nil)
NIL
```

*Note to instructors: To simplify the upcoming discussion of how APPEND works, we consider only the two-input case. In Common Lisp, APPEND can accept any number of inputs.

APPEND works on nested lists too. It only looks at the top level of each cons cell chain, so it doesn't notice if a list is nested or not.

```
> (append '((a 1) (b 2)) '((c 3) (d 4)))
((A 1) (B 2) (C 3) (D 4))
```

APPEND does not change the value of any variable or modify any existing cons cells. For this reason, it is called a **nondestructive** function.

```
> (setf who '(only the good))
(ONLY THE GOOD)

> (append who '(die young))
(ONLY THE GOOD DIE YOUNG)

> who
(ONLY THE GOOD)            The value of WHO is unchanged.
```

APPEND may appear to treat its two inputs symmetrically, but this is just an illusion caused by the use of parenthesis notation. APPEND treats its two inputs quite differently. When we append the list (A B C) to the list (D E), APPEND *copies* the first input but not the second. It makes the cdr of the last cell of the copy point to the second input, and returns a pointer to the copy, as shown in Figure 6-1.

This description of how APPEND really works also explains why it is an error for the first input to APPEND to be a non-list, but it's okay if the second input is a non-list.

```
(append 'a '(b c d))   ⇒   Error! A is not a list.

(append '(w x y) 'z)   ⇒   (W X Y . Z)
```

APPEND wants to copy the cons cells that make up its first input. It can't when the first input is A because that isn't a list, so it signals an error. But when we append (W X Y) to Z, APPEND *can* copy its first input and make the cdr of the last cell point to the second input, so it doesn't have to signal an error. In this case the second input is Z rather than a list, so the result looks odd because the cons cell chain doesn't end in NIL.

Let us now return to the problem of adding an element to the end of a list. If we first make a list of the element, we can solve this problem by using APPEND.

```
(append '(a b c) '(d))   ⇒   (A B C D)
```

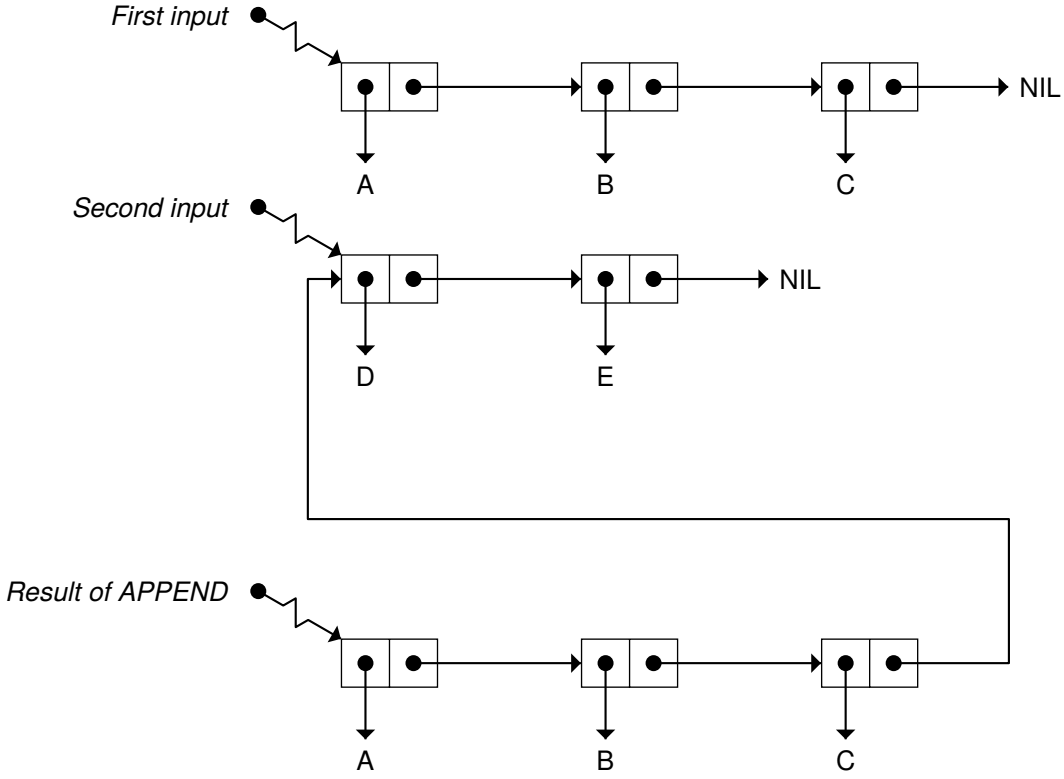


Figure 6-1 Result of appending (A B C) to (D E).

```
(defun add-to-end (x e)
  "Adds element E to the end of list X."
  (append x (list e)))

(add-to-end '(a b c) 'd)  ⇒  (A B C D)
```

6.4 COMPARING CONS, LIST, AND APPEND

Beginninng Lispers often have trouble distinguishing among CONS, LIST, and APPEND, since all three functions are used to build list structures. Here is a brief review of what each function does and when it should be used:

- CONS creates one new cons cell. It is often used to add an element to the front of a list.
- LIST makes new lists by accepting an arbitrary number of inputs and building a chain of cons cells ending in NIL. The car of each cell points to the corresponding input.
- APPEND appends lists together by copying its first input and making the cdr of the last cell of the copy point to the second input. It is an error for the first input to APPEND to be a non-list.

Now let's try some examples for comparison. First, consider the case where the first input is a symbol and the second input a list:

```
> (cons 'rice '(and beans))
(RICE AND BEANS)

> (list 'rice '(and beans))
(RICE (AND BEANS))

> (append 'rice '(and beans))
Error: RICE is not a list.
```

Next, let's see what happens when both inputs are lists:

```
> (cons '(here today) '(gone tomorrow))
((HERE TODAY) GONE TOMORROW)

> (list '(here today) '(gone tomorrow))
((HERE TODAY) (GONE TOMORROW))

> (append '(here today) '(gone tomorrow))
(HERE TODAY GONE TOMORROW)
```

Finally, let's try making the first input a list and the second input a symbol. This is the trickiest case to understand; you must think in terms of cons cells rather than parentheses and dots.

```
> (cons '(eat at) 'joes)
((EAT AT) . JOES)

> (list '(eat at) 'joes)
((EAT AT) JOES)

> (append '(eat at) 'joes)
(EAT AT . JOES)
```

To further develop your intuitions about CONS, LIST, and APPEND, try the above examples using the SDRAW tool described in the Lisp Toolkit section of this chapter. SDRAW draws cons cell diagrams.

6.5 MORE FUNCTIONS ON LISTS

Lisp provides many simple functions for operating on lists. We've already discussed CONS, LIST, APPEND, and LENGTH. Now we will cover REVERSE, NTH, NTHCDR, LAST, and REMOVE. Some of these functions must copy their first input, while others don't have to. See if you can figure out the reason for this.

6.5.1 REVERSE

REVERSE returns the reversal of a list.

```
> (reverse '(one two three four five))
(FIVE FOUR THREE TWO ONE)

> (reverse '(l i v e))
(E V I L)

> (reverse 'live)
Error: Wrong type input.

> (reverse '((my oversight)
             (your blunder)
             (his negligence)))
((HIS NEGLIGENCE) (YOUR BLUNDER) (MY OVERSIGHT))
```

Notice that REVERSE reverses only the *top level* of a list. It does not reverse the individual elements of a list of lists. Another point about REVERSE is that it doesn't work on symbols. REVERSE of the list (L I V E) gives the list (E V I L), but REVERSE of the symbol LIVE gives a wrong-type input error.

Like APPEND, REVERSE is nondestructive. It copies its input rather than modifying it.

```
> (setf vow '(to have and to hold))
(TO HAVE AND TO HOLD)

> (reverse vow)
(HOLD TO AND HAVE TO)

> vow
(TO HAVE AND TO HOLD)
```

We can use REVERSE to add an element to the end of a list, as follows. Suppose we want to add D to the end of the list (A B C). The reverse of (A B C) is (C B A). If we cons D onto that we get (D C B A). Then, reversing the result of CONS gives (A B C D).

```
(defun add-to-end (x y)
    (reverse (cons y (reverse x))))

(add-to-end '(a b c) 'd)  ⇒  (a b c d)
```

Now you know two ways to add an element to the end of a list. The APPEND solution is considered better style than the double REVERSE solution because the latter makes two copies of the list. APPEND is more efficient. Efficiency issues are further discussed in an Advanced Topics section at the end of this chapter.

6.5.2 NTH and NTHCDR

The NTHCDR function returns the *n*th successive cdr of a list. Of course, if we take zero cdrs we are left with the list itself. If we take one too many cdrs, we end up with the atom that terminates the cons cell chain, which usually is NIL.

```
(nthcdr 0 '(a b c))  ⇒  (a b c)

(nthcdr 1 '(a b c))  ⇒  (b c)
```

```
(nthcdr 2 '(a b c))  ⇒  (c)

(nthcdr 3 '(a b c))  ⇒  nil
```

Using inputs greater than 3 does not cause an error; we simply get the same result as for 3. This is one of the consequences of making the cdr of NIL be NIL.

```
(nthcdr 4 '(a b c))  ⇒  nil

(nthcdr 5 '(a b c))  ⇒  nil
```

However, if the list ends in an atom other than NIL, going too far with NTHCDR will cause an error.

```
(nthcdr 2 '(a b c . d))  ⇒  (c . d)

(nthcdr 3 '(a b c . d))  ⇒  d

(nthcdr 4 '(a b c . d))  ⇒  Error! D is not a list.
```

The NTH function takes the CAR of the NTHCDR of a list.

```
(defun nth (n x)
  "Returns the Nth element of the list X,
   counting from 0."
  (car (nthcdr n x)))
```

Since (NTHCDR 0 *x*) is the list *x*, (NTH 0 *x*) is the first element. Therefore, (NTH 1 *x*) is the second, and so on.

```
(nth 0 '(a b c))  ⇒  a

(nth 1 '(a b c))  ⇒  b

(nth 2 '(a b c))  ⇒  c

(nth 3 '(a b c))  ⇒  nil
```

The convention of numbering things from zero rather than from one is used throughout Common Lisp. You will encounter it again when we discuss arrays in Chapter 13.

EXERCISES

6.1. Why is (NTH 4 '(A B C)) equal to NIL?

6.2. What is the value of (NTH 3 '(A B C . D)), and why?

6.5.3 LAST

LAST returns the last cons cell of a list, in other words, the cell whose car is the list's last element. By definition, the cdr of this cell is an atom; otherwise it wouldn't be the last cell of the list. If the list is empty, LAST just returns NIL.

```
(last '(all is forgiven))  ⇒  (forgiven)

(last nil)  ⇒  nil

(last '(a b c . d))  ⇒  (c . d)

(last 'nevermore)  ⇒  Error! NEVERMORE is not a list.
```

EXERCISES

6.3. What is the value of (LAST '(ROSEBUD)) ?

6.4. What is the value of (LAST '((A B C))), and why?

6.5.4 REMOVE

REMOVE removes an item from a list. Normally it removes all occurrences of the item, although there are ways to tell it to remove only some (see the Advanced Topics section). The result returned by REMOVE is a new list, without the deleted items.

```
(remove 'a '(b a n a n a))  ⇒  (b n n)

(remove 1 '(3 1 4 1 5 9))  ⇒  (3 4 5 9)
```

REMOVE is a nondestructive function. It does not change any variables or cons cells when removing elements from a list. REMOVE builds its result out of fresh cons cells by copying (parts of) the list.

```
> (setf spell '(a b r a c a d a b r a))
(A B R A C A D A B R A)

> (remove 'a spell)
(B R C D B R)

> spell
(A B R A C A D A B R A)
```

The following table should help you remember which functions copy their input and which do not. APPEND, REVERSE, and REMOVE return a new cons cell chain that is not contained in their input, so they must copy their input to produce the new chain. Functions such as NTHCDR, NTH, and LAST return a pointer to some component of their input. They do not need to copy anything because, by definition, the exact object they want to return already exists.

Function	Copies its input?
APPEND	yes *(execept for the last input)*
REVERSE	yes
NTHCDR	no
NTH	no
LAST	no
REMOVE	yes *(only the second input)*

EXERCISES

6.5. Write an expression to set the global variable LINE to the list (ROSES ARE RED). Then write down what each of the following expressions evaluates to:

```
(reverse line)

(first (last line))

(nth 1 line)

(reverse (reverse line))

(append line (list (first line)))

(append (last line) line)

(list (first line) (last line))

(cons (last line) line)

(remove 'are line)

(append line '(violets are blue))
```

6.6. Use the LAST function to write a function called LAST-ELEMENT that returns the last element of a list instead of the last cons cell. Write

another version of LAST-ELEMENT using REVERSE instead of LAST. Write another version using NTH and LENGTH.

6.7. Use REVERSE to write a NEXT-TO-LAST function that returns the next-to-last element of a list. Write another version using NTH.

6.8. Write a function MY-BUTLAST that returns a list with the last element removed. (MY-BUTLAST '(ROSES ARE RED)) should return the list (ROSES ARE). (MY-BUTLAST '(G A G A)) should return (G A G).

6.9. What primitive function does the following reduce to?

```
(defun mystery (x) (first (last (reverse x))))
```

6.10. A palindrome is a sequence that reads the same forwards and backwards. The list (A B C D C B A) is a palindrome; (A B C A B C) is not. Write a function PALINDROMEP that returns T if its input is a palindrome.

6.11. Write a function MAKE-PALINDROME that makes a palindrome out of a list, for example, given (YOU AND ME) as input it should return (YOU AND ME ME AND YOU).

6.6 LISTS AS SETS

A set is an unordered collection of items. Each item appears only once in the set. Some typical sets are the set of days of the week, the set of integers (an infinite set), and the set of people in Hackensack, New Jersey, who had spaghetti for dinner last night.

Sets are undoubtedly one of the more useful data structures one can build from lists. The basic set operations are testing if an item is a **member** of a set; taking the **union**, **intersection**, and **set difference** (also called set subtraction) of two sets; and testing if one set is a **subset** of another. The Lisp functions for all these operations are described in the following subsections.

6.6.1 MEMBER

The MEMBER predicate checks whether an item is a member of a list. If the item is found in the list, the sublist beginning with that item is returned. Otherwise NIL is returned. MEMBER never returns T, but by tradition it is counted as a predicate because the value it returns is non-NIL (hence true) if and only if the item is in the list.