8 Recursion

8.1 INTRODUCTION

Because some instructors prefer to teach recursion as the first major control structure, this chapter and the preceding one may be taught in either order. They are independent.

Recursion is one of the most fundamental and beautiful ideas in computer science. A function is said to be ‘‘recursive’’ if it calls itself. Recursive control structure is the main topic of this chapter, but we will also take a look at recursive data structures in the Advanced Topics section. The insight necessary to recognize the recursive nature of many problems takes a bit of practice to develop, but once you ‘‘get it,’’ you’ll be amazed at the interesting things you can do with just a three- or four-line recursive function.

We will use a combination of three techniques to illustrate what recursion is all about: dragon stories, program traces, and recursion templates. Dragon stories are the most controversial technique: Students enjoy them and find them helpful, but computer science professors aren’t always as appreciative. If you don’t like dragons, you may skip Sections 8.2, 8.4, 8.6, and 8.9. The intervening sections will still make sense; they just won’t be as much fun.

8.2 MARTIN AND THE DRAGON

In ancient times, before computers were invented, alchemists studied the mystical properties of numbers. Lacking computers, they had to rely on dragons to do their work for them. The dragons were clever beasts, but also lazy and bad-tempered. The worst ones would sometimes burn their keeper to a crisp with a single fiery belch. But most dragons were merely uncooperative, as violence required too much energy. This is the story of how Martin, an alchemist's apprentice, discovered recursion by outsmarting a lazy dragon.

One day the alchemist gave Martin a list of numbers and sent him down to the dungeon to ask the dragon if any were odd. Martin had never been to the dungeon before. He took a candle down with him, and in the furthest, darkest corner found an old dragon, none too friendly looking. Timidly, he stepped forward. He did not want to be burnt to a crisp.

''What do *you* want?'' grumped the dragon as it eyed Martin suspiciously.

''Please, dragon, I have a list of numbers, and I need to know if any of them are odd'' Martin began. ''Here it is.'' He wrote the list in the dirt with his finger:

```
(3142 5798 6550 8914)
```

The dragon was in a disagreeable mood that day. Being a dragon, it always was. ''Sorry, boy'' the dragon said. ''I might be willing to tell you if the *first* number in that list is odd, but that's the best I could possibly do. Anything else would be too complicated; probably not worth my trouble.''

''But I need to know if *any* number in the list is odd, not just the first number'' Martin explained.

''Too bad for you!'' the dragon said. ''I'm only going to look at the first number of the list. But I'll look at as many lists as you like if you give them to me one at a time.''

Martin thought for a while. There had to be a way around the dragon's orneriness. ''How about this first list then?'' he asked, pointing to the one he had drawn on the ground:

```
(3142 5798 6550 8914)
```

''The first number in that list is not odd,'' said the dragon.

Martin then covered the first part of the list with his hand and drew a new left parenthesis, leaving

```
    (5798 6550 8914)
```

and said ‘‘How about this list?’’

‘‘The first number in that list is not odd,’’ the dragon replied.

Martin covered some more of the list. ‘‘How about this list then?’’

```
         (6550 8914)
```

‘‘The first number in that list isn’t odd either,’’ said the dragon. It sounded bored, but at least it was cooperating.

‘‘And this one?’’ asked Martin.

```
              (8914)
```

‘‘Not odd.’’

‘‘And this one?’’

```
                  ()
```

‘‘That’s the empty list!’’ the dragon snorted. ‘‘There can’t be an odd number in there, because there’s *nothing* in there.’’

‘‘Well,’’ said Martin, ‘‘I now know that not one of the numbers in the list the alchemist gave me is odd. They’re *all* even.’’

‘‘I NEVER said that!!!’’ bellowed the dragon. Martin smelled smoke. ‘‘I only told you about the *first* number in each list you showed me.’’

‘‘That’s true, Dragon. Shall I write down all of the lists you looked at?’’

‘‘If you wish,’’ the dragon replied. Martin wrote in the dirt:

```
(3142 5798 6550 8914)
     (5798 6550 8914)
          (6550 8914)
               (8914)
                   ()
```

‘‘Don’t you see?’’ Martin asked. ‘‘By telling me that the first element of each of those lists wasn’t odd, you told me that *none* of the elements in my original list was odd.’’

‘‘That’s pretty tricky,’’ the dragon said testily. ‘‘It looks liked you’ve discovered recursion. But don’t ask me what that means—you’ll have to figure it out for yourself.’’ And with that it closed its eyes and refused to utter another word.

8.3 A FUNCTION TO SEARCH FOR ODD NUMBERS

Here is a recursive function ANYODDP that returns T if any element of a list of numbers is odd. It returns NIL if none of them are.

```
(defun anyoddp (x)
  (cond ((null x) nil)
        ((oddp (first x)) t)
        (t (anyoddp (rest x)))))
```

If the list of numbers is empty, ANYODDP should return NIL, since as the dragon noted, there can't be an odd number in a list that contains nothing. If the list is not empty, we go to the second COND clause and test the first element. If the first element is odd, there is no need to look any further; ANYODDP can stop and return T. When the first element is even, ANYODDP must call itself on the rest of the list to keep looking for odd elements. That is the recursive part of the definition.

To see better how ANYODDP works, we can use DTRACE to announce every call to the function and every return value. (The DTRACE tool used here was introduced in the Lisp Toolkit section of Chapter 7. If your Lisp doesn't have DTRACE, use TRACE instead.)

```
(defun anyoddp (x)
  (cond ((null x) nil)
        ((oddp (first x)) t)
        (t (anyoddp (rest x)))))

(dtrace anyoddp)
```

We'll start with the simplest cases: an empty list, and a list with one odd number.

```
> (anyoddp nil)
----Enter ANYODDP
|     X = NIL
 \--ANYODDP returned NIL     First COND clause returns NIL.
NIL

> (anyoddp '(7))
----Enter ANYODDP
|     X = (7)
 \--ANYODDP returned T       Second COND clause returns T.
T
```

Now let's consider the case where the list contains one even number. The tests in the first two COND clauses will be false, so the function will end up at the third clause, where it calls itself recursively on the REST of the list. Since the REST is NIL, this reduces to a previously solved problem: (ANYODDP NIL) is NIL due to the first COND clause.

```
> (anyoddp '(6))
----Enter ANYODDP
|      X = (6)
|    ----Enter ANYODDP           Third clause: recursive call.
|    |      X = NIL
|     \--ANYODDP returned NIL    First clause returns NIL.
 \--ANYODDP returned NIL
NIL
```

If the list contains two elements, an even number followed by an odd number, the recursive call will trigger the second COND clause instead of the first:

```
> (anyoddp '(6 7))
----Enter ANYODDP
|      X = (6 7)
|    ----Enter ANYODDP           Third clause:  recursive call.
|    |      X = (7)
|     \--ANYODDP returned T      Second COND clause returns T.
 \--ANYODDP returned T
T
```

Finally, let's consider the general case where there are multiple even and odd numbers:

```
> (anyoddp '(2 4 6 7 8 9))
----Enter ANYODDP
|      X = (2 4 6 7 8 9)
|    ----Enter ANYODDP
|    |      X = (4 6 7 8 9)
|    |    ----Enter ANYODDP
|    |    |      X = (6 7 8 9)
|    |    |    ----Enter ANYODDP
|    |    |    |      X = (7 8 9)
|    |    |     \--ANYODDP returned T
|    |     \--ANYODDP returned T
|     \--ANYODDP returned T
 \--ANYODDP returned T
T
```

Note that in this example the function did not have to recurse all the way down to NIL. Since the FIRST of (7 8 9) is odd, ANYODDP could stop and return T at that point.

EXERCISES

8.1. Use a trace to show how ANYODDP would handle the list (3142 5798 6550 8914). Which COND clause is never true in this case?

8.2. Show how to write ANYODDP using IF instead of COND.

8.4 MARTIN VISITS THE DRAGON AGAIN

‘‘Hello Dragon!’’ Martin called as he made his way down the rickety dungeon staircase.

‘‘Hmmmph! You again. I’m on to your recursive tricks.’’ The dragon did not sound glad to see him.

‘‘I’m supposed to find out what five factorial is,’’ Martin said. ‘‘What’s *factorial* mean, anyway?’’

At this the dragon put on a most offended air and said, ‘‘I’m not going to tell you. Look it up in a book.’’

‘‘All right,’’ said Martin. ‘‘Just tell me what five factorial is and I’ll leave you alone.’’

‘‘You don’t know what factorial means, but you want *me* to tell you what factorial of five is??? All right buster, I’ll tell you, not that it will do you any good. Factorial of five is five times factorial of four. I hope you’re satisfied. Don’t forget to bolt the door on your way out.’’

‘‘But what’s factorial of four?’’ asked Martin, not at all pleased with the dragon’s evasiveness.

‘‘Factorial of four? Why, it’s four times factorial of three, of course.’’

‘‘And I suppose you’re going to tell me that factorial of three is three times factorial of two,’’ Martin said.

‘‘What a clever boy you are!’’ said the dragon. ‘‘Now go away.’’

‘‘Not yet,’’ Martin replied. ‘‘Factorial of two is two times factorial of one. Factorial of one is one times factorial of zero. Now what?’’

‘‘Factorial of zero is one,’’ said the dragon. ‘‘That’s really all you ever need to remember about factorials.’’

‘‘Hmmm,’’ said Martin. ‘‘There’s a pattern to this factorial function. Perhaps I should write down the steps I’ve gone through.’’ Here is what he wrote:

```
Factorial(5) = 5 × Factorial(4)
             = 5 × 4 × Factorial(3)
             = 5 × 4 × 3 × Factorial(2)
             = 5 × 4 × 3 × 2 × Factorial(1)
             = 5 × 4 × 3 × 2 × 1 × Factorial(0)
             = 5 × 4 × 3 × 2 × 1 × 1
```

‘‘Well,’’ said the dragon, ‘‘you’ve recursed all the way down to factorial of zero, which you know is one. Now why don’t you try working your way back up to....’’ When it realized what it was doing, the dragon stopped in mid-sentence. Dragons aren’t supposed to be helpful.

Martin started to write again:

```
                1 × 1= 1
            2 × 1 × 1= 2
        3 × 2 × 1 × 1= 6
    4 × 3 × 2 × 1 × 1= 24
5 × 4 × 3 × 2 × 1 × 1= 120
```

‘‘Hey!’’ Martin yelped. ‘‘Factorial of 5 is 120. That’s the answer! Thanks!!’’

‘‘*I* didn’t tell you the answer,’’ the dragon said testily. ‘‘*I* only told you that factorial of zero is one, and factorial of *n* is *n* times factorial of *n*−1. You did the rest yourself. Recursively, I might add.’’

‘‘That’s true,’’ said Martin. ‘‘Now if I only knew what ‘recursively’ really meant.’’

8.5 A LISP VERSION OF THE FACTORIAL FUNCTION

The dragon’s words gave a very precise definition of factorial: *n* factorial is *n* times *n−1* factorial, and zero factorial is one. Here is a function called FACT that computes factorials recursively:

```
(defun fact (n)
  (cond ((zerop n) 1)
        (t (* n (fact (- n 1))))))
```

And here is how Lisp would solve Martin's problem:

```
(dtrace fact)

> (fact 5)
----Enter FACT
|     N = 5
|   ----Enter FACT
|   |     N = 4
|   |   ----Enter FACT
|   |   |     N = 3
|   |   |   ----Enter FACT
|   |   |   |     N = 2
|   |   |   |   ----Enter FACT
|   |   |   |   |     N = 1
|   |   |   |   |   ----Enter FACT
|   |   |   |   |   |     N = 0
|   |   |   |   |    \--FACT returned 1
|   |   |   |    \--FACT returned 1
|   |   |    \--FACT returned 2
|   |    \--FACT returned 6
|    \--FACT returned 24
 \--FACT returned 120
120
```

EXERCISE

8.3. Why does (FACT 20.0) produce a different result than (FACT 20)? Why do (FACT 0.0) and (FACT 0) both produce the same result?

8.6 THE DRAGON'S DREAM

The next time Martin returned to the dungeon, he found the dragon rubbing its eyes, as if it had just awakened from a long sleep.

''I had a most curious dream,'' the dragon said. ''It was a recursive dream, in fact. Would you like to hear about it?''

Martin was stunned to find the dragon in something resembling a friendly mood. He forgot all about the alchemist's latest problem. ''Yes, please do tell me about your dream,'' he said.

''Very well,'' began the dragon. ''Last night I was looking at a long loaf of bread, and I wondered how many slices it would make. To answer my question I actually went and cut one slice from the loaf. I had one slice, and one slightly shorter loaf of bread, but no answer. I puzzled over the problem until I fell asleep.''

``And that's when you had the dream?'' Martin asked.

``Yes, a very curious one. I dreamt about another dragon who had a loaf of bread just like mine, except his was a slice shorter. And he too wanted to know how many slices his loaf would make, but he had the same problem I did. He cut off a slice, like me, and stared at the remaining loaf, like me, and then he fell asleep like me as well.''

``So neither one of you found the answer,'' Martin said disappointedly. ``You don't know how long your loaf is, and you don't know how long his is either, except that it's one slice shorter than yours.''

``But I'm not done yet,'' the dragon said. ``When the dragon in *my* dream fell asleep, *he* had a dream as well. He dreamt about—if you can imagine this—a dragon whose loaf of bread was one slice shorter than *his own* loaf. And this dragon also wanted to find out how many slices his loaf would make, and he tried to find out by cutting a slice, but that didn't tell him the answer, so he fell asleep thinking about it.''

``Dreams within dreams!!'' Martin exclaimed. ``You're making my head swim. Did that last dragon have a dream as well?''

``Yes, and he wasn't the last either. Each dragon dreamt of a dragon with a loaf one slice shorter than his own. I was piling up a pretty deep stack of dreams there.''

``How did you manage to wake up then?'' Martin asked.

``Well,'' the dragon said, ``eventually one of the dragons dreamt of a dragon whose loaf was so small it wasn't there at all. You might call it 'the empty loaf.' That dragon could see his loaf contained no slices, so he knew the answer to his question was zero; he didn't fall asleep.

``When the dragon who dreamt of that dragon woke up, he knew that since his own loaf was one slice longer, it must be exactly one slice long. So he awoke knowing the answer to his question.

``And, when the dragon who dreamt of *that* dragon woke up, *he* knew that his loaf had to be two slices long, since it was one slice longer than that of the dragon he dreamt about. And when the dragon who dreamt of *him* woke up...."

``I get it!'' Martin said. ``He added one to the length of the loaf of the dragon he dreamed about, and that answered his own question. And when *you* finally woke up, you had the answer to yours. How many slices did your loaf make?''

``Twenty-seven,'' said the dragon. ``It was a very long dream.''

8.7 A RECURSIVE FUNCTION FOR COUNTING SLICES OF BREAD

If we represent a slice of bread by a symbol, then a loaf can be represented as a list of symbols. The problem of finding how many slices a loaf contains is thus the problem of finding how many elements a list contains. This is of course what LENGTH does, but if we didn't have LENGTH, we could still count the slices recursively.

```
(defun count-slices (loaf)
  (cond ((null loaf) 0)
        (t (+ 1 (count-slices (rest loaf))))))

(dtrace count-slices)
```

If the input is the empty list, then its length is zero, so COUNT-SLICES simply returns zero.

```
> (count-slices nil)
----Enter COUNT-SLICES
|     LOAF = NIL
 \--COUNT-SLICES returned 0
0
```

If the input is the list (X), COUNT-SLICES calls itself recursively on the REST of the list, which is NIL, and then adds one to the result.

```
> (count-slices '(x))
----Enter COUNT-SLICES
|     LOAF = (X)
|   ----Enter COUNT-SLICES
|   |     LOAF = NIL
|    \--COUNT-SLICES returned 0
 \--COUNT-SLICES returned 1
1
```

When the input is a longer list, COUNT-SLICES has to recurse more deeply to get to the empty list so it can return zero. Then as each recursive call returns, one is added to the result.

```
> (count-slices '(x x x x x))
----Enter COUNT-SLICES
|     LOAF = (X X X X X)
|   ----Enter COUNT-SLICES
|   |     LOAF = (X X X X)
|   |   ----Enter COUNT-SLICES
|   |   |     LOAF = (X X X)
|   |   |   ----Enter COUNT-SLICES
|   |   |   |     LOAF = (X X)
|   |   |   |   ----Enter COUNT-SLICES
|   |   |   |   |     LOAF = (X)
|   |   |   |   |   ----Enter COUNT-SLICES
|   |   |   |   |   |     LOAF = NIL
|   |   |   |   |    \--COUNT-SLICES returned 0
|   |   |   |    \--COUNT-SLICES returned 1
|   |   |    \--COUNT-SLICES returned 2
|   |    \--COUNT-SLICES returned 3
|    \--COUNT-SLICES returned 4
 \--COUNT-SLICES returned 5
5
```

8.8 THE THREE RULES OF RECURSION

The dragon, beneath its feigned distaste for Martin's questions, actually enjoyed teaching him about recursion. One day it decided to formally explain what recursion means. The dragon told Martin to approach every recursive problem as if it were a journey. If he followed three rules for solving problems recursively, he would always complete the journey successfully. The dragon explained the rules this way:

1. Know when to stop.
2. Decide how to take one step.
3. Break the journey down into that step plus a smaller journey.

Let's see how each of these rules applies to the Lisp functions we wrote. The first rule, ''know when to stop,'' warns us that any recursive function must check to see if the journey has been completed before recursing further. Usually this is done in the first COND clause. In ANYODDP the first clause checks if the input is the empty list, and if so the function stops and returns NIL, since the empty list doesn't contain any numbers. The factorial function, FACT, stops when the input gets down to zero. Zero factorial is one, and, as the dragon said, that's all you ever need to remember about factorial. The rest is computed recursively. In COUNT-SLICES the first COND clause checks for NIL, ''the empty loaf.'' COUNT-SLICES returns zero if NIL is the input. Again, this is based on the realization that the empty loaf contains no slices, so we do not have to recurse any further.

The second rule, ''decide how to take one step,'' asks us to break off from the problem one tiny piece that we instantly know how to solve. In ANYODDP we check whether the FIRST of a list is an odd number; if so we return T. In the factorial function we perform a single multiplication, multiplying the input N by factorial of N−1. In COUNT-SLICES the step is the + function: For each slice we cut off the loaf, we add one to whatever the length of the resulting loaf turned out to be.

The third rule, ''break the journey down into that step plus a smaller journey,'' means find a way for the function to call itself recursively on the slightly smaller problem that results from breaking a tiny piece off. The ANYODDP function calls itself on the REST of the list, a shorter list than the original, to see if there are any odd numbers there. The factorial function recursively computes factorial of N-1, a slightly simpler problem than factorial of N, and then uses the result to get factorial of N. In COUNT-SLICES we use a recursive call to count the number of slices in the REST of a loaf, and then add one to the result to get the size of the whole loaf.

The Dragon's Three Recursive Functions				
Function	*Stop When Input Is*	*Return*	*Step to Take*	*Rest of Problem*
ANYODDP	NIL	NIL	(ODDP (FIRST X))	(ANYODDP (REST X))
FACT	0	1	N × ...	(FACT (- N 1))
COUNT-SLICES	NIL	0	1 + ...	(COUNT-SLICES (REST LOAF))

Table 8-1 Applying the three rules of recursion.

Table 8-1 sums up our understanding of how the three rules apply to ANYODDP, FACT, and COUNT-SLICES. Now that you know the rules, you can write your own recursive functions.

FIRST RECURSION EXERCISE

8.4. We are going to write a function called LAUGH that takes a number as input and returns a list of that many HAs. (LAUGH 3) should return the list (HA HA HA). (LAUGH 0) should return a list with no HAs in it, or, as the dragon might put it, ‘‘the empty laugh.’’

Here is a skeleton for the LAUGH function:

```
(defun laugh (n)
  (cond (α  β)
        (t (cons 'ha γ))))
```

Under what condition should the LAUGH function stop recursing? Replace the symbol α in the skeleton with that condition. What value should LAUGH return for that case? Replace symbol β in the skeleton with that value. Given that a single step for this problem is to add a HA onto the result of a subproblem, fill in that subproblem by replacing the symbol γ.

Type your LAUGH function into the computer. Then type (DTRACE LAUGH) to trace it, and (LAUGH 5) to test it. Do you get the result you want? What happens for (LAUGH 0)? What happens for (LAUGH -1)?

Note: If the function looks like it’s in an infinite loop, break out of it and get back to the read-eval-print loop. (Exactly how this is done depends on the particular version of Lisp you use. Ask your local Lisp expert if you need help.) Then use DTRACE to help you understand what’s going on.

EXERCISES

8.5. In this exercise we are going to write a function ADD-UP to add up all the numbers in a list. (ADD-UP ’(2 3 7)) should return 12. You already know how to solve this problem applicatively with REDUCE; now you’ll learn to solve it recursively. Before writing ADD-UP we must answer three questions posed by our three rules of recursion.

a. When do we stop? Is there any list for which we immediately *know* what the sum of all its elements is? What is that list? What value should the function return if it gets that list as input?

b. Do we know how to take a single step? Look at the second COND clause in the definition of COUNT-SLICES or FACT.

Does this give you any ideas about what the single step should be for ADD-UP?

c. How should ADD-UP call itself recursively to solve the rest of the problem? Look at COUNT-SLICES or FACT again if you need inspiration.

Write down the complete definition of ADD-UP. Type it into the computer. Trace it, and then try adding up a list of numbers.

8.6. Write ALLODDP, a recursive function that returns T if all the numbers in a list are odd.

8.7. Write a recursive version of MEMBER. Call it REC-MEMBER so you don't redefine the built-in MEMBER function.

8.8. Write a recursive version of ASSOC. Call it REC-ASSOC.

8.9. Write a recursive version of NTH. Call it REC-NTH.

8.10. For x a nonnegative integer and y a positive integer, $x+y$ equals $x+1+(y-1)$. If y is zero then $x+y$ equals x. Use these equations to build a recursive version of + called REC-PLUS out of ADD1, SUB1, COND and ZEROP. You'll have to write ADD1 and SUB1 too.

8.9 MARTIN DISCOVERS INFINITE RECURSION

On his next trip down to the dungeon Martin brought with him a parchment scroll. ''Look dragon,'' he called, ''someone else must know about recursion. I found this scroll in the alchemist's library.''

The dragon peered suspiciously as Martin unrolled the scroll, placing a candlestick at each end to hold it flat. ''This scroll makes no sense,'' the dragon said. ''For one thing, it's got far too many parentheses.''

''The writing *is* a little strange,'' Martin agreed, ''but I think I've figured out the message. It's an algorithm for computing Fibonacci numbers.''

''I already know how to compute Fibonacci numbers,'' said the dragon.

''Oh? How?''

''Why, I wouldn't *dream* of spoiling the fun by telling you,'' the dragon replied.

''I didn't think you would,'' Martin shot back. ''But the scroll says that Fib of n equals Fib of $n-1$ plus Fib of $n-2$. That's a *recursive* definition, and I