COMMON LISP: A Gentle Introduction to Symbolic Computation

David S. Touretzky

Carnegie Mellon University

The Benjamin/Cummings Publishing Company,Inc.

Redwood City, California • Fort Collins, Colorado • Menlo Park, California
Reading, Massachusetts • New York • Don Mill, Ontario • Workingham, U.K.
Amsterdam • Bonn • Sydney • Singapore • Tokyo • Madrid • San Juan

Sponsoring Editor: Alan Apt
Developmental Editor: Mark McCormick
Production Coordinator: John Walker
Copy Editor: Steven Sorenson
Text and Cover Designer: Michael Rogondino
Cover image selected by David S. Touretzky
Cover: La Grande Vitesse, sculpture by Alexander Calder

Published by The Benjamin/Cummings Publishing Company, Inc.

Library of Congress Cataloging-in-Publication Data

Touretzky, David S.
Common LISP : a gentle introduction to symbolic computation /
David S. Touretzky
p. cm.
Includes index.
ISBN 0-8053-0492-4
1. COMMON LISP (Computer program language) I. Title.
QA76.73.C28T68 1989
005.13'3–dc20 89-15180
CIP

ISBN 0-8053-0492-4

ABCDEFGHIJK - DO - 8932109

The Benjamin/Cummings Publishing Company, Inc.
390 Bridge Parkway
Redwood City, California 94065

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

already know how to work with recursion.''

''What else does the scroll say?'' the dragon asked.

''Nothing else. Should it say more?''

Suddenly the dragon assumed a most ingratiating tone. Martin found the change startling. ''Dearest boy! Would you do a poor old dragon one tiny little favor? Compute a Fibonacci number for me. I promise to only ask you for a small one.''

''Well, I'm supposed to be upstairs now, cleaning the cauldrons,'' Martin began, but seeing the hurt look on the dragon's face he added, ''but I guess I have time for a *small* one.''

''You won't regret it,'' promised the dragon. ''Tell me: What is Fib of four?''

Martin traced his translation of the Fibonacci algorithm in the dust:

```
Fib(n)  =  Fib(n-1) + Fib(n-2)
```

Then he began to compute Fib of four:

```
Fib(4)   =  Fib(3) + Fib(2)
Fib(3)   =  Fib(2) + Fib(1)
Fib(2)   =  Fib(1) + Fib(0)
Fib(1)   =  Fib(0) + Fib(-1)
Fib(0)   =  Fib(-1) + Fib(-2)
Fib(-1)  =  Fib(-2) + Fib(-3)
Fib(-2)  =  Fib(-3) + Fib(-4)
Fib(-3)  =  Fib(-4) + Fib(-5)
```

''Finished?'' the dragon asked innocently.

''No,'' Martin replied. ''Something is wrong. The numbers are becoming increasingly negative.''

''Well, will you be finished soon?''

''It looks like I won't ever be finished,'' Martin said. ''This recursion keeps going on forever.''

''Aha! You see? You're stuck in an *infinite* recursion!'' the dragon gloated. ''I noticed it at once.''

''Then why didn't you say something?'' Martin demanded.

The dragon grimaced and gave a little snort; blue flame appeared briefly in its nostrils. ''How will you *ever* come to master recursion if you rely on a dragon to do your thinking for you?''

Martin wasn't afraid, but he stepped back a bit anyway to let the smoke clear. ''Well, how did you spot the problem so *quickly*, dragon?''

''Elementary, boy. The scroll told how to take a single step, and how to break the journey down to a smaller one. It said nothing at all about when you get to stop. Ergo,'' the dragon grinned, ''you don't.''

8.10 INFINITE RECURSION IN LISP

Lisp functions can be made to recurse infinitely by ignoring the dragon's first rule of recursion, which is to know when to stop. Here is the Lisp implementation of Martin's algorithm:

```
(defun fib (n)
  (+ (fib (- n 1))
     (fib (- n 2))))

(dtrace fib)

> (fib 4)
----Enter FIB
|     N = 4
|   ----Enter FIB
|   |     N = 3
|   |   ----Enter FIB
|   |   |     N = 2
|   |   |   ----Enter FIB
|   |   |   |     N = 1
|   |   |   |   ----Enter FIB
|   |   |   |   |     N = 0
|   |   |   |   |   ----Enter FIB
|   |   |   |   |   |     N = -1
|   |   |   |   |   |   ----Enter FIB
|   |   |   |   |   |   |     N = -2
|   |   |   |   |   |   |   ----Enter FIB
|   |   |   |   |   |   |   |     N = -3
```

ad infinitum

Usually a good programmer can tell just by looking at a function whether it will exhibit infinite recursion, but in some cases this can be quite difficult to determine. Try tracing the following function C, giving it inputs that are small positive integers:

```
(defun c (n)
  (cond ((equal n 1) t)
        ((evenp n) (c (/ n 2)))
        (t (c (+ (* 3 n) 1)))))
```

```
> (c 3)
----Enter C
|     N = 3
|   ----Enter C
|   |     N = 10
|   |   ----Enter C
|   |   |     N = 5
|   |   |   ----Enter C
|   |   |   |     N = 16
|   |   |   |   ----Enter C
|   |   |   |   |     N = 8
|   |   |   |   |   ----Enter C
|   |   |   |   |   |     N = 4
|   |   |   |   |   |   ----Enter C
|   |   |   |   |   |   |     N = 2
|   |   |   |   |   |   |   ----Enter C
|   |   |   |   |   |   |   |     N = 1
|   |   |   |   |   |   |   |    \--C returned T
|   |   |   |   |   |   |    \--C returned T
|   |   |   |   |   |    \--C returned T
|   |   |   |   |    \--C returned T
|   |   |   |    \--C returned T
|   |   |    \--C returned T
|   |    \--C returned T
|    \--C returned T
 \--C returned T
T
```

Try calling C on other values between one and ten. Notice that there is no obvious relationship between the size of the input and the number of recursive calls that result. Number theorists believe the function returns T for every positive integer, in other words, there are no inputs which cause it to recurse infinitely. This is known as Collatz's conjecture. But until the conjecture is proved, we can't say for certain whether or not C always returns.

EXERCISES

8.11. The missing part of Martin's Fibonacci algorithm is the rule for Fib(1) and Fib(0). Both of these are defined to be one. Using this

information, write a correct version of the FIB function. (FIB 4) should return five. (FIB 5) should return eight.

8.12. Consider the following version of ANY-7-P, a recursive function that searches a list for the number seven:

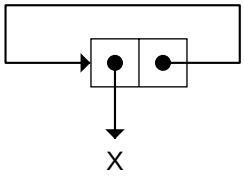
```
(defun any-7-p (x)
  (cond ((equal (first x) 7) t)
        (t (any-7-p (rest x)))))
```

Give a sample input for which this function will work correctly. Give one for which the function will recurse infinitely.

8.13. Review the definition of the factorial function, FACT, given previously. What sort of input could you give it to cause an infinite recursion?

8.14. Write the very shortest infinite recursion function you can.

8.15. Consider the circular list shown below. What is the car of this list? What is the cdr? What will the COUNT-SLICES function do when given this list as input?



8.11 RECURSION TEMPLATES

Most recursive Lisp functions fall into a few standard forms. These are described by **recursion templates**, which capture the essence of the form in a fill-in-the-blanks pattern. You can create new functions by choosing a template and filling in the blanks. Also, once you've mastered them, you can use the templates to analyze existing functions to see which pattern they fit.

8.11.1 Double-Test Tail Recursion

The first template we'll study is double-test tail recursion, which is shown in Figure 8-1. ''Double-test'' indicates that the recursive function has two end tests; if either is true, the corresponding end value is returned instead of proceeding with the recursion. When both end tests are false, we end up at the

Double-Test Tail Recursion

Template:

```
(DEFUN func (X)
  (COND (end-test-1  end-value-1)
        (end-test-2  end-value-2)
        (T (func  reduced-x))))
```

Example:

Func:	ANYODDP
End-test-1:	(NULL X)
End-value-1:	NIL
End-test-2:	(ODDP (FIRST X))
End-value-2:	T
Reduced-x:	(REST X)

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

Figure 8-1 Template for double-test tail recursion.

last COND clause, where the function reduces the input somehow and then calls itself recursively. This template is said to be **tail-recursive** because the action part of the last COND clause does not do any work after the recursive call. Whatever result the recursive call produces, that is what the COND returns, so that is what each parent call returns. ANYODDP is an example of a tail-recursive function.

EXERCISES

8.16. What would happen if we switched the first and second COND clauses in ANYODDP?

8.17. Use double-test tail recursion to write FIND-FIRST-ODD, a function that returns the first odd number in a list, or NIL if there are none. Start by copying the recursion template values for ANYODDP; only a small change is necessary to derive FIND-FIRST-ODD.

8.11.2 Single-Test Tail Recursion

A simpler but less frequently used template is single-test tail recursion, which is shown in Figure 8-2. Suppose we want to find the first atom in a list, where the list may be nested arbitrarily deeply. We can do this by taking successive FIRSTs of the list until we reach an atom. The function FIND-FIRST-ATOM does this:

```
(find-first-atom '(ooh ah eee))  ⇒  ooh

(find-first-atom '((((a f)) i) r))  ⇒  a

(find-first-atom 'fred)  ⇒  fred
```

In general, single-test recursion is used when we know the function will always find what it's looking for eventually; FIND-FIRST-ATOM is guaranteed to find an atom if it keeps taking successive FIRSTs of its input. We use double-test recursion when there is the possibility the function might not find what it's looking for. In ANYODDP, for example, the second test checked if it had found an odd number, but first a test was needed to see if the function had run off the end of the list, in which case it should return NIL.

EXERCISES

8.18. Use single-test tail recursion to write LAST-ELEMENT, a function that returns the last element of a list. LAST-ELEMENT should recursively

Single-Test Tail Recursion

Template:

```
(DEFUN func (X)
  (COND (end-test end-value)
        (T (func reduced-x))))
```

Example:

Func:	FIND-FIRST-ATOM
End-test:	(ATOM X)
End-value:	X
Reduced-x:	(FIRST X)

```
(defun find-first-atom (x)
  (cond ((atom x) x)
        (t (find-first-atom (first x)))))
```

Figure 8-2 Template for single-test tail recursion.

travel down the list until it reaches the last cons cell (a cell whose cdr is an atom); then it should return the car of this cell.

8.19. Suppose we decided to convert ANYODDP to single-test tail recursion by simply eliminating the COND clause with the NULL test. For which inputs would it still work correctly? What would happen in those cases where it failed to work correctly?

8.11.3 Augmenting Recursion

Augmenting recursive functions like COUNT-SLICES build up their result bit-by-bit. We call this process **augmentation**. Instead of dividing the problem into an initial step plus a smaller journey, they divide it into a smaller journey plus a final step. The final step consists of choosing an augmentation value and applying it to the result of the previous recursive call. In COUNT-SLICES, for example, we built up the result by first making a recursive call and then adding one to the result. A template for single-test augmenting recursion is shown in Figure 8-3.

No augmentation of the result is permitted in tail-recursive functions. Therefore, the value returned by a tail-recursive function is always equal to one of the end-values in the function definition; it isn't built up bit-by-bit as each recursive call returns. Compare ANYODDP, which always returns T or NIL; it never augments its result.

EXERCISES

8.20. Of the three templates we've seen so far, which one describes FACT, the factorial function? Write down the values of the various template components for FACT.

8.21. Write a recursive function ADD-NUMS that adds up the numbers N, N−1, N−2, and so on, down to 0, and returns the result. For example, (ADD-NUMS 5) should compute 5+4+3+2+1+0, which is 15.

8.22. Write a recursive function ALL-EQUAL that returns T if the first element of a list is equal to the second, the second is equal to the third, the third is equal to the fourth, and so on. (ALL-EQUAL '(I I I I)) should return T. (ALL-EQUAL '(I I E I)) should return NIL. ALL-EQUAL should return T for lists with less than two elements. Does this problem require augmentation? Which template will you use to solve it?

Single-Test Augmenting Recursion

Template:

```
(DEFUN func (X)
  (COND (end-test  end-value)
        (T (aug-fun  aug-val
                     (func  reduced-x)))))
```

Example:

Func:	COUNT-SLICES
End-test:	(NULL X)
End-value:	0
Aug-fun:	+
Aug-val:	1
Reduced-x:	(REST X)

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

Figure 8-3 Template for single-test augmenting recursion.

8.12 VARIATIONS ON THE BASIC TEMPLATES

The templates we've learned so far have many uses. Certain ways of using them are especially common in Lisp programming, and deserve special mention. In this section we'll cover four variations on the basic templates.

8.12.1 List-Consing Recursion

List-consing recursion is used very frequently in Lisp. It is a special case of augmenting recursion where the augmentation function is CONS. As each recursive call returns, we create one new cons cell. Thus, the depth of the recursion is equal to the length of the resulting cons cell chain, plus one (because the last call returns NIL instead of a cons). The LAUGH function you wrote in the first recursion exercise is an example of list-consing recursion. See Figure 8-4 for the template.

EXERCISES

8.23. Suppose we evaluate (LAUGH 5). Make a table showing, for each call to LAUGH, the value of N (from five down to zero), the value of the first input to CONS, the value of the second input to CONS, and the result returned by LAUGH.

8.24. Write COUNT-DOWN, a function that counts down from *n* using list-consing recursion. (COUNT-DOWN 5) should produce the list (5 4 3 2 1).

8.25. How could COUNT-DOWN be used to write an applicative version of FACT? (You may skip this problem if you haven't read Chapter 7 yet.)

8.26. Suppose we wanted to modify COUNT-DOWN so that the list it constructs ends in zero. For example, (COUNT-DOWN 5) would produce (5 4 3 2 1 0). Show two ways this can be done.

8.27. Write SQUARE-LIST, a recursive function that takes a list of numbers as input and returns a list of their squares. (SQUARE-LIST '(3 4 5 6)) should return (9 16 25 36).

List-Consing Recursion
(A Special Case of Augmenting Recursion)

Template:

```
(DEFUN func (N)
  (COND (end-test NIL)
        (T (CONS new-element
                 (func reduced-n)))))
```

Example:

Func:	LAUGH
End-test:	(ZEROP N)
New-element:	'HA
Reduced-n:	(- N 1)

```
(defun laugh (n)
  (cond ((zerop n) nil)
        (t (cons 'ha (laugh (- n 1))))))
```

Figure 8-4 Template for list-consing recursion.

8.12.2 Simultaneous Recursion on Several Variables

Simultaneous recursion on multiple variables is a straightforward extension to any recursion template. Instead of having only one input, the function has several, and one or more of them is ‘‘reduced’’ with each recursive call. For example, suppose we want to write a recursive version of NTH, called MY-NTH. Recall that (NTH 0 *x*) is (FIRST *x*); this tells us which end test to use. With each recursive call we reduce *n* by one and take successive RESTs of the list *x*. The resulting function demonstrates single-test tail recursion with simultaneous recursion on two variables. The template is shown in Figure 8-5. Here is a trace in which you can see the two variables being reduced simultaneously.

```
(defun my-nth (n x)
  (cond ((zerop n) (first x))
        (t (my-nth (- n 1) (rest x)))))

> (my-nth 2 '(a b c d e))
----Enter MY-NTH
|     N = 2
|     X = (A B C D E)
|   ----Enter MY-NTH
|   |     N = 1
|   |     X = (B C D E)
|   |   ----Enter MY-NTH
|   |   |     N = 0
|   |   |     X = (C D E)
|   |    \--MY-NTH returned C
|    \--MY-NTH returned C
 \--MY-NTH returned C
C
```

EXERCISES

8.28. The expressions (MY-NTH 5 '(A B C)) and (MY-NTH 1000 '(A B C)) both run off the end of the list. and hence produce a NIL result. Yet the second expression takes quite a bit longer to execute than the first. Modify MY-NTH so that the recursion stops as soon the function runs off the end of the list.

8.29. Write MY-MEMBER, a recursive version of MEMBER. This function will take two inputs, but you will only want to reduce one of them with each successive call. The other should remain unchanged.

Simultaneous Recursion on Several Variables
(Using the Single-Test Tail Recursion Template)

Template:

```
(DEFUN func (N X)
  (COND (end-test end-value)
        (T (func reduced-n reduced-x))))
```

Example:

Func:	MY-NTH
End-test:	(ZEROP N)
End-value:	(FIRST X)
Reduced-n:	(- N 1)
Reduced-x:	(REST X)

```
(defun my-nth (n x)
  (cond ((zerop n) (first x))
        (t (my-nth (- n 1) (rest x)))))
```

Figure 8-5 Template for simultaneous recursion on several variables, using single-test tail recursion.

8.30. Write MY-ASSOC, a recursive version of ASSOC.

8.31. Suppose we want to tell as quickly as possible whether one list is shorter than another. If one list has five elements and the other has a million, we don't want to have to go through all one million cons cells before deciding that the second list is longer. So we must not call LENGTH on the two lists. Write a recursive function COMPARE-LENGTHS that takes two lists as input and returns one of the following symbols: SAME-LENGTH, FIRST-IS-LONGER, or SECOND-IS-LONGER. Use triple-test simultaneous recursion. *Hint:* If *x* is shorter than *y* and both are nonempty, then (REST *x*) is shorter than (REST *y*).

8.12.3 Conditional Augmentation

In some list-processing problems we want to skip certain elements of the list and use only the remaining ones to build up the result. This is known as **conditional augmentation**. For example, in EXTRACT-SYMBOLS, defined on the facing page, only elements that are symbols will be included in the result.

```
> (extract-symbols '(3 bears and 1 girl))
----Enter EXTRACT-SYMBOLS
|     X = (3 BEARS AND 1 GIRL)
|   ----Enter EXTRACT-SYMBOLS
|   |     X = (BEARS AND 1 GIRL)
|   |   ----Enter EXTRACT-SYMBOLS
|   |   |     X = (AND 1 GIRL)
|   |   |   ----Enter EXTRACT-SYMBOLS
|   |   |   |     X = (1 GIRL)
|   |   |   |   ----Enter EXTRACT-SYMBOLS
|   |   |   |   |     X = (GIRL)
|   |   |   |   |   ----Enter EXTRACT-SYMBOLS
|   |   |   |   |   |     X = NIL
|   |   |   |   |    \--EXTRACT-SYMBOLS returned NIL
|   |   |   |    \--EXTRACT-SYMBOLS returned (GIRL)
|   |   |    \--EXTRACT-SYMBOLS returned (GIRL)
|   |    \--EXTRACT-SYMBOLS returned (AND GIRL)
|    \--EXTRACT-SYMBOLS returned (BEARS AND GIRL)
 \--EXTRACT-SYMBOLS returned (BEARS AND GIRL)
(BEARS AND GIRL)
```

The body of EXTRACT-SYMBOLS contains two recursive calls. One call is nested inside an augmentation expression, which in this case conses a new

Conditional Augmentation

Template:

```
(DEFUN func (X)
  (COND (end-test end-value)
        (aug-test (aug-fun aug-val
                           (func reduced-x))
        (T (func reduced-x))))
```

Example:

Func:	EXTRACT-SYMBOLS
End-test:	(NULL X)
End-value:	NIL
Aug-test:	(SYMBOLP (FIRST X))
Aug-fun:	CONS
Aug-val:	(FIRST X)
Reduced-x:	(REST X)

```
(defun extract-symbols (x)
  (cond ((null x) nil)
        ((symbolp (first x))
         (cons (first x)
               (extract-symbols (rest x))))
        (t (extract-symbols (rest x)))))
```

Figure 8-6 Template for conditional augmentation.

element onto the result list. The other call is unaugmented; instead its result is simply returned. In the preceding trace output you'll note that sometimes two successive calls return the same value, such as two lists (GIRL) and two lists (BEARS AND GIRL); that's because one of each pair of calls chose the unaugmented COND clause. When the augmented clause was chosen, the result got longer, as when we went from NIL to (GIRL), from there to (AND GIRL), and from there to (BEARS AND GIRL). See Figure 8-6 for the general template for conditional augmentation.

EXERCISES

8.32. Write the function SUM-NUMERIC-ELEMENTS, which adds up all the numbers in a list and ignores the non-numbers. (SUM-NUMERIC-ELEMENTS '(3 BEARS 3 BOWLS AND 1 GIRL)) should return seven.

8.33. Write MY-REMOVE, a recursive version of the REMOVE function.

8.34. Write MY-INTERSECTION, a recursive version of the INTERSECTION function.

8.35. Write MY-SET-DIFFERENCE, a recursive version of the SET-DIFFERENCE function.

8.36. The function COUNT-ODD counts the number of odd elements in a list of numbers; for example, (COUNT-ODD '(4 5 6 7 8)) should return two. Show how to write COUNT-ODD using conditional augmentation. Then write another version of COUNT-ODD using the regular augmenting recursion template. (To do this you will need to write a conditional expression for the augmentation value.)

8.12.4 Multiple Recursion

A function is **multiple recursive** if it makes more than one recursive call with each invocation. (Don't confuse simultaneous with multiple recursion. The former technique just reduces several variables simultaneously; it does not involve multiple recursive calls with each invocation.) The Fibonacci function is a classic example of multiple recursion. Fib(N) calls itself twice: once for Fib(N−1) and again for Fib(N−2). The results of the two calls are combined using +. A general template for multiple recursion is shown in Figure 8-7.

A good way to visualize the process of multiple recursion is to look at the shape of the nested calls in the trace output. Let's define a **terminal call** as a

Multiple Recursion

Template:

```
(DEFUN func (N)
  (COND (end-test-1  end-value-1)
        (end-test-2  end-value-2)
        (T (combiner  (func  first-reduced-n)
                      (func  second-reduced-n)))))
```

Example:

Func:	FIB
End-test-1:	(EQUAL N 0)
End-value-1:	1
End-test-2:	(EQUAL N 1)
End-value-2:	1
Combiner:	+
First-reduced-n:	(– N 1)
Second-reduced-n:	(– N 2)

```
(defun fib (n)
  (cond ((equal n 0) 1)
        ((equal n 1) 1)
        (t (+ (fib (- n 1))
              (fib (- n 2))))))
```

Figure 8-7 Template for multiple recursion.

call that does not recurse any further. In all previous functions, successive calls were nested strictly one inside the other, and the innermost call was the only terminal call. Then, the return values flowed in a straight line from the innermost call back to the outermost. But with a multiple-recursive function such as FIB, each call produces *two* new calls. The two are nested inside the parent call, but they cannot nest inside each other. Instead they appear side by side within the parent. Multiple recursive functions therefore have many terminal calls. In the following trace output, there are three terminal calls and two nonterminal calls.

```
> (fib 3)
----Enter FIB
|     N = 3
|   ----Enter FIB
|   |     N = 2
|   |   ----Enter FIB
|   |   |     N = 1
|   |    \--FIB returned 1
|   |   ----Enter FIB
|   |   |     N = 0
|   |    \--FIB returned 1
|    \--FIB returned 2
|   ----Enter FIB
|   |     N = 1
|    \--FIB returned 1
 \--FIB returned 3
3
```

EXERCISE

8.37. Define a simple function COMBINE that takes two numbers as input and returns their sum. Now replace the occurence of + in FIB with COMBINE. Trace FIB and COMBINE, and try evaluating (FIB 3) or (FIB 4). What can you say about the relationship between COMBINE, terminal calls, and nonterminal calls?

8.13 TREES AND CAR/CDR RECURSION

Sometimes we want to process all the elements of a nested list, not just the top-level elements. If the list is irregularly shaped, such as (((GOLDILOCKS . AND)) (THE . 3) BEARS), this might appear difficult. When we write our function, we won't know how long or how deeply nested its inputs will be.

CAR/CDR Recursion
(A Special Case of Multiple Recursion)

Template:

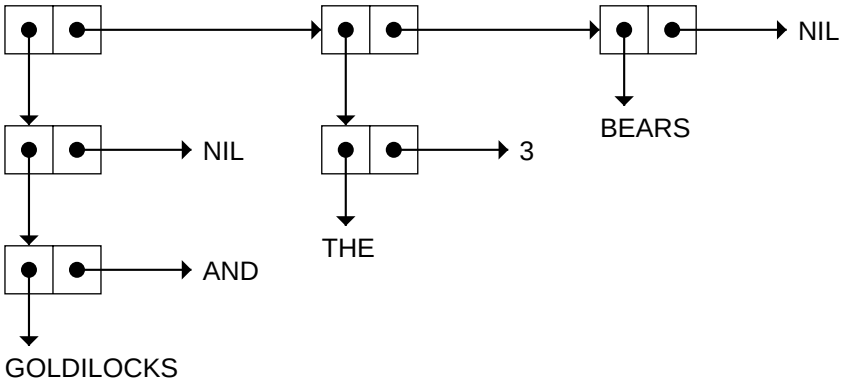
```
(DEFUN func (X)
  (COND (end-test-1  end-value-1)
        (end-test-2  end-value-2)
        (T (combiner (func (CAR X))
                     (func (CDR X))))))
```

Example:

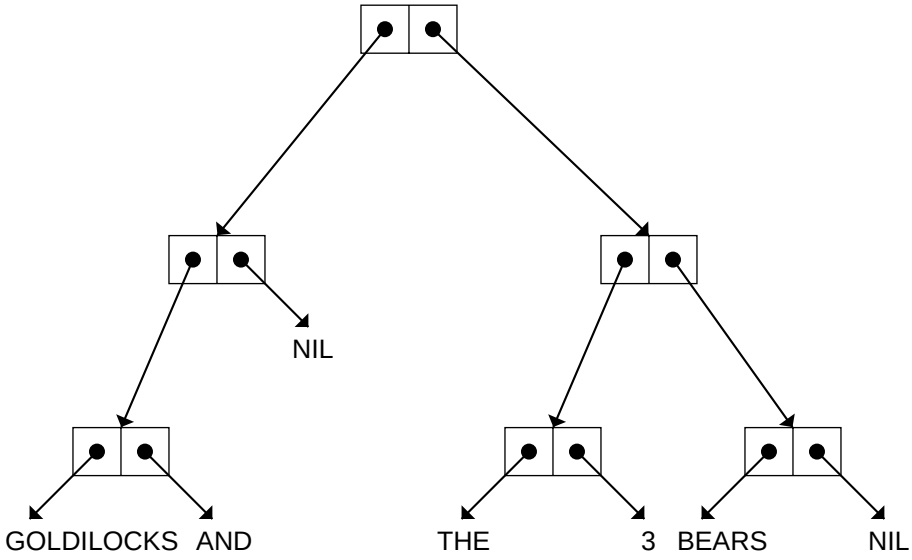
Func:	FIND-NUMBER
End-test-1:	(NUMBERP X)
End-value-1:	X
End-test-2:	(ATOM X)
End-value-2:	NIL
Combiner:	OR

```
(defun find-number (x)
  (cond ((numberp x) x)
        ((atom x) nil)
        (t (or (find-number (car x))
               (find-number (cdr x))))))
```

Figure 8-8 Template for CAR/CDR recursion.



The trick to solving this problem is not to think of the input as an irregularly shaped nested list, but rather as a binary tree (see the following illustration.) Binary trees are very regular: Each node is either an atom or a cons with two branches, the car and the cdr. Therefore all our function has to do is process the atoms, and call itself recursively on the car and cdr of each cons. This technique is called CAR/CDR recursion; it is a special case of multiple recursion.



For example, suppose we want a function FIND-NUMBER to search a tree and return the first number that appears in it, or NIL if there are none. Then we should use NUMBERP and ATOM as our end tests and OR as the combiner. (See the template in Figure 8-8.) Note that since OR is a conditional, as soon as one clause of the OR evaluates to true, the OR stops and returns that value. Thus we don't have to search the whole tree; the function will stop recursing as soon as any call results in a non-NIL value.

Besides tree searching, another common use for CAR/CDR recursion is to build trees by using CONS as the combiner. For example, here is a function that takes a tree as input and returns a new tree in which every non-NIL atom has been replaced by the symbol Q.

```
(defun atoms-to-q (x)
  (cond ((null x) nil)
        ((atom x) 'q)
        (t (cons (atoms-to-q (car x))
                 (atoms-to-q (cdr x))))))

> (atoms-to-q '(a . b))
(Q . Q)

> (atoms-to-q '(hark (harold the angel) sings))
(Q (Q Q Q) Q)
```

EXERCISES

8.38. What would be the effect of deleting the first COND clause in ATOMS-TO-Q?

8.39. Write a function COUNT-ATOMS that returns the number of atoms in a tree. (COUNT-ATOMS '(A (B) C)) should return five, since in addition to A, B, and C there are two NILs in the tree.

8.40. Write COUNT-CONS, a function that returns the number of cons cells in a tree. (COUNT-CONS '(FOO)) should return one. (COUNT-CONS '(FOO BAR)) should return two. (COUNT-CONS '((FOO))) should also return two, since the list ((FOO)) requires two cons cells. (COUNT-CONS 'FRED) should return zero.

8.41. Write a function SUM-TREE that returns the sum of all the numbers appearing in a tree. Nonnumbers should be ignored. (SUM-TREE '((3 BEARS) (3 BOWLS) (1 GIRL))) should return seven.

8.42. Write MY-SUBST, a recursive version of the SUBST function.

8.43. Write FLATTEN, a function that returns all the elements of an arbitrarily nested list in a single-level list. (FLATTEN '((A B (R)) A C (A D ((A (B)) R) A))) should return (A B R A C A D A B R A).

8.44. Write a function TREE-DEPTH that returns the maximum depth of a binary tree. (TREE-DEPTH '(A . B)) should return one. (TREE-DEPTH '((A B C D))) should return five, and (TREE-DEPTH '((A . B) . (C . D))) should return two.

8.45. Write a function PAREN-DEPTH that returns the maximum depth of nested parentheses in a list. (PAREN-DEPTH '(A B C)) should return one, whereas TREE-DEPTH would return three. (PAREN-DEPTH '(A B ((C) D) E)) should return three, since there is an element C that is nested in three levels of parentheses. *Hint:* This problem can be solved by CAR/CDR recursion, but the CAR and CDR cases will not be exactly symmetric.

8.14 USING HELPING FUNCTIONS

For some problems it is useful to structure the solution as a helping function plus a recursive function. The recursive function does most of the work. The helping function is the one that you call from top level; it performs some special service either at the beginning or the end of the recursion. For example, suppose we want to write a function COUNT-UP that counts from one up to *n*:

```
(count-up 5)  ⇒  (1 2 3 4 5)

(count-up 0)  ⇒  nil
```

This problem is harder than COUNT-DOWN because the innermost recursive call must terminate the recursion when the input reaches five (in the preceding example), not zero. In general, how will the function know when to stop? The easiest way is to supply the original value of N to the recursive function so it can decide when to stop. We must also supply an extra argument: a counter that tells the function how far along it is in the recursion. The job of the helping function is to provide the initial value for the counter.

```
(defun count-up (n)
  (count-up-recursively 1 n))
```

```
(defun count-up-recursively (cnt n)
  (cond ((> cnt n) nil)
        (t (cons cnt
                 (count-up-recursively
                   (+ cnt 1) n)))))

(dtrace count-up count-up-recursively)
```

```
> (count-up 3)
----Enter COUNT-UP
|     N = 3
|   ----Enter COUNT-UP-RECURSIVELY
|   |     CNT = 1
|   |     N = 3
|   |   ----Enter COUNT-UP-RECURSIVELY
|   |   |     CNT = 2
|   |   |     N = 3
|   |   |   ----Enter COUNT-UP-RECURSIVELY
|   |   |   |     CNT = 3
|   |   |   |     N = 3
|   |   |   |   ----Enter COUNT-UP-RECURSIVELY
|   |   |   |   |     CNT = 4
|   |   |   |   |     N = 3
|   |   |   |    \--COUNT-UP-RECURSIVELY returned NIL
|   |   |    \--COUNT-UP-RECURSIVELY returned (3)
|   |    \--COUNT-UP-RECURSIVELY returned (2 3)
|    \--COUNT-UP-RECURSIVELY returned (1 2 3)
 \--COUNT-UP returned (1 2 3)
(1 2 3)
```

EXERCISES

8.46. Another way to solve the problem of counting upward is to to add an element to the end of the list with each recursive call instead of adding elements to the beginning. This approach doesn't require a helping function. Write this version of COUNT-UP.

8.47. Write MAKE-LOAF, a function that returns a loaf of size N. (MAKE-LOAF 4) should return (X X X X). Use IF instead of COND.

8.48. Write a recursive function BURY that buries an item under *n* levels of parentheses. (BURY 'FRED 2) should return ((FRED)), while (BURY 'FRED 5) should return (((((FRED))))). Which recursion template did you use?

8.49. Write PAIRINGS, a function that pairs the elements of two lists. (PAIRINGS '(A B C) '(1 2 3)) should return ((A 1) (B 2) (C 3)). You may assume that the two lists will be of equal length.

8.50. Write SUBLISTS, a function that returns the successive sublists of a list. (SUBLISTS '(FEE FIE FOE)) should return ((FEE FIE FOE) (FIE FOE) (FOE)).

8.51. The simplest way to write MY-REVERSE, a recursive version of REVERSE, is with a helping function plus a recursive function of two inputs. Write this version of MY-REVERSE.

8.52. Write MY-UNION, a recursive version of UNION.

8.53. Write LARGEST-EVEN, a recursive function that returns the largest even number in a list of nonnegative integers. (LARGEST-EVEN '(5 2 4 3)) should return four. (LARGEST-EVEN NIL) should return zero. Use the built-in MAX function, which returns the largest of its inputs.

8.54. Write a recursive function HUGE that raises a number to its own power. (HUGE 2) should return 2^2, (HUGE 3) should return $3^3 = 27$, (HUGE 4) should return $4^4 = 256$, and so on. Do not use REDUCE.

8.15 RECURSION IN ART AND LITERATURE

Recursion can be found not only in computer programs, but also in stories and in paintings. The classic *One Thousand and One Arabian Nights* contains stories within stories within stories, giving it a recursive flavor. A similar effect is expressed visually in some of Dr. Seuss's drawings in *The Cat in the Hat Comes Back.* One of these is shown in Figure 8-9. The nesting of cats within hats is like the nesting of contexts when a recursive function calls itself. In the story, each cat's taking off his hat plays the role of a recursive function call. Little cat B has his hat on at this point, but the recursion eventually gets all the way to Z, and terminates with an explosion. (If this story has any moral, it would appear to be, ''Know when to stop!'')

Some of the most imaginative representations of recursion and self-referentiality in art are the works of the Dutch artist M. C. Escher, whose lithograph ''Drawing Hands'' appears in Figure 8-10. Douglas Hofstadter discusses the role of recursion in music, art, and mathematics in his book *Godel, Escher, Bach: An Eternal Golden Braid.* The dragon stories in this chapter were inspired by characters in Hofstadter's book.

Figure 8-9 Recursively nested cats, from *The Cat in the Hat Comes Back*, by Dr. Suess. Copyright (c) 1958 by Dr. Suess. Reprinted by permission of Random House, Inc.

Figure 8-10 ‘‘Drawing Hands’’ by M. C. Escher. Copyright (c) 1989 M. C. Escher heirs/Cordon Art–Baarn–Holland.

SUMMARY

Recursion is a very powerful control structure, and one of the most important ideas in computer science. A function is said to be ‘‘recursive’’ if it calls itself. To write a recursive function, we must solve three problems posed by the Dragon’s three rules of recursion:

1. Know when to stop.

2. Decide how to take one step.

3. Break the journey down into that step plus a smaller journey.

We've seen a number of recursion templates in this chapter. Recursion templates capture the essence of certain stereotypical recursive solutions. They can be used for writing new functions, or for analyzing existing functions. The templates we've seen so far are:

1. Double-test tail recursion.
2. Single-test tail recursion.
3. Single-test augmenting recursion.
4. List-consing recursion.
5. Simultaneous recursion on several variables.
6. Conditional augmentation.
7. Multiple recursive calls.
8. CAR/CDR recursion.

REVIEW EXERCISES

8.55. What distinguishes a recursive function from a nonrecursive one?

8.56. Write EVERY-OTHER, a recursive function that returns every other element of a list—the first, third, fifth, and so on. (EVERY-OTHER '(A B C D E F G)) should return (A C E G). (EVERY-OTHER '(I CAME I SAW I CONQUERED)) should return (I I I).

8.57. Write LEFT-HALF, a recursive function in two parts that returns the first $n/2$ elements of a list of length n. Write your function so that the list does not have to be of even length. (LEFT-HALF '(A B C D E)) should return (A B C). (LEFT-HALF '(1 2 3 4 5 6 7 8)) should return (1 2 3 4). You may use LENGTH but not REVERSE in your definition.

8.58. Write MERGE-LISTS, a function that takes two lists of numbers, each in increasing order, as input. The function should return a list that is a merger of the elements in its inputs, in order. (MERGE-LISTS '(1 2 6 8 10 12) '(2 3 5 9 13)) should return (1 2 2 3 5 6 8 9 10 12 13).

8.59. Here is another definition of the factorial function:

```
Factorial(0) = 1
Factorial(N) = Factorial(N+1) / (N+1)
```

Verify that these equations are true. Is the definition recursive? Write a Lisp function that implements it. For which inputs will the function return the correct answer? For which inputs will it fail to return the correct answer? Which of the three rules of recursion does the definition violate?

Lisp Toolkit: The Debugger

All beginning Lispers quickly learn one debugger command, because as soon as they type something wrong, that's where they end up: in the debugger. They have to learn how to get out! Lisp implementations differ substantially when it comes to debuggers, so there will be no standard way to recover from an error. Some of you have probably been typing Q for Quit or :A for Abort, while others may be typing Control-C or Control-G. In any case, now that you're confident you can exit the debugger whenever you like, why not stay around a while?

The debugger does not actually remove bugs from programs. What it does is let you examine the state of the computation when an error has occurred. This also makes it a good tool for learning about recursion. We can use the BREAK function to enter the debugger at a strategic point in the computation. The argument to BREAK is a message, in string quotes, to be printed when the debugger is entered. Here is a modified version of FACT that demonstrates the use of BREAK:

```
(defun fact (n)
  (cond ((zerop n) (break "N is zero."))
        (t (* n (fact (- n 1))))))

> (fact 5)
N is zero.
Entering the debugger:

Debug>
```

We are now sitting in the debugger; ‘‘Debug>’’ is the debugger's prompt. (Your debugger may use a different prompt.) One of the things we can do at this point is display a **backtrace** of the control stack, which shows all the recursive calls that are currently stacked up. If you're not familiar with terms like ‘‘control stack’’ and ‘‘stack frame,’’ just play around with the debugger for a while and you'll get the hang of what's going on. (The control stack is Lisp's way of keeping track of a collection of nested function calls. A stack frame is an entry on the stack that describes one of these function calls.) In my debugger the command for displaying a backtrace is BK.

```
Debug> bk
(BREAK "N is zero.")
(FACT (- N 1))
(FACT (- N 1))
(FACT (- N 1))
(FACT (- N 1))
(FACT (- N 1))
(FACT 5)
<Bottom of Stack>
```

Variants of the BK command allow different sorts of control stack information to be displayed. In my debugger, BKFV gives a display of function names and their local variables.

```
Debug> bkfv
BREAK
  N = 0
FACT
  N = 1
FACT
  N = 2
FACT
  N = 3
FACT
  N = 4
FACT
  N = 5
FACT
<Bottom of Stack>
```

While inside the debugger we can look at the values of variables, and type arbitrary Lisp expressions using them.

```
Debug> n
0

Debug> (cons 'foo n)
(FOO . 0)
```

When we enter the debugger, we are sitting at the top of the stack. We can move around the stack using the commands called (in my debugger) UP and DOWN. If we move down the stack, we can see other local variables named N.

```
Debug> down
(FACT (- N 1))
```

```
Debug> down
(FACT (- N 1))

Debug> down
(FACT (- N 1))

Debug> bkv
(BREAK "N is zero.")
  N = 0
(FACT (- N 1))
  N = 1
(FACT (- N 1))
  N = 2
(FACT (- N 1))   <-- Current stack frame
  N = 3
(FACT (- N 1))
  N = 4
(FACT (- N 1))
  N = 5
(FACT 5)
<Bottom of Stack>

Debug> n
3
```

Finally, we can use the debugger to return from any one of the function calls currently on the stack. This causes the computation to resume as if the function had returned normally:

```
Debug> return 10
600
```

When we returned 10 from the current stack frame, the computation resumed at that point, and the value produced was $5 \times 4 \times 3 \times 10 = 600$.

Your debugger won't look exactly like mine, and it may provide somewhat different capabilities, but the basic idea of examining the control stack is common to all Lisp debuggers. Look in the user's manual for your Lisp implementation to see which debugger commands are offered. Typing HELP or :H or ‘‘?’’ to your debugger may also produce a list of commands.

Keyboard Exercise

In this exercise we will extract different sorts of information from a genealogical database. The database gives information for five generations of a family, as shown in Figure 8-11. Such diagrams are usually called family trees, but this family's genealogical history is not a simple tree structure. Marie has married her first cousin Nigel. Wanda has had one child with Vincent and another with Ivan. Zelda and Robert, the parents of Yvette, have two great grandparents in common. (This might explain why Yvette turned out so weird.) And only Tamara knows who Frederick's father is; she's not telling.

Figure 8-11 Genealogy information for five generations of a family.

```
(setf family
      '((colin nil nil)
        (deirdre nil nil)
        (arthur nil nil)
        (kate nil nil)
        (frank nil nil)
        (linda nil nil)
        (suzanne colin deirdre)
        (bruce arthur kate)
        (charles arthur kate)
        (david arthur kate)
        (ellen arthur kate)
        (george frank linda)
        (hillary frank linda)
        (andre nil nil)
        (tamara bruce suzanne)
        (vincent bruce suzanne)
        (wanda nil nil)
        (ivan george ellen)
        (julie george ellen)
        (marie george ellen)
        (nigel andre hillary)
        (frederick nil tamara)
        (zelda vincent wanda)
        (joshua ivan wanda)
        (quentin nil nil)
        (robert quentin julie)
        (olivia nigel marie)
        (peter nigel marie)
        (erica nil nil)
        (yvette robert zelda)
        (diane peter erica)))
```

Figure 8-12 The genealogy database.

Each person in the database is represented by an entry of form

(*name father mother*)

When someone's father or mother is unknown, a value of NIL is used.

The functions you write in this keyboard exercise need not be recursive, except where indicated. For functions that return lists of names, the exact order in which these names appear is unimportant, but there should be no duplicates.

EXERCISE

8.60. If the genealogy database is already stored on the computer for you, load the file containing it. If not, you will have to type it in as it appears in Figure 8-12. Store the database in the global variable FAMILY.

a. Write the functions FATHER, MOTHER, PARENTS, and CHILDREN that return a person's father, mother, a list of his or her known parents, and a list of his or her children, respectively. (FATHER 'SUZANNE) should return COLIN. (PARENTS 'SUZANNE) should return (COLIN DEIRDRE). (PARENTS 'FREDERICK) should return (TAMARA), since Frederick's father is unknown. (CHILDREN 'ARTHUR) should return the set (BRUCE CHARLES DAVID ELLEN). If any of these functions is given NIL as input, it should return NIL. This feature will be useful later when we write some recursive functions.

b. Write SIBLINGS, a function that returns a list of a person's siblings, including genetic half-siblings. (SIBLINGS 'BRUCE) should return (CHARLES DAVID ELLEN). (SIBLINGS 'ZELDA) should return (JOSHUA).

c. Write MAPUNION, an applicative operator that takes a function and a list as input, applies the function to every element of the list, and computes the union of all the results. An example is (MAPUNION #'REST '((1 A B C) (2 E C J) (3 F A B C D))), which should return the set (A B C E J F D). *Hint:* MAPUNION can be defined as a combination of two applicative operators you already know.

d. Write GRANDPARENTS, a function that returns the set of a person's grandparents. Use MAPUNION in your solution.

e. Write COUSINS, a function that returns the set of a person's genetically related first cousins, in other words, the children of any of their parents' siblings. (COUSINS 'JULIE) should return the set (TAMARA VINCENT NIGEL). Use MAPUNION in your solution.

f. Write the two-input *recursive* predicate DESCENDED-FROM that returns a true value if the first person is descended from the second. (DESCENDED-FROM 'TAMARA 'ARTHUR) should return T. (DESCENDED-FROM 'TAMARA 'LINDA) should return NIL. (*Hint:* You are descended from someone if he is one of your parents, or if either your father or mother is descended from him. This is a recursive definition.)

g. Write the *recursive* function ANCESTORS that returns a person's set of ancestors. (ANCESTORS 'MARIE) should return the set (ELLEN ARTHUR KATE GEORGE FRANK LINDA). (*Hint:* A person's ancestors are his parents plus his parents' ancestors. This is a recursive definition.)

h. Write the *recursive* function GENERATION-GAP that returns the number of generations separating a person and one of his or her ancestors. (GENERATION-GAP 'SUZANNE 'COLIN) should return one. (GENERATION-GAP 'FREDERICK 'COLIN) should return three. (GENERATION-GAP 'FREDERICK 'LINDA) should return NIL, because Linda is not an ancestor of Frederick.

i. Use the functions you have written to answer the following questions:

   1. Is Robert descended from Deirdre?
   2. Who are Yvette's ancestors?
   3. What is the generation gap between Olivia and Frank?
   4. Who are Peter's cousins?
   5. Who are Olivia's grandparents?