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
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:

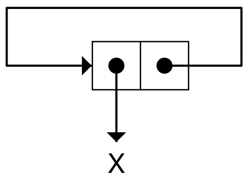
```
(defun any-7-p (x)
  (cond ((equal (first x) 7) t)
        (t (any-7-p (rest x)))))
```

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:

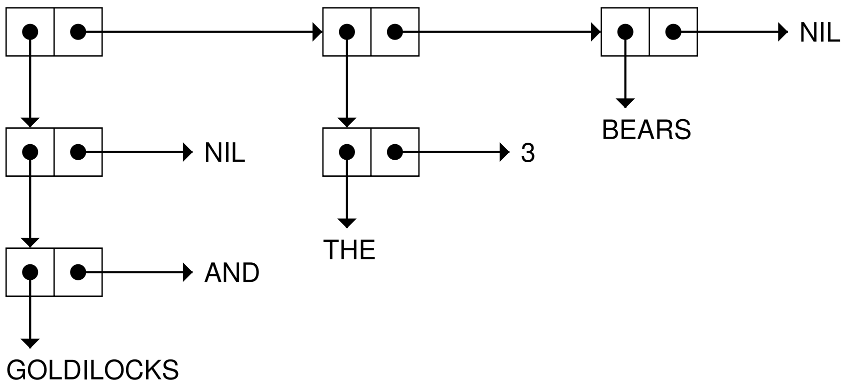
```
(DEFUN func (X)
  (COND (end-test-1 end-value-1)
        (end-test-2 end-value-2)
        (T (combiner (func (CAR X))
                     (func (CDR X))))))
```

Example:

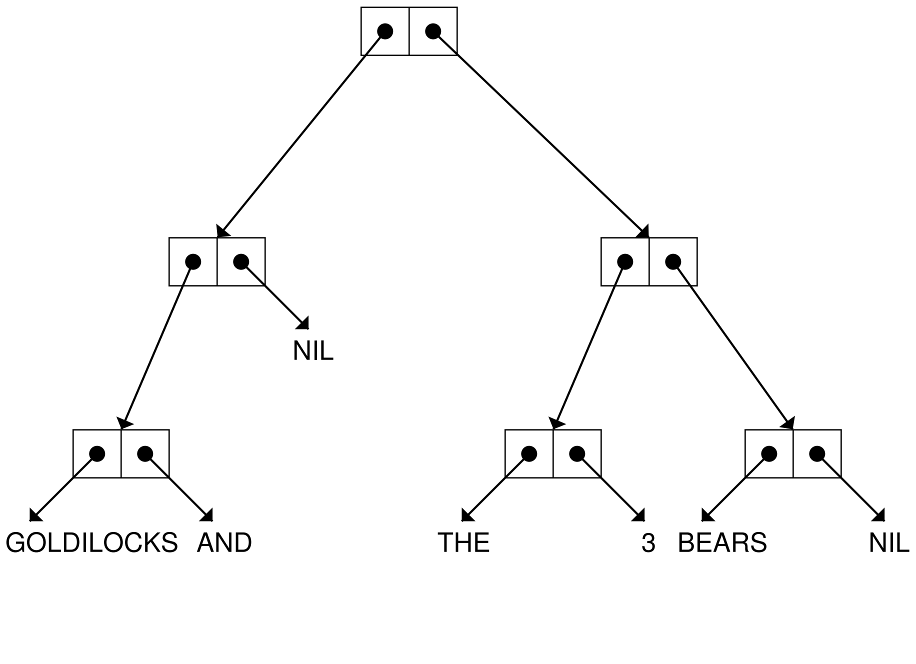
Func:	FIND-NUMBER
End-test-1:	(NUMBERP X)
End-value-1:	X
End-test-2:	(ATOM X)
End-value-2:	NIL
Combiner:	OR

```
(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?