

Lisp: Lists and Recursion

March, 1983

SINCE I ended the previous column with a timely newsbreak about the homely Glazunkian porpuquine, I felt it only fitting to start off the present column with more about that little-known but remarkable beast. As you may recall, the quills on any porpuquine (except for the tiniest ones) are smaller porpuquines. The tiniest porpuquines have no quills but do have a nose, and a very important nose at that, since the Glazunkians base their entire monetary system on that little nose. Consider, for instance, the value of 3-inch porpuquines in Outer Glazunkia. Each one always has nine quills (contrasting with their cousins in Inner Glazunkia, which always have seven); thus each one has nine 2-inch porpuquines sticking out of its body. Each of those in turn sports nine 1-inch porpuquines, out of each of which sprout nine zero-inch porpuquines, each of which has one nose. All told, this comes to  $9 \times 9 \times 9 \times 1$  noses, which means that a 3-inch porpuquine in Outer Glazunkia has a buying power of 729 noses. If, by contrast, we had been in Inner Glazunkia and had started with a 4-inch, that porpuquine would have a buying power of  $7 \times 7 \times 7 \times 7 \times 1 = 2,401$  noses.

Let's see if we can't come up with a general recipe for calculating the buying power (measured in noses) of any old porpuquine. It seems to me that it would go something like this:

The buying power of a porpuquine with a given quill count and size is: if its size = 0, then 1;

otherwise, figure out the buying power of a porpuquine with the same quill count but of the next smaller size, and multiply that by the quill count.

We can shorten this recipe by adopting some symbolic notation. First, let  $q$  stand for the quill count and  $s$  for the size. Then let  $\text{cond}$  stand for "if" and  $t$  for "otherwise". Finally, use a sort of condensed algebraic notation in which the English names of operations are placed to the left of their operands, inside parentheses. We get something like this:

(buying-power  $q$   $s$ ) Is:  $\text{cond} ((\text{eq } s \ 0) \ 1;$   
 $t \ (\text{times } q \ (\text{buying-power } q \ (\text{next-smaller } s))))$

This is an exact translation of the earlier English recipe into a slightly more symbolic form. We can make it a little more compact and symbolic by adopting a couple of new conventions. Let each of the two cases (the case where  $s$  equals zero and the "otherwise" case) be enclosed in parentheses; in general, use parentheses to enclose each logical unit completely. Finally, indicate by the words  $\text{def}$  and  $\lambda$  that this is a definition of a general notion called buying-power with two variables (quill count  $q$  and size  $s$ ). Now we get:

( $\text{def buying-power } (\lambda (q \ s) (\text{cond } ((\text{eq } s \ 0) \ 1)$   
 $(t \ (\text{times } q \ (\text{buying-power } q \ (\text{next-smaller } s))))))$ )

I mentioned above that the buying power of a 9-quill, 3-inch porpuquine is 729 noses. This could be expressed by saying that (buying-power 9 3) equals 729. Similarly, (buying-power 7 4) equals 2,401.

---

Well, so much for porpuquines. Now let's limp back to Lisp after this rather long digression. I had posed a puzzle, toward the end of last month's column, in which the object was to write a Lisp function that subsumed a whole family of related functions called square, cube, 4th-power, 5th-power, and so on. I asked you to come up with one general function called power, having two variables, such that (power 9 3) gives 729, (power 7 4) gives 2,401, and so on. I had presented a "tower of power"-that is; an infinitely tall tower of separate Lisp definitions, one for each power, connecting it to the preceding power. Thus a typical floor in this tower would be:

```
(def 102nd-power (lambda (q) (times q (101st-power q))))
```

Of course, 101st-power would refer to 100th-power in its definition, and so on, thus creating a rather long regress back to the simplest, or "embryonic", case. Incidentally, that very simplest case, rather than square or even 1st-power, is this:

```
(def 0th-power (lambda (q) 1))
```

I told you that you had all the information necessary to assemble the proper definition. All you needed to observe is, of course, that each floor of the

tower rests on the "next-smaller" floor (except for the bottom floor, which is a "stand-alone" floor). By "next-smaller", I mean the following:

```
(def next-smaller (lambda (s) (difference s 1)))
```

Thus (next-smaller 102) yields 101. Actually, Lisp has a standard name for this operation (namely, `sub1`) as well as for its inverse operation (namely, `add1`). If we put all our observations together, we come up with the following universal definition:

```
(def power (lambda (q s) (cond ((eq s 0) 1)
                                (t (times q (power q (next-smaller s)))))))
```

This is the answer to the puzzle I posed. Hmmm, that's funny ... I have the strangest sense of *deja vu*. I wonder why!

---

The definition presented here is known as a recursive definition, for the reason that inside the definiens, the definiendum is used. This is a fancy way of saying that I appear to be defining something in terms of itself, which ought to be considered gauche if not downright circular in anyone's book! To see whether the Lisp genie looks askance upon such trickery, let's ask it to figure out (power 9 3):

(power 9 3) 729

Well, fancy that! No complaints? No choking? How can the Lisp genie swallow such nonsense?

The best explanation I can give is to point out that no circularity is actually involved. While it is true that the definition of power uses the word power inside itself, the two occurrences are referring to different circumstances. In a nutshell, (power q s) is being defined in terms of a simpler case, namely, (power q (next-smaller s)). Thus I am defining the 44th power in terms of the 43rd power, and that in terms of the next- smaller power, and so on down the line until we come to the “bottom line”, as I call it-the 0th power, which needs no recursion at all. It suffices to tell the genie that its value is 1. So when you look carefully, you see that this recursive definition is no more circular than the “tower of power” was-and you can’t get any straighter than an infinite straight line! In fact, this one compact definition really is just a way of getting the whole tower of power into one finite expression. Far from being circular, it is just a handy summary of infinitely many different definitions, all belonging to one family.

In case you still have a trace of skepticism about this sleight of hand, perhaps I should let you watch what the Lisp genie will do if you ask for a “trace” of the function, and then ask it once again to evaluate (power 9 3).

-> (power 9 3).

```
ENTERING power (q=9, s=3) ENTERING power (q=9, s=2) ENTERING
power (q=9, s=1) ENTERING power (q=9, s=0) EXITING power (value: 1)
EXITING power (value: 9) EXITING power (value: 81) EXITING power (value:
729) 729 ->
```

On the lines marked ENTERING, the genie prints the values of the two arguments, and on the lines marked EXITING, it prints the value it has computed and is returning. For each ENTERING line there is of course an EXITING line, and the two are aligned vertically-that is, they have the same amount of indentation.

You can see that in order to figure out what (power 9 3) is, the genie must first calculate (power 9 2). But this is not a given; instead it requires knowing the value of (power 9 1), and this in turn requires (power 9 0). Ah! But we were given this one-it is just 1. And now we can bounce back “up”, remembering that in order to get one answer from the “deeper” answer, we must multiply by 9. Hence we get 9, then 81, then 729, and we are done.

I say “we”, but of course it is not we but the Lisp genie who must keep track of these things. The Lisp genie has to be able to suspend one computation to work on another one whose answer was requested by the first one. And the second computation, too, may request the answer to a third one, thus putting itself on hold-as may the third, and so on, recursively. But eventually, there will come a case where the buck stops that is, where a process runs to completion and returns a value-and that will enable other stacked-up processes to finally return

values, like stacked- up airplanes that have circled for hours finally getting to land, each landing opening up the way for another landing.

Ordinarily, the Lisp genie will, not print out a trace of what it is thinking unless you ask for it. However, whether you ask to see it or not, this kind of thing is going on behind the scenes whenever a function call is evaluated. One of the enjoyable things about Lisp is that it can deal with such recursive definitions without getting flustered.

---

I am not so naive as to expect that you've now totally got the hang of recursion and could go out and write huge recursive programs with the greatest of ease. Indeed, recursion can be a remarkably subtle means of defining functions, and sometimes even an expert can have trouble figuring out the meaning of a complicated recursive definition. So I thought I'd give you some practice in working with recursion. Let me give a simple example based on this silly riddle: "How do you make a pile of 13 stones?" Answer: "Put one stone on top of a pile of 12 stones." (Ask a silly question and get an answer 12/13 as silly.) Suppose we want to make a Lisp function that will give us not a pile of 13 stones, but a list consisting of 13 copies of the atom stone-or in general, n copies of that atom. We can base our answer on the riddle's silly-seeming yet correct recursive answer. The general notion is to build the answer for n out of the answer for n's predecessor. Build how? Using the list-building function cons, that's how. What's the embryonic case? That is, for which value of

- does this riddle present absolutely no problem at all? That's easy: when
- equals 0, our list should be empty, which means the answer is nil. We can now put our observations together as follows:

```
(def bunch-of-stones (lambda (n) (cond ((eq n 0) nil)
                                         (t (cons 'stone (bunch-of-stones (next-smaller n)))))))
```

Now let's watch the genie put together a very small bunch of stones (with trace on, just for fun):

```
-> (bunch-of-stones 2)
```

```
ENTERING bunch-of-stones (n=2)
```

```
ENTERING bunch-of-stones (n=1)
```

```
ENTERING bunch-of-stones (n=0) EXITING bunch-of-stones (value: nil) EXITING bunch-of-stones (value: (stone)) EXITING bunch-of-stones (value: (stone stone)) (stone stone) ->
```

This is what is called "consing up a list". Now let's try another one. This one is an old chestnut of Lisp and indeed of recursion in general. Look at the definition and see if you can figure out what it's supposed to do; then read on to see if you were right.

[illegible]

Remember, *subl* means the same as *next-smaller*. For a lark, why don't you calculate the value of (*wow* 100)? (If you ate your mental Wheaties this morning, try it in your head.)

It happens that Lisp genies often mumble out loud while they are executing wishes, and I just happen to have overheard this one as it was executing the wish (wow 100). Its soliloquy ran something like this:

Hmm ... (wow 100), eh? Well, 100 surely isn't equal to 0, so I guess the answer has to be 100 times what it would have been, had the problem been (wow 99). All rightie-now all I need to do is figure out what (wow 99) is. Oh, this is going to be a piece of cake! Let's see, is 99 equal to 0? No, seems not to be, so I guess the answer to this problem must be 99 times what the answer would have been, had the problem been (wow 98). Oh, this is going to be child's play! Let's see ...

At this point, the author, having some pressing business at the bank, had to leave the happy genie, and did not again pass the spot until some milliseconds afterwards. When he did so, the genie was just finishing up, saying:

... And now I just need to multiply that by 100, and I've got my final answer. Easy as pie! I believe it comes out to be 9332621544394415268169923856266700490715968264381621468592963895217599993229915608941463976156518286253697920827223758251185210916864000000000000000000000000-if I'm not mistaken.

Is that the answer you got, dear reader? No? Ohhh, I see where you went wrong. It was in your multiplication by 52. Go back and try it again from that point on, and be a little more careful in adding those long columns up. I'm quite sure you'll get it right this time.

This wow function is ordinarily called factorial,- n factorial is usually defined to be the product of all the numbers from 1 through n. But a recursive definition looks at things slightly differently: speaking recursively, n factorial is simply the product of n and the previous factorial. It reduces the given problem to a simpler sort of the same type. That simpler one will in turn be reduced, and so on down the line, until you come to the simplest problem of that type, which I call the “embryonic case” or the “bottom line”. People often speak, in fact, of a recursion “bottoming out”.

A New Yorker cartoon from a few years back illustrates the concept perfectly. It shows a fifty-ish man holding a photograph of himself roughly ten years earlier. In that photograph, he is likewise holding a photograph of himself, ten years earlier than that. And on it goes, until eventually it “bottoms out”—quite

literally-in a photograph of a bouncy baby boy in his birthday suit (bottom in the air). This idea of recursive photos catching you as you grow up is quite appealing. I wish my parents had thought of it!

Contrast it with the more famous Morton Salt infinite regress, in which the Morton Salt girl holds a box of Morton Salt with her picture on it-but as the girl in the picture is no younger, there is no bottom line and the regress is endless, at least theoretically. Incidentally, the Dutch cocoa called “Droste’s” has a similar illustration on its boxes, and very likely so do some other products.

The recursive approach works when you have a family of related problems, at least one of which is so simple that it can be answered immediately. This I call the embryonic case. (In the factorial example, that’s the (eq  $n = 0$ ) case, whose answer is 1.) Each problem (“What is 100 factorial?”, for instance) can be viewed as a particular case of one general problem (“How do you calculate factorials?”). Recursion takes advantage of the fact that the answers to various cases are related in some logical way to each other. (For example, I could very easily tell you the value of 100 factorial if only somebody would hand me the value of 99 factorial-all I need to do is multiply by 100.) You could say that the “Recursioneer’s Motto” is: “Gee, I could solve this case if only someone would magically hand me the answer to the case that’s one step closer to the embryonic case.” Of course, this motto presumes that certain cases are, in some sense, “nearer” to the embryonic case than others are-in fact, it presumes that there is a natural pathway leading from any case through simpler cases all the way down to the embryonic case, a pathway whose steps are clearly marked all along the way.

As it turns out, this is a very reasonable assumption to make in all sorts of circumstances. To spell out the exact nature of this recursion-guiding pathway, you have to answer two Big Questions:

- (1) What is the embryonic case?
- (2) What is the relationship of a typical case to the next simpler case?

Now actually, both of these Big Questions break up into two subquestions (as befits any self-respecting recursive question!), one concerning how you recognize where you are or how to move, the other concerning what the answer is at any given stage. Thus, spelled out more explicitly, our Big Questions are:

- (1a) How can you know when you’ve reached the embryonic case? (1b) What is the embryonic answer?
- (2a) From a typical case, how do you take exactly one step toward the embryonic case?
- (2b) How do you build this case’s answer out of the “magically given” answer to the simpler case?

’ Question (2a) concerns the nature of the descent toward the embryonic case,

or bottom line. Question (2b) concerns the inverse aspect, namely, the ascent that carries you back up from the bottom to the top level. In the case of the factorial, the answers to the Big Questions are:

(1a) The embryonic case occurs when the argument is 0. (1b) The embryonic answer is 1.

(2a) Subtract 1 from the present argument. (2b) Multiply the “magic” answer by the present argument. Notice how the answers to these four questions are all neatly incorporated in the recursive definition of wow.

---

Recursion relies on the assumption that sooner or later you will bottom out. One way to be sure you’ll bottom out is to have all the simplifying or “descending” steps move in the same direction at the same rate, so that your pathway is quite obviously linear. For instance, it’s obvious that by subtracting 1 over and over again, you will eventually reach 0, provided you started with a positive integer. Likewise, it’s obvious that by performing the list-shortening operation of `cdr`, you will eventually reach `nil`, provided you started with a finite list. For this reason, recursions using `SUM` or `cdr` to define their pathway of descent toward the bottom are commonplace. I’ll show a `cdr`-based recursion shortly, but first I want to show a funny numerical recursion in which the pathway toward the embryonic case is anything but linear and smooth. In fact, it is so much like a twisty mountain road that to describe it as moving “towards the embryonic case” seems hardly accurate. And yet, just as mountain roads, no matter how many hairpin turns they make, eventually do hit their destinations, so does this path.

Consider the famous “ $3n - 1$ ” problem, in which you start with any positive integer, and if it is even, you halve it; otherwise, you multiply it by 3 and add 1. Let’s call the result of this operation on `n` (`hotpo n`) (standing for “half or triple plus one”). Here is a Lisp definition of `hotpo`:

```
(def hotpo (lambda (n)
  (cond ((even n) (half n))
        (t (addl (times 3 n))))))
```

This definition presumes that two other functions either have been or will be defined elsewhere for the Lisp genie, namely `even` and `half` (`addl` and `times` being, as mentioned earlier, intrinsic parts of Lisp). Here are the lacking definitions:

```
(def even (lambda (n) (eq (remainder n 2) 0))) (def half (lambda (n) (quotient n 2)))
```

What do you think happens if you begin with some integer and perform `hotpo` over and over again? Take 7, for instance, as your starting point. Before you do the arithmetic, take a guess as to what sort of behavior might occur.

As it turns out, the pathway followed is often surprisingly chaotic and bumpy. For instance, if we begin with 7, the process leads us to 22, then 11, then 34,

17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, 4, 2, 1.... Note that we have wound up in a short loop (a 3-cycle, in the terminology of Chapter 16). Suppose we therefore agree that if we ever reach 1, we have “hit bottom” and may stop. You might well ask, “Who says we will hit 1? Is there a guarantee?” (Again in the terminology of Chapter 16, we could ask, “Is the 1-4-2-1 cycle an attractor?”) Indeed, before you try it out in a number of cases, you have no particular reason to suspect that you will ever hit 1, let alone always. (It would be very surprising if someone correctly anticipated what would happen in the case of, say, 7 before trying it out.) However, numerical experimentation reveals a remarkable reliability to the process; it seems that no matter where you start, you always do enter the 1-4-2-1 cycle sooner or later. (Try starting with 27 as seed if you want a real roller-coaster ride!)

Can you write a recursive function to reveal the pathway followed from an arbitrary starting point “down” to 1? Note that I say “down” advisedly, since many of the steps are in fact up! Thus the pathway starting at 3 would be the list (3 10 5 16 8 4 2 1). In order to solve this puzzle, you need to go back and answer for yourself the two Big Questions of Recursion, as they apply here. Note:

```
(cond ((not (want help)) (not (read further))) (t (read further)))
```

---

First-about the embryonic case. This is easy. It has already been defined as the arrival at 1; and the embryonic, or simplest possible, answer is the list (1), a tiny but valid pathway from 1 to 1 .

Second-about the more typical cases. What operation will carry us from typical 7 one step closer to embryonic 1? Certainly not the subl operation. No-by definition it's the function hotpo itself that brings you ever “nearer” to 1-even when it carries you up ! This teasing quality is of course the whole point of the example. What about (2b)-how to recursively build a list documenting our wildly oscillating pathway? Well, the pathway belonging to 7 is gotten by tacking (i.e., consing) 7 onto the shorter pathway belonging to (hotpo 7), or 22. After all, 22 is one step closer to being embryonic than 7 is!

These answers enable us to write down the desired function definition, using tato as our dummy variable (tato being a well-known acronym for

tato (and tato only), which recursively expands to tato (and tato only) (and tato (and tato only) only)-and so forth).

```
(def pathway-to-1 (lambda (tato) (cond ((eq tato 1) '(1))
(t (cons tato (pathway-to-1 (hotpo tato)))))))
```

Look at the way the Lisp genie “thinks” (as revealed when the trace feature is on):

```
-> (pathway-to-i 3)
```



ENTERING pathway-to-i (tato=3) ENTERING pathway-to-1 (tato=10) ENTERING pathway-to-1 (tato=5) ENTERING pathway-to-1 (tato=16) ENTERING pathway-to-1 (tato=8) ENTERING pathway-to-i (tato=4) ENTERING pathway-tai (tato=2) ENTERING pathway-to-1 (tato=1) EXITING pathway-to-1 (value: (1)) EXITING pathway-to-1 (value: (2 1)) EXITING pathway-to-i (value: (4 2 1)) EXITING pathway-to-1 (value: (8 4 2 1)) EXITING pathway-to-1 (value: (16 8 4 2 1)) EXITING pathway-to-1 (value: (5 16 8 4 2 1)) EXITING pathway-to-1 (value: (10 5 16 8 4 2 1)) EXITING pathway-to-1 (value: (3 10 5 16 8 4 2 1)) (3 10 5 16 8 4 2 1) ->

Notice the total regularity (the sideways V V shape) of the left margin of the trace diagram, despite the chaos of the numbers involved. Not all recursions are so geometrically pretty, when traced. This is because some problems request more than one subproblem to be solved. As a practical real-life example of such a problem, consider how you might go about counting up all the unicorns in Europe. This is certainly a nontrivial undertaking, yet there is an elegant recursive answer: Count up all the unicorns in Portugal, also count up all the unicorns in the other 30-odd countries of Europe, and finally add those two results together.

Notice how this spawns two smaller unicorn-counting subproblems, which in turn will spawn two subproblems each, and so on. Thus, how can one count all the unicorns in Portugal? Easy: Add the number of unicorns in the Estremadura region to the number of unicorns in the rest of Portugal! And how do you count up the unicorns in Estremadura (not to mention those in the remaining regions of Portugal)? By further breakup, of course.

But what is the bottom line? Well, regions can be broken up into districts, districts into square kilometers, square kilometers into hectares, hectares into square meters-and presumably we can handle each square meter without further breakup.

Although this may sound rather arduous, there really is no other way to conduct a thorough census than to traverse every single part on every level of the full structure that you have, no matter how giant it may be. There is a perfect Lisp counterpart to this unicorn census: it is the problem of determining how many atoms there are inside an arbitrary list. How can we write a Lisp function called atomcount that will give us the answer 15 when it is shown the following strange-looking list (which we'll call brahma)?

((ac ab cb) ac (ba be ac)) ab ((cb ca ba) cb (ac ab cb)))

One method, expressed recursively, is exactly parallel to that for ascertaining the unicorn population of Europe. See if you can come up with it on your own.

---

The idea is this. We want to construct the answer-namely, 15-out of the answers to simpler atom-counting problems. Well, it is obvious that one simpler

atom-counting problem than (atomcount brahma) is (atomcount (car brahma)). Another one is (atomcount (cdr brahma)). The answers to these two problems are, respectively, 7 and 8. Now clearly, 15 is made out of 7 and 8 by addition—which makes sense, after all, since the total number of atoms must be the number in the car plus the number in the cdr. There’s nowhere else for any atoms to hide! Well, this analysis gives us the following recursive definition, with s as the dummy variable:

```
(def atomcount (lambda (s)
  (plus (atomcount (car s)) (atomcount (cdr s))))))
```

It looks very simple, but it has a couple of flaws. First, we have written the recursive part of the definition, but we have utterly forgotten the other equally vital half—the “bottom line”. It reminds me of the Maryland judge I once read about in the paper, who ruled: “A horse is a four-legged animal that is produced by two other horses.” This is a lovely definition, but where does it bottom out? Similarly for atomcount. What is the simplest case, the embryonic case, of atomcount? Simple: It is when we are asked to count the atoms in a single atom. The answer, in such a case, is of course 1. But how can we know when we are looking at an atom? Fortunately, Lisp has a built-in function called atom that returns t (meaning “true”) whenever we are looking at an atom, and nil otherwise. Thus (atom ‘plop) returns t, while (atom’(a b c)) returns nil. Using that, we can patch up our definition:

```
(def atomcount (lambda (s) (cond ((atom s) 1)
  (t (plus (atomcount (car s)) (atomcount (cdr s)))))))
```

Still, though, it is not quite right. If we ask the genie for atomcount of (a b c), instead of getting 3 for an answer, we will get 4. Shocking! How come this happens? Well, we can pin the problem down by trying an even simpler example: if we ask for (atomcount ‘(a)) we find we get 2 instead of 1. Now the error should be clearer:  $2 = 1 + 1$ , with 1 each coming from the car and cdr of (a). The car is the atom a which indeed should be counted as 1, but the cdr is nil, which should not. So why does nil give an atomcount of 1? Because nil is not only an empty list, it is also an atom! To suppress this bad effect, we simply insert another cond clause at the very top:

```
(def atomcount (lambda (s) (cond ((null s) 0) ((atom s) 1)
  (t (plus (atomcount (car s)) (atomcount (cdr s)))))))
```

I wrote (null s), which is just another way of saying (eq s nil). In general, if you want to determine whether the value of some expression is nil or not, you can use the in-built function null, which returns t if yes, nil if no. Thus, for example, (null (null nil)) evaluates to nil, since the inner function call evaluates to t, and t is not nil!

Notice in this recursion that we have more than one type of embryonic case (the null case and the atom case), and more than one way of descending toward the

embryonic case (via both car and cdr). Thus, our Big Questions can be revised a bit further:

- (1a) Is there just one embryonic case, or are there several, or even an infinite class of them? (1b) How can you know when you’ve reached an embryonic case? (1c) What are the answers to the various embryonic cases?
- (2a) From a typical case, is there exactly one way to step toward an embryonic case, or are there various possibilities? (2b) From a typical case, how do you determine which of the various routes toward an embryonic case to take? (2c) How do you build this case’s answer out of the “magically given” answers to one or more simpler cases?

Now what happens when we trace our function as it counts the atoms in brahma, our original target? The result is shown in Figure 18-1. Notice the more complicated topography of this recursion, with its many ins and outs.

```
ENTERING alomcount ls>((itA cb) ac (M ie ac|W ENTERING alomeouni
(s=(ac id PhW ENTERING atonicoUM (s=ac) EXITING atomcountl (value: 1)
ENTERING alwncotwt (s = (ih cfc» ENTERING alomeouni (i=b) EXITING
atunvcouM (value; 1) ENTERING atuKcouM <=.( <• » ENTT111ING itemen-
uol (s=cb> EXITING alomfouul (value: 1) ENTERING atOmenua.1 (s=null)
EXITING alomeount {vata« 0) F_xmNG alomeouni (value: 1) EXITING atom-
couHt (value: 2) EXITING alomcoum (value: 3) ENTERING alomcount (i=(«c
(b» be ae))) ENTERING atomcount li=ac) EXITING acomtouut (value: 1)
ENTERING alomcount (s = (Oa be ac)» F.V1 EKing atomcount Is»(ta be
«c)) ENTEHING alhmcountl (s»l»> EX111INC atomosnl (value: ENTERING
alokaounl (s=(be ac)) ENTtRINC alomcount ENTERING alomeouni k=Bb)
EXI I ING alomeouni (value: 1)
```

```
ENTERING ato«itlwil (i=(c. c« ta) Hi (ae «!«: 1) ENTERING itomcnum
(s~l(oc ab CWfl ENTERING atomcount (s = (« lb cb» EN I EH ING alomeouni
(s=ae) EXITING alomeouni (value: 1) ENTERING atomcount =ull) EXITING
atomcountl (value: 0) I X 1 1 ING aiomtount lvalue: 1) EXITING atomeouni
(value: Z) EXITING atomeouni (vabte: 3) ENT ERING atomcount b = nil)
EXITING alomeouni (value: B) EXI1 ING itomeounl (vahc; 3) EXITING
alomcount (value; t) EXITING atomcount (value: 1) ENTFRING alomcount
(i=null) EXITING atomeouni [value: J) EXITING atomeouni (value: 7) EXITING
alomeouni (value; 8) EXITING a Rrmcountl lvalue: 15)
```

Whereas the previous ‘V’-shaped recursion looked like a simple descent into a smooth-walled canyon and then a simple climb back up the other side, this recursion looks like a descent into a much craggier canyon, where on your way up and down each wall you encounter various “subcanyons” that you must treat in the same way- and who knows how many levels of such structure you will be called on to deal with

in your exploration? Shapes with substructure that goes on indefinitely like that, never bottoming out in ordinary curves, are called fractals. Their nature forms an important area of inquiry in mathematics today. An excellent introduction can be found in Martin Gardner's "Mathematical Games" for April, 1978, and a much fuller treatment in Benoit Mandelbrot's splendid book *The Fractal Geometry of Nature*. For a dynamic view of a few historically revolutionary fractals, there is Nelson Max's marvelous film *Space-Filling Curves*, where so-called "pathological" shapes are constructed step by step before your eyes, and their mathematical significance is geometrically presented. Then, as eerie electronic music echoes all about, you start shrinking like Alice in Wonderland-but unlike her, you can't stop, and as you shrink towards oblivion, you get to see ever more microscopic views of the infinitely detailed fractal structures. It's a great visual adventure, if you're willing to experience infinity-vertigo!

---

One of the most elegant recursions I know of originates with the famous disk-moving puzzle known variously as "Lucas' Tower", the "Tower of Hanoi", and the "Tower of Brahma". Apparently it was originated by the French mathematician Edouard Lucas in the nineteenth century. The legend that is popularly attached to the puzzle goes like this:

In the great Temple of Brahma in Benares, on a brass plate beneath the dome that marks the Center of the World, there are 64 disks of pure gold which the priests carry one at a time between three diamond needles according to Brahma's immutable law: No disk may be placed on a smaller disk. In the Beginning of the World, all 64 disks formed the Tower of Brahma on one needle. Now, however, the process of transfer of the tower from one needle to another is in midcourse. When the last disk is finally in place, once again forming the Tower of Brahma but on a different needle, then will come the End

of the World, and All will turn to dust.

A picture of the puzzle is shown in Figure 18-2. In it, the three needles are labeled a, b, and c.

If you work at it, you certainly can discover the systematic method that the priests must follow in order to get the disks from needle a to needle b. For only three disks, for instance, it is very easy to write down the order in which the moves go: ab ac be ab ca cb ab

FIGURE 18-2. The Tower of Brahma puzzle, with 64 disks to be transferred. [Drawing by David Moser. ]

Here, the Lisp atom ab represents a jump from needle a to needle b. There is a structure to what is going on, however, that is not revealed by a mere listing of such atoms. It is better revealed if one groups the atoms as follows:

ab ac be ab ca cb ab

The first group accomplishes a transfer of a 2-tower from needle a to needle c, thus freeing up the largest disk. Then the middle move, ab, picks up that big heavy disk and carries it over from needle a to needle b. The final group is very much like the initial group, in that it transfers the 2-tower back from needle c to needle b. Thus the solution to moving three depends on being able to move two. Similarly, the solution to moving 64 depends on being able to move 63. Enough said? Now try to write a Lisp function that will give you a solution to the Tower of Brahma for n disks. (You may prefer to label the three needles with digits rather than letters, so that moves are represented by two-digit numbers such as 12.) I will present the solution in the next column-unless, of course, the dedicated priests, working by day and by night to bring about the end of the world, should chance to reach their cherished goal before then ...