METAMAGICAL THEMAS:

Questing for the Essence of Mind and Pattern

DOUGLAS R. HOFSTADTER

Basic Books, Inc., Publishers New York

~ATETA~
MAGICAL
THAMAR

By Douglas R.
Hofstadter

17

Lisp: Atoms and Lists

February, 1983

In previous columns I have written quite often about the field of artificial intelligence—the search for ways to program computers so that they might come to behave with flexibility, common sense, insight, creativity, self-awareness, humor, and so on. The quest for AI started in earnest over two decades ago, and since then has bifurcated many times, so that today it is a very active and multifaceted research area. In the United States there are perhaps a couple of thousand people professionally involved in AI, and there are a similar number abroad. Although there is among these workers a considerable divergence of opinion concerning the best route to AI, one thing that is nearly unanimous is the choice of programming language. Most AI research efforts are carried out in a language called "Lisp". (The name is not quite an acronym; it stands for "list processing".)

Why is most AI work done in Lisp? There are many reasons, most of which are somewhat technical, but one of the best is quite simple: Lisp is crisp. Or as Marilyn Monroe said in *The Seven-Year Itch,* "I think it's jus-telegant!" Every computer language has arbitrary features, and most languages are in fact overloaded with them. A few, however, such as Lisp and Algol, are built around a kernel that seems as natural as a branch of mathematics. The kernel of Lisp has a crystalline purity that not only appeals to the esthetic sense, but also makes Lisp a far more flexible language than most others. Because of Lisp's beauty and centrality in this important area of modern science, then, I have decided to devote a trio of columns to some of the basic ideas of Lisp.

The deep roots of Lisp lie principally in mathematical logic. Mathematical pioneers such as Thoralf Skolem, Kurt Gödel, and Alonzo Church contributed seminal ideas to logic in the 1920's and 1930's that were incorporated decades later into Lisp. Computer programming in earnest began in the 1940's, but so-called "higher-level" programming languages (of which Lisp is one) came into existence only in the 1950's. The earliest list-processing language was not Lisp but IPL ("Information Processing Language"), developed in the mid-1950's by Herbert Simon, Allen Newell, and J. C. Shaw. In the years 1956–58, John McCarthy, drawing on all these

previous sources, came up with an elegant algebraic list-processing language he called Lisp. It caught on quickly with the young crowd around him at the newly-formed MIT Artificial Intelligence Project, was implemented on the IBM 704, spread to other AI groups, infected them, and has stayed around all these years. Many dialects now exist, but all of them share that central elegant kernel.

* * *

Let us now move on to the way Lisp really works. One of the most appealing features of Lisp is that it is *interactive,* as contrasted with most other higher-level languages, which are noninteractive. What this means is the following. When you want to program in Lisp, you sit down at a terminal connected to a computer and you type the word "lisp" (or words to that effect). The next thing you will see on your screen is a so-called "prompt" —a characteristic symbol such as an arrow or asterisk. I like to think of this prompt as a greeting spoken by a special "Lisp genie", bowing low and saying to you, "Your wish is my command—and now, what is your next wish?" The genie then waits for you to type something to it. This genie is usually referred to as the *Lisp interpreter,* and it will do anything you want—but you have to take great care in expressing your desires precisely, otherwise you may reap some disastrous effects. Shown below is the prompt, the sign that the Lisp genie is ready to do your bidding:

```
→
```

The genie is asking us for our heart's desire, so let us type in a simple expression:

```
→ (plus 2 2)
```

and then a carriage return. (By the way, all Lisp expressions and words will be printed in **Helvetica** in this and the following two chapters.) Even non-Lispers can probably anticipate that the Lisp genie will print in return the value **4**. Then it will also print a fresh prompt, so that the screen will now appear this way:

```
→ (plus 2 2)
4
→
```

The genie is now ready to carry out our next command—or, more politely stated, our next wish—should we have one. The carrying-out of a wish expressed as a Lisp statement is called *evaluation* of that statement. The preceding short interchange between human and computer exemplifies the

behavior of the Lisp interpreter: it *reads* a statement, *evaluates* it, *prints* the appropriate value, and then signals its readiness to read a new statement. For this reason, the central activity of the Lisp interpreter is referred to as the *read-eval-print loop.*

The existence of this Lisp genie (the Lisp interpreter) is what makes Lisp interactive. You get immediate feedback as soon as you have typed a "wish" —a complete statement—to Lisp. And the way to get a bunch of wishes carried out is to type one, then ask the genie to carry it out, then type another, ask the genie again, and so on.

By contrast, in many higher-level computer languages you must write out an entire program consisting of a vast number of wishes to be carried out in some specified order. What's worse is that later wishes usually depend strongly on the consequences of earlier wishes—and of course, you don't get to try them out one by one. The execution of such a program may, needless to say, lead to many unexpected results, because so many wishes have to mesh perfectly together. If you've made the slightest conceptual error in designing your wish list, then a total foul-up is likely—in fact, almost inevitable. Running a program of this sort is like launching a new space probe, untested: you can't possibly have anticipated all the things that might go wrong, and so all you can do is sit back and watch, hoping that it will work. If it fails, you go back and correct the one thing the failure revealed, and then try another launch. Such a gawky, indirect, expensive way of programming is in marked contrast to the direct, interactive, one-wish-at-a-time style of Lisp, which allows "incremental" program development and debugging. This is another major reason for the popularity of Lisp.

* * *

What sorts of wishes can you type to the Lisp genie for evaluation, and what sorts of things will it print back to you? Well, to begin with, you can type arithmetical expressions expressed in a rather strange way, such as **(times (plus 6 3) (difference 6 3))**. The answer to this is 27, since **(plus 6 3)** evaluates to 9, and **(difference 6 3)** evaluates to 3, and their product is 27. This notation, in which each operation is placed to the left of its operands, was invented by the Polish logician Jan Łukasiewicz before computers existed. Unfortunately for Łukasiewicz, his name was too formidable-looking for most speakers of English, and so this type of notation came to be called *Polish notation.* Here is a simple problem in this notation for you, in which you are to play the part of the Lisp genie:

→ **(quotient (plus 21 13) (difference 23 (times 2 (difference 7 (plus 2 2)))))**

Perhaps you have noticed that statements of Lisp involve parentheses. A profusion of parentheses is one of the hallmarks of Lisp. It is not uncommon to see an expression that terminates in a dozen right parentheses! This

makes many people shudder at first—and yet once you get used to their characteristic appearance, Lisp expressions become remarkably intuitive, even charming, to the eye, especially when *pretty-printed,* which means that a careful indentation scheme is followed that reveals their logical structure. All of the expressions in displays in this article have been pretty-printed.

The heart of Lisp is its manipulable structures. All programs in Lisp work by creating, modifying, and destroying structures. Structures come in two types: atomic and composite, or, as they are usually called, *atoms* and *lists.* Thus, every Lisp object is either an atom or a list (but not both). The only exception is the special object called **nil**, which is both an atom and a list. More about **nil** in a moment. What are some other typical Lisp atoms? Here are a few:

hydrogen, helium, j-s-bach, 1729, 3.14159, pi,
arf, foo, bar, baz, buttons-&-bows

Lists are the flexible data structures of Lisp. A list is pretty much what it sounds like: a collection of some parts in a specific order. The parts of a list are usually called its *elements* or *members.* What can these members be? Well, not surprisingly, lists can have atoms as members. But just as easily, lists can contain lists as members, and those lists can in turn contain other lists as members, and so on, recursively. Oops! I jumped the gun with that word. But no harm done. You certainly understood what I meant, and it will prepare you for a more technical definition of the term to come later.

A list printed on your screen is recognizable by its parentheses. In Lisp, anything bounded by matching parentheses constitutes a list. So, for instance, **(zonk blee strill (cronk flonk))** is a four-element list whose last element is itself a two-element list. Another short list is **(plus 2 2)**, illustrating the fact that *Lisp statements themselves are lists.* This is important because it means that the Lisp genie, by manipulating lists and atoms, can actually construct new wishes by itself. Thus the object of a wish can be the construction—and subsequent evaluation—of a new wish!

Then there is the *empty* list—the list with no elements at all. How is this written down? You might think that an empty pair of parentheses—**()**—would work. Indeed, it will work—but there is a second way of indicating the empty list, and that is by writing **nil**. The two notations are synonymous, although **nil** is more commonly written than **()** is. The empty list, **nil**, is a key concept of Lisp; in the universe of lists, it is what zero is in the universe of numbers. To use another metaphor for nil, it is like the earth in which all structures are rooted. But for you to understand what this means, you will have to wait a bit.

* * *

The most commonly exploited feature of an atom is that it has (or can be given) a *value.* Some atoms have permanent values, while others are variables. As you might expect, the value of the atom **1729** is the integer

1729, and this is permanent. (I am distinguishing here between the atom whose *print name* or *pname* is the four-digit string **1729**, and the eternal Platonic essence that happens to be the sum of two cubes in two different ways—*i.e.*, the number 1729.) The value of **nil** is also permanent, and it is —**nil**! Only one other atom has itself as its permanent value, and that is the special atom **t**.

Aside from **t**, **nil**, and atoms whose names are numerals, atoms are generally variables, which means that you can assign values to them and later change their values at will. How is this done? Well, to assign the value 4 to the atom **pie**, you can type to the Lisp genie **(setq pie 4)**. Or you could just as well type **(setq pie (plus 2 2))**—or even **(setq pie (plus 1 1 1 1))**. In any of these cases, as soon as you type your carriage return, **pie**'s value will become 4, and so it will remain forevermore—or at least until you do another **setq** operation on the atom **pie**.

Lisp would not be crisp if the only values atoms could have were numbers. Fortunately, however, an atom's value can be set to any kind of Lisp object —any atom or list whatsoever. For instance, we might want to make the value of the atom **pi** be a list such as **(a b c)** or perhaps **(plus 2 2)** instead of the number 4. To do the latter, we again use the **setq** operation. To illustrate, here follows a brief conversation with the genie:

```
→ (setq pie (plus 2 2))
4
→ (setq pi '(plus 2 2))
(plus 2 2)
→
```

Notice the vast difference between the values assigned to the atoms **pie** and **pi** as a result of these two wishes asked of the Lisp genie, which differ merely in the presence or absence of a small but critical *quote mark* in front of the inner list **(plus 2 2)**. In the first wish, containing no quote mark, that inner **(plus 2 2)** must be *evaluated.* This returns 4, which is assigned to the variable **pie** as its new value. On the other hand, in the second wish, since the quote mark is there, the list **(plus 2 2)** is never executed as a command, but is treated merely as an inert lump of Lispstuff, much like meat on a butcher's shelf. It is ever so close to being "alive", yet it is dead. So the value of **pi** in this second case is the list **(plus 2 2)**, a fragment of Lisp code. The following interchange with the genie confirms the values of these atoms.

```
→ pie
4
→ pi
(plus 2 2)
→ (eval pi)
4
→
```


What is this last step? I wanted to show how you can ask the genie to *evaluate* the value of an expression, rather than simply *printing* the value of that expression. Ordinarily, the genie automatically performs just *one* level of evaluation, but by writing **eval**, you can get a second stage of evaluation carried out. (And of course, by using **eval** over and over again, you can carry this as far as you like.) This feature often proves invaluable, but it is a little too advanced to discuss further at this stage.

* * *

Every list but **nil** has at least one element. This first element is called the list's **car**. Thus the **car** of **(eval pi)** is the atom **eval**. The cars of the lists **(plus 2 2)**, **(setq x 17)**, **(eval pi)**, and **(car pi)** are all names of operations, or, as they are more commonly called in Lisp, *functions.* The **car** of a list need not be the name of a function; it need not even be an atom. For instance, **((1) (2 2) (3 3 3))** is a perfectly fine list. Its **car** is the list **(1)**, whose **car** in turn is not a function name but merely a numeral.

If you were to remove a list's **car**, what would remain? A shorter list. This is called the list's **cdr**, a word that sounds about halfway between "kidder" and "could 'er". (The words "car" and "cdr" are quaint relics from the first implementation of Lisp on the IBM 704. The letters in "car" stand for "Contents of the Address part of Register" and those in "cdr" for "Contents of the Decrement part of Register", referring to specific hardware features of that machine, now long since irrelevant.) The **cdr** of **(a b c d)** is the list **(b c d)**, whose **cdr** is **(c d)**, whose **cdr** is **(d)**, whose **cdr** is **nil.** And **nil** has no **cdr**, just as it has no **car**. Attempting to take the **car** or **cdr** of **nil** causes (or should cause) the Lisp genie to cough out an error message, just as attempting to divide by zero should evoke an error message.

Here is a little table showing the **car** and **cdr** of a few lists, just to make sure the notions are unambiguous.

list	**car**	**cdr**
((a) b (c))	**(a)**	**(b (c))**
(plus 2 2)	**plus**	**(2 2)**
((car x) (car y))	**(car x)**	**((car y))**
(nil nil nil nil)	**nil**	**(nil nil nil)**
(nil)	**nil**	**nil**
nil	***ERROR***	***ERROR***

Just as **car** and **cdr** are called *functions,* so the things that they operate on are called their *arguments.* Thus in the command **(plus pie 2)**, **plus** is the function name, and the arguments are the atoms **pie** and **2**. In evaluating this command (and most commands), the genie figures out the values of the arguments, and then applies the function to those values. Thus, since the

value of the atom **pie** is 4, and the value of the atom **2** is 2, the genie returns the atom **6**.

* * *

Suppose you have a list and you'd like to see a list just like it, only one element longer. For instance, suppose the value of the atom **x** is **(cake cookie)** and you'd like to create a new list called **y** just like **x**, except with an extra atom—say **pie**—at the front. You can then use the function called **cons** (short for "construct"), whose effect is to make a new list out of an old list and a suggested **car**. Here's a transcript of such a process:

```
→ (setq x '(cake cookie))
(cake cookie)
→ (setq y (cons 'pie x))
(pie cake cookie)
→ x
(cake cookie)
```

Two things are worth noticing here. I asked for the value of **x** to be printed out after the **cons** operation, so you could see that **x** itself was not changed by the **cons**. The **cons** operation created a new list and made that list be the value of **y**, but left **x** entirely alone. The other noteworthy fact is that I used that quote mark again, in front of the atom **pie**. What if I had not used it? Here's what would have happened.

```
→ (setq z (cons pie x))
(4 cake cookie)
```

Remember, after all, that the atom **pie** still has the value 4, and whenever the genie sees an unquoted atom inside a wish, it will always use the *value* belonging to that atom, rather than the atom's *name*. (Always? Well, almost always. I'll explain in a moment. In the meantime, look for an exception—you've already encountered it.)

Now here are a few exercises—some a bit tricky—for you. Watch out for the quote marks! Oh, one last thing: I use the function **reverse**, which produces a list just like its argument, only with its elements in reverse order. For instance, the genie, upon being told **(reverse '((a b) (c d e)))** will write **((c d e) (a b))**. The genie's lines in this dialogue are given afterward.

```
→ (setq w (cons pie '(cdr z)))
→ (setq v (cons 'pie (cdr z)))
→ (setq u (reverse v))
→ (cdr (cdr u))
→ (car (cdr u))
→ (cons (car (cdr u)) u)
→ u
```



```
→ (reverse '(cons (car u) (reverse (cdr u))))
→ (reverse (cons (car u) (reverse (cdr u))))
→ u
→ (cons 'cookie (cons 'cake (cons 'pie nil)))
```

Answers (as printed by the genie):

```
(4 cdr z)
(pie cake cookie)
(cookie cake pie)
(pie)
cake
(cake cookie cake pie)
(cookie cake pie)
((reverse (cdr u)) (car u) cons)
(cake pie cookie)
(cookie cake pie)
(cookie cake pie)
```

The last example, featuring repeated use of **cons**, is often called, in Lisp slang, "consing up a list". You start with **nil**, and then do repeated **cons** operations. It is analogous to building a positive integer by starting at zero and then performing the successor operation over and over again. However, whereas at any stage in the latter process there is a unique way of performing the successor operation, given any list there are infinitely many different items you can **cons** onto it, thus giving rise to a vast branching tree of lists instead of the unbranching number line. It is on account of this image of a tree growing out of the ground of **nil** and containing all possible lists that I earlier likened **nil** to "the earth in which all structures are rooted".

As I mentioned a moment ago, the genie doesn't *always* replace (unquoted) atoms by their values. There are cases where a function treats its arguments, though unquoted, as if quoted. Did you go back and find such a case? It's easy. The answer is the function **setq**. In particular, in a **setq** command, the first atom is taken straight—not evaluated. As a matter of fact, the **q** in **setq** stands for "quote", meaning that the first argument is treated as if quoted. Things can get quite tricky when you learn about **set**, a function similar to **setq** except that it *does* evaluate its first argument. Thus, if the value of the atom **x** is the atom **k**, then saying **(set x 7)** will not do anything to **x**—its value will remain the atom **k**—but the value of the atom **k** will now become 7. So watch closely:

```
→ (setq a 'b)
→ (setq b 'c)
→ (setq c 'a)
→ (set a c)
→ (set c b)
```

Such behavior is to be contrasted with that of functions that leave "side effects" in their wake. Such side effects are usually in the form of changed variable bindings, although there are other possibilities, such as causing input or output to take place. A typical "harmful" command is a **setq**, and proponents of the "applicative" school of programming—the school that says you should never make any side effects whatsoever—are profoundly disturbed by the mere mention of **setq**. For them, all results must come about purely by the way that functions compute their values and hand them to other functions.

The only bindings that the advocates of the applicative style approve of are transitory "lambda bindings"—those that arise when a function is applied to its arguments. Whenever any function is called, that function's dummy variables temporarily assume "lambda bindings". These bindings are just like those caused by a **setq**, except that they are fleeting. That is, the moment the function is finished computing, they go away—vanishing without a trace. For example, during the computation of **(rac '(a b c))**, the lambda binding of the dummy variable **lyst** is the list **(a b c)**; but as soon as the answer **c** is passed along to the function or person that requested the **rac**, the value of the atom **lyst** used in getting that answer is totally forgotten. The Lisp interpreter will tell you that **lyst** is an "unbound atom" if you ask for its value. Applicative programmers much prefer lambda bindings to ordinary **setq** bindings.

I personally am not a fanatic about avoiding **setq**'s and other functions that cause side effects. Though I find the applicative style to be jus-telegant, I find it impractical when it comes to the construction of large AI-style programs. Therefore I shall not advocate the applicative style here, though I shall adhere to it when possible. Strictly speaking, in applicative programming, you cannot even define new functions, since a **def** statement causes a permanent change to take place in the genie's memory—namely, the permanent storage in memory of the function definition. So the ideal applicative approach would have functions, like variable bindings, being created only temporarily, and their definitions would be discarded the moment after they had been used. But this is extreme "applicativism".

For your edification, here are a few more simple function definitions.

```
→ (def rdc (lambda (lyst) (reverse (cdr (reverse lyst)))))
→ (def snoc (lambda (x lyst) (reverse (cons x (reverse lyst)))))
→ (def twice (lambda (n) (plus n n)))
```

The functions **rdc** and **snoc** are analogous to **cdr** and **cons**, only backwards. Thus, the **rdc** of **(a b c d e)** is **(a b c d)**, and if you type **(snoc 5 '(1 2 3 4))**, you will get **(1 2 3 4 5)** as your answer.

* * *

All of this is mildly interesting so far, but if you want to see the genie do anything *truly* surprising, you have to allow it to make some decisions based on things that happen along the way. These are sometimes called "conditional wishes". A typical example would be the following:

```
→ (cond ((eq x 1) 'land) ((eq x 2) 'sea))
```

The value returned by this statement will be the atom **land** if **x** has value 1, and the atom **sea** if **x** has value 2. Otherwise, the value returned will be **nil** (*i.e.*, if **x** is 5). The atom **eq** (pronounced "eek") is the name of a common Lisp function that returns the atom **t** (standing for "true") if its two arguments have the same value, and **nil** (for "no" or "false") if they do not.

A **cond** statement is a list whose **car** is the function name **cond**, followed by any number of *cond clauses*, each of which is a two-element list. The first element of each clause is called its *condition*, the second element its *result*. The clauses' conditions are checked out by the Lisp genie one by one, in order; as soon as it finds a clause whose condition is "true" (meaning that the condition returns anything other than **nil**!), it begins calculating that clause's *result*, whose value gets returned as the value of the whole **cond** statement. None of the further clauses is even so much as glanced at! This may sound more complex than it ought to. The real idea is no more complex than saying that it looks for the first condition that is satisfied, then it returns the corresponding result.

Often one wants to have a catch-all clause at the end whose condition is *sure* to be satisfied, so that, if all other conditions fail, at least this one will be true and the accompanying result, rather than **nil**, will be returned. It is easy as pie to make a condition whose value is non-**nil**; just choose it to be **t**, for instance, as in the following:

```
→ (cond ((eq x 1) 'land)
        ((eq x 2) 'sea)
        (t 'air))
```

Depending on what the value of **x** is, we will get either **land**, **sea**, or **air** as the value of this **cond**, but we'll never get **nil**. Now here are a few sample **cond** statements for you to play genie to:

```
→ (cond ((eq (eval pi) pie) (eval (snoc pie pi)))
        (t (eval (snoc (rac pi) pi))))
→ (cond ((eq 2 2) 2) ((eq 3 3) 3))
→ (cond (nil 'no-no-no)
        ((eq '(car nil) '(cdr nil)) 'hmmm)
        (t 'yes-yes-yes))
```

The answers are: **8, 2**, and **yes-yes-yes**. Did you notice that **(car nil)** and **(cdr nil)** were quoted?

I shall close this portion of the column by displaying a patterned family of function definitions, so obvious in their pattern that you would think that the Lisp genie would just sort of "get the hang of it" after seeing the first few . . . Unfortunately, though, Lisp genies are frustratingly dense (or at least they play at being dense), and they will not jump to any conclusion unless it has been completely spelled out. Look first at the family:

```
→ (def square (lambda (k) (times k k)))
→ (def cube (lambda (k) (times k (square k))))
→ (def 4th-power (lambda (k) (times k (cube k))))
→ (def 5th-power (lambda (k) (times k (4th-power k))))
→ (def 6th-power (lambda (k) (times k (5th-power k))))
  .
  .
  .
  .
```

My question for you is this: Can you invent a definition for a *two*-parameter function that subsumes *all* of these in one fell swoop? More concretely, the question is: How would one go about defining a two-parameter function called **power** such that, for instance, **(power 9 3)** yields 729 on being evaluated, and **(power 7 4)** yields 2,401? I have supplied you, in this column, with all the necessary tools to do this, provided you exercise some ingenuity.

* * *

I thought I would end this column with a newsbreak about a freshly discovered beast—the homely Glazunkian porpuquine, so called because it is found only on the island of Glazunkia (claimed by Upper Bitbo, though it is just off the coast of Burronymede). And what is a porpuquine, you ask? Why, it's a strange breed of porcupine, whose quills—of which, for some reason, there are always exactly nine (in Outer Glazunkia) or seven (in Inner Glazunkia)—are smaller porpuquines. Oho! This would certainly seem to be an infinite regress! But no. It's just that I forgot to mention that there is a smallest size of porpuquine: the zero-inch type, which, amazingly enough, is totally bald of quills. So, quite luckily (or perhaps unluckily, depending on your point of view), that puts a stop to the threatened infinite regress. This remarkable beast is shown in a rare photograph in Figure 17-1.

Students of zoology might be interested to learn that the quills on 5-inch porpuquines are always 4-inch porpuquines, and so on down the line. And students of anthropology might be equally intrigued to know that the residents of Glazunkia (both Outer and Inner) utilize the nose (yes, the nose) of the zero-inch porpuquine as a unit of barter—an odd thing to our

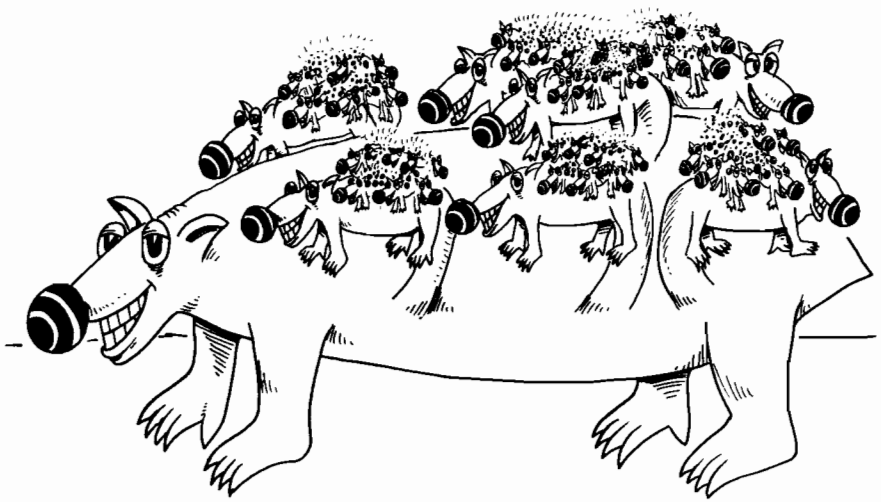

FIGURE 17-1. *The homely Inner Glazunkian porpuquine,* Porpuquinus verdimontianus. *The size of this particular specimen has not been ascertained, although it appears to be at least a 4-incher. The buying power of a porpuquine is the number of zero-inch noses on it. Larger noses, oddly enough, are worth nothing.* [*Photograph by David J. Moser.*]

minds; but then, who are you and I to question the ancient wisdom of the Outer and Inner Glazunkians? Thus, since a largish porpuquine—say a 3-incher or 4-incher—contains many, many such tiny noses, it is a most valuable commodity. The value of a porpuquine is sometimes referred to as its "buying power", or just "power" for short. For instance, a 2-incher found in Inner Glazunkia is almost twice as powerful as a 2-incher found in Outer Glazunkia. Or did I get it backward? It's rather confusing!

Anyway, why am I telling you all this? Oh, I just thought you'd like to hear about it. Besides, who knows? You just might wind up visiting Glazunkia (Inner or Outer) one of these fine days. And then all of this could come in mighty handy.

18

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 $9\times9\times9\times1$ 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-incher, that porpuquine would have a buying power of $7\times7\times7\times7\times1=2{,}401$ 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 **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:
   cond (eq s 0) 1;
        t (times q (buying-power q (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 **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:

```
(def buying-power (lambda (q s)
   (cond ((eq s 0) 1)
         (t (times q (buying-power q (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 *déjà 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 naïve 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 **n** does this riddle present absolutely no problem at all? That's easy: when **n** 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.

```
→ (def wow (lambda (n)
     (cond ((eq n 0) 1)
           (t (times n (wow (sub1 n)))))))
```

Remember, **sub1** 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 933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000000—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:

(1*a*) How can you know when you've reached the embryonic case?
(1*b*) What is the embryonic answer?

(2*a*) From a typical case, how do you take exactly one step toward the embryonic case?
(2*b*) How do you build this case's answer out of the "magically given" answer to the simpler case?

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

or bottom line. Question (2*b*) 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:

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

(2*a*) Subtract 1 from the present argument.
(2*b*) 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 **sub1** 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 (add1 (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** (**add1** 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 **sub1** 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 (2*b*)—how to recursively build a list documenting our wildly oscillating pathway? Well, the pathway belonging to 7 is gotten by tacking (*i.e.*, **cons**ing) 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-1 3)
   ENTERING pathway-to-1 (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-1 (tato=4)
                     ENTERING pathway-to-1 (tato=2)
                        ENTERING pathway-to-1 (tato=1)
                        EXITING pathway-to-1 (value: (1))
                     EXITING pathway-to-1 (value: (2 1))
                  EXITING pathway-to-1 (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' 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 bc 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:

(1*a*) Is there just one embryonic case, or are there several, or even an infinite class of them?
(1*b*) How can you know when you've reached an embryonic case?
(1*c*) What are the answers to the various embryonic cases?

(2*a*) From a typical case, is there exactly one way to step toward an embryonic case, or are there various possibilities?
(2*b*) From a typical case, how do you determine which of the various routes toward an embryonic case to take?
(2*c*) 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.

```
-> (atomcount brahma)
 ENTERING atomcount (s=(((ac ab cb) ac (ba bc ac)) ab ((cb ca ba) cb (ac ab cb))))
  ENTERING atomcount (s=((ac ab cb) ac (ba bc ac)))
   ENTERING atomcount (s=(ac ab cb))
    ENTERING atomcount (s=ac)
    EXITING  atomcount (value: 1)
    ENTERING atomcount (s=(ab cb))
     ENTERING atomcount (s=ab)
     EXITING  atomcount (value: 1)
     ENTERING atomcount (s=(cb))
      ENTERING atomcount (s=cb)
      EXITING  atomcount (value: 1)
      ENTERING atomcount (s=nil)
      EXITING  atomcount (value: 0)
     EXITING  atomcount (value: 1)
    EXITING  atomcount (value: 2)
   EXITING  atomcount (value: 3)
   ENTERING atomcount (s=(ac (ba bc ac)))
    ENTERING atomcount (s=ac)
    EXITING  atomcount (value: 1)
    ENTERING atomcount (s=((ba bc ac)))
     ENTERING atomcount (s=(ba bc ac))
      ENTERING atomcount (s=ba)
      EXITING  atomcount (value: 1)
      ENTERING atomcount (s=(bc ac))
       ENTERING atomcount (s=bc)
       EXITING  atomcount (value: 1)
       ENTERING atomcount (s=(ac))
        ENTERING atomcount (s=ac)
        EXITING  atomcount (value: 1)
        ENTERING atomcount (s=nil)
        EXITING  atomcount (value: 0)
       EXITING  atomcount (value: 1)
      EXITING  atomcount (value: 2)
     EXITING  atomcount (value: 3)
     ENTERING atomcount (s=nil)
     EXITING  atomcount (value: 0)
    EXITING  atomcount (value: 3)
   EXITING  atomcount (value: 4)
  EXITING  atomcount (value: 7)
  ENTERING atomcount (s=(ab ((cb ca ba) cb (ac ab cb))))
   ENTERING atomcount (s=ab)
   EXITING  atomcount (value: 1)
   ENTERING atomcount (s=(((cb ca ba) cb (ac ab cb))))
    ENTERING atomcount (s=((cb ca ba) cb (ac ab cb)))
     ENTERING atomcount (s=(cb ca ba))
      ENTERING atomcount (s=cb)
      EXITING  atomcount (value: 1)
      ENTERING atomcount (s=(ca ba))
       ENTERING atomcount (s=ca)
       EXITING  atomcount (value: 1)
       ENTERING atomcount (s=(ba))
        ENTERING atomcount (s=ba)
        EXITING  atomcount (value: 1)
        ENTERING atomcount (s=nil)
        EXITING  atomcount (value: 0)
       EXITING  atomcount (value: 1)
      EXITING  atomcount (value: 2)
     EXITING  atomcount (value: 3)
     ENTERING atomcount (s=(cb (ac ab cb)))
      ENTERING atomcount (s=cb)
      EXITING  atomcount (value: 1)
      ENTERING atomcount (s=((ac ab cb)))
       ENTERING atomcount (s=(ac ab cb))
        ENTERING atomcount (s=ac)
        EXITING  atomcount (value: 1)
        ENTERING atomcount (s=(ab cb))
         ENTERING atomcount (s=ab)
         EXITING  atomcount (value: 1)
         ENTERING atomcount (s=(cb))
          ENTERING atomcount (s=cb)
          EXITING  atomcount (value: 1)
          ENTERING atomcount (s=nil)
          EXITING  atomcount (value: 0)
         EXITING  atomcount (value: 1)
        EXITING  atomcount (value: 2)
       EXITING  atomcount (value: 3)
       ENTERING atomcount (s=nil)
       EXITING  atomcount (value: 0)
      EXITING  atomcount (value: 3)
     EXITING  atomcount (value: 4)
    EXITING  atomcount (value: 7)
    ENTERING atomcount (s=nil)
    EXITING  atomcount (value: 0)
   EXITING  atomcount (value: 7)
  EXITING  atomcount (value: 8)
 EXITING  atomcount (value: 15)
15
->
```

FIGURE 18–1. *A trace of the execution of the Lisp function-call* **(atomcount brahma)**. *Recursion in action!*

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 Benoît 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 bc ab ca cb ab

Now tell me: What are the values of the atoms **a**, **b**, and **c**? Here comes the answer, so don't peek. They are, respectively: **a**, **a**, and **a**. This may seem a bit confusing. You may be reassured to know that in Lisp, **set** is not very commonly used, and such confusions do not arise that often.

* * *

Psychologically, one of the great powers of programming is the ability to define new compound operations in terms of old ones, and to do this over and over again, thus building up a vast repertoire of ever more complex operations. It is quite reminiscent of evolution, in which ever more complex molecules evolve out of less complex ones, in an ever-upward spiral of complexity and creativity. It is also quite reminiscent of the industrial revolution, in which people used very simple early machines to help them build more complex machines, then used those in turn to build even more complex machines, and so on, once again in an ever-upward spiral of complexity and creativity. At each stage, whether in evolution or revolution, the products get more flexible and more intricate, more "intelligent" and yet more vulnerable to delicate "bugs" or breakdowns.

Likewise with programming in Lisp, only here the "molecules" or "machines" are now Lisp functions defined in terms of previously known Lisp functions. Suppose, for instance, that you wish to have a function that will always return the last element of a list, just as **car** always returns the first element of a list. Lisp does not come equipped with such a function, but you can easily create one. Do you see how? To get the last element of a list called **lyst**, you simply do a **reverse** on **lyst** and then take the **car** of that: **(car (reverse lyst))**. To dub this operation with the name **rac** (**car** backwards), we use the **def** function, as follows:

→ **(def rac (lambda (lyst) (car (reverse lyst))))**

Using **def** this way creates a *function definition.* In it, the word **lambda** followed by **(lyst)** indicates that the function we are defining has only one parameter, or dummy variable, to be called **lyst**. (It could have been called anything; I just happen to like the atom **lyst**.) In general, the list of parameters (dummy variables) must immediately follow the word **lambda**. After this "def wish" has been carried out, the **rac** function is as well understood by the genie as is **car**. Thus **(rac '(your brains))** will yield the atom **brains**. And we can use **rac** itself in definitions of yet further functions. The whole thing snowballs rather miraculously, and you can quickly become overwhelmed by the power you wield.

Here is a simple example. Suppose you have a situation where you know you are going to run into many big long lists and you know it will often be useful to form, for each such long list, a short list that contains just its **car** and **rac**. We can define a one-parameter function to do this for you:

→ **(def readers-digest-condensed-version**
(lambda (biglonglist)
(cons (car biglonglist) (cons (rac biglonglist) nil))))

Thus if we apply our new function **readers-digest-condensed-version** to the entire text of James Joyce's *Finnegans Wake* (treating it as a big long list of words), we will obtain the shorter list **(riverrun the)**. Unfortunately, reapplying the condensation operator to this new list will not simplify it any further.

It would be nice as well as useful if we could create an inverse operation to **readers-digest-condensed-version** called **rejoyce** that, given any two words, would create a novel beginning and ending with them, respectively —and such that James Joyce would have written it (had he thought of it). Thus execution of the Lisp statement **(rejoyce 'Stately 'Yes)** would result in the Lisp genie generating from scratch the entire novel *Ulysses.* Writing this function is left as an exercise for the reader. To test your program, see what it does with **(rejoyce 'karma 'dharma)**.

* * *

One goal that has seemed to some people to be both desirable and feasible using Lisp and related programming languages is (1) to make every single statement return a *value* and (2) to have it be through this returned value and *only* through it that the statement has any effect. The idea of (1) is that values are handed "upward" from the innermost function calls to the outermost ones, until the full statement's value is returned to you. The idea of (2) is that during all these calls, no atom has its value changed at all (unless the atom is a dummy variable). In all dialects of Lisp known to me, (1) is true, but (2) is not necessarily true.

Thus if **x** is bound to **(a b c d e)** and you say **(car (cdr (reverse x)))**, the first thing that happens is that **(reverse x)** is calculated; then this value is handed "up" to the **cdr** function, which calculates the **cdr** of that list; finally, this shorter list is handed to the **car** function, which extracts one element—namely the atom **d**—and returns it. In the meantime, the atom **x** has suffered no damage; it is still bound to **(a b c d e)**.

It might seem that an expression such as **(reverse x)** would change the value of **x** by reversing it, just as carrying out the oral command "Turn your sweater inside out" will affect the sweater. But actually, carrying out the wish **(reverse x)** no more changes the value of **x** than carrying out the wish **(plus 2 2)** changes the value of 2. Instead, executing **(reverse x)** causes a *new* (unnamed) list to come into being, just like **x**, only reversed. And that list is the value of the statement; it is what the statement returns. The value of **x** itself, however, is untouched. Similarly, evaluating **(cons 5 pi)** will not change the list named **pi** in the slightest; it merely returns a new list with **5** as its **car** and whatever **pi**'s value is as its **cdr**.

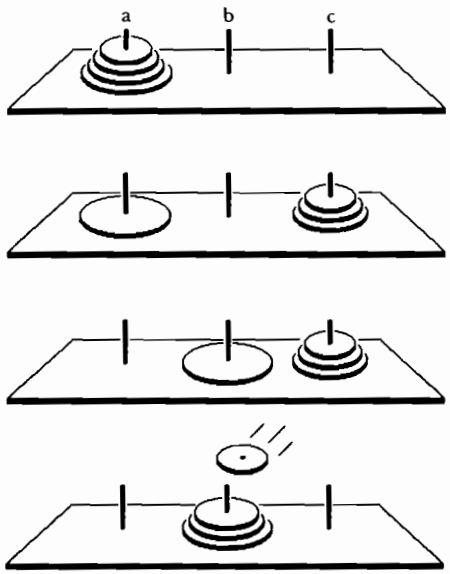

FIGURE 19–1. *A smaller Tower of Brahma puzzle. At the top, the starting position. Below it is shown in an intermediate stage, in which a three-high pile has been transferred from needle* a *to needle* c. *At this point, the biggest disk has become free, and can be jumped to needle* b. *Then all that is left is to re-transfer the three-high pile from* c *to* b. *When this is done, the world will end. Thus, the final picture shows an artist's conception of the world a mere split-second before it all turns to dust.*

merely need to know how to move 62. And to move 62, you merely need to know how to move 61. On it goes down the line, until you "bottom out" at the "embryonic case" of the Tower of Brahma puzzle, the 1-disk puzzle. Now, I'll admit that you have to keep track of where you are in this process, and that may be a bit tedious—but that's merely bookkeeping. In principle, you now could actually carry the whole process out—if you were bent on seeing the world end!

As our first approximation to a Lisp function, let's write an English description of the method. Let's call the three needles **sn**, **dn**, and **hn**, standing for "source-needle", "destination-needle", and "helping-needle". Here goes:

To move a tower of height n from sn to dn making use of hn:
if n = 1, then just carry that one disk directly from sn to dn;
otherwise, do the following three steps:

(1) move a tower of height n−1 from sn to hn making use of dn;
(2) carry 1 disk from sn to dn;
(3) move a tower of height n−1 from hn to dn making use of sn.

Here, lines (1) and (3) are the two recursive calls; skirting paradox, they seem to call upon the very ability they are helping to define. The saving feature is that they involve **n−1** disks instead of **n**. Note that in line (1), **hn** plays the "destination" role while **dn** plays the "helper" role. And in (3), **hn** plays the "source" role while **sn** plays the "helper" role. Since the whole thing is recursive, every needle will be switching hats many times over during the course of the transfer. That's the beauty of this puzzle and in a way it's the beauty of recursion.

Now how do we make the transition from English to Lisp? It's quite simple:

```
(def move-tower (lambda (n sn dn hn)
   (cond ((eq n 1) (carry-one-disk sn dn))
         (t (move-tower (sub1 n) sn hn dn)
            (carry-one-disk sn dn)
            (move-tower (sub1 n) hn dn sn)))))
```

Where are the Lisp equivalents of the English words "from", "to", and "making use of"? They seem to have disappeared! So how can the genie know which needle is to play which role at each stage? The answer is, this information is conveyed *positionally*. There are, in this function definition, four parameters: one integer and three "dummy needles". The first of these three is the source, the second the destination, the third the helper. Thus in the initial list of parameters (following the **lambda**) they are in the order **sn dn hn.** In the first recursive call, the Lisp translation of line (1), they are in the order **sn hn dn**, showing how **hn** and **dn** have switched hats. In the second recursive call, the Lisp translation of line (3), you can see that **hn** and **sn** have switched hats.

The point is that the atom names **sn**, **dn**, and **hn** carry no intrinsic meaning to the genie. They could as well have been **apple**, **banana**, and **cherry**. Their meanings are defined operationally, by the places where they appear in the various parts of the function definition. Thus it would have been a gross blunder to have written, for instance, **(move-tower (sub1 n) sn dn hn)** as Lisp for line (1), because this contains no indication that **hn** and **dn** must switch roles in that line.

* * *

An important question remains. What happens when that friendly Lisp genie comes to a line that says **carry-one-disk**? Does it suddenly zoom off

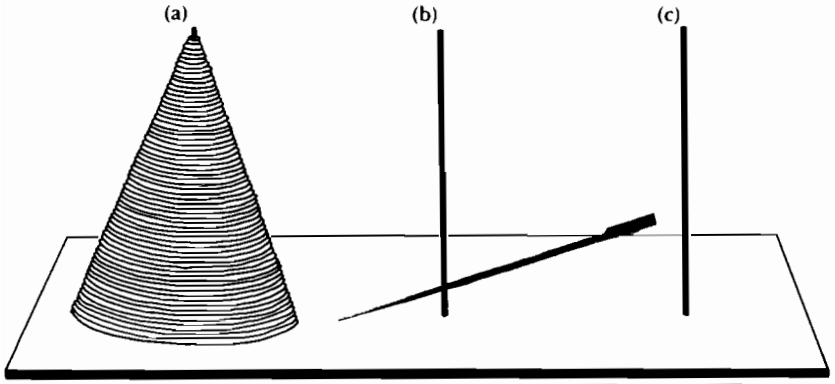

FIGURE 18–2. *The Tower of Brahma puzzle, with 64 disks to be transferred.* [*Drawing by David Moser.*]

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 bc 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 . . .

19

Lisp: Recursion and Generality

April, 1983

In the preceding column, I described Edouard Lucas' Tower of Brahma puzzle, in which the object is to transfer a tower of 64 gold disks from one diamond needle to another, making use of a third needle on which disks can be placed temporarily. Disks must be picked up and moved one at a time, the only other constraint being that no disk may ever sit on a smaller one. The problem I posed for readers was to come up with a recursive description, expressed as a Lisp function, of how to accomplish this goal (and thereby end the world).

I pointed out that the recursion is evident enough: to transfer 64 disks from one needle to another (using a third), it suffices to know how to transfer 63 disks from one needle to another (using a third). To recap it, the idea is this. Suppose that the 64-disk tower of Brahma starts out on needle **a.** Figure 19-1 shows a schematic picture, representing all 64 disks by a mere 4. First of all, using your presumed 63-disk-moving ability, transfer 63 disks from needle **a** to needle **c**, using needle **b** as your "helping needle". Figure 19-1*b* shows how the set-up now looks. (Note: In the figure, 4 plays the role of 64, so 3 plays the role of 63, but for some reason, 1 doesn't play the role of 61. Isn't that peculiar?) All right. Now simply pick up the one remaining **a**-disk—the biggest disk of all—and plunk it down on needle **b**, as is shown in Figure 19-1*c*. Now you can see how easy it will be to finish up—simply re-exploit your 63-disk ability so as to transfer that pile on **c** back to **b**, this time using **a** as the "helping needle". Notice how in this maneuver, needle **a** plays the helping role that needle **b** played in the previous 63-disk maneuver. Figure 19-1*d* shows the situation just a split second before the last disk is put in place. Why not *after* it's in place? Simple: Because the entire world then turns to dust, and it's too hard to draw dust.

* * *

Now someone might complain that I left out all the hard parts: "You just magically assumed an ability to move 63 disks!" So it might seem, but there's nothing magical about such an assumption. After all, to move 63, you

to the Temple at Benares and literally heft a solid gold disk? Or, more prosaically, does it pick up a plastic disk on a table and transfer it from one plastic needle to another? In other words, does some physical action, rather than a mere computation, take place?

Well, in theory that is quite possible. In fact, even in practice the execution of a Lisp command could actually cause a mechanical arm to move to a specific location, to pick up whatever its mechanical hand grasps there, to carry that object to another specific location, and then to release it there. In these days of industrial robots, there is nothing science-fictional about that. However, in the absence of a mechanical arm and hand to move physical disks, what could it mean?

One obvious and closely related possibility is to have there be a display of the puzzle on a screen, and for **carry-one-disk** to cause a *picture* of a hand to move, to *appear* to grasp a disk, pick it up, and replace it somewhere else. This would amount to simulating the puzzle graphically, which can be done with varying degrees of realism, as anyone who has seen science-fiction films using state-of-the-art computer graphics techniques knows.

However, suppose that we don't have fancy graphics hardware or software at our disposition. Suppose that all we want is to create a printed recipe telling us how to move our own soft, organic, human hands so as to solve the puzzle. Thus in the case of a three-disk puzzle, the desired recipe might read **ab ac bc ab ca cb ab** or equally well **12 13 23 12 31 32 12**. What could help us reach this humbler goal?

If we had a program that moved an arm, we would be concerned not with the value it returned, but with the patterned sequence of "side effects" it carried out. Here, by contrast, we are most concerned with the value that our program is going to return—a patterned list of atoms. The list for **n** = 3 has got to be built up from two lists for **n** = 2. This idea was shown at the end of last month's column:

ab ac bc ab ca cb ab

In Lisp, to set groups apart, rather than using wider spaces, we use parentheses. Thus our goal for **n** = 3 might be to produce the sandwich-like list **((ab ac bc) ab (ca cb ab))**. One way to produce a list out of its components is to use **cons** repeatedly. Thus if the values of the atoms **apple**, **banana**, and **cherry** are 1, 2, and 3, respectively, then the value returned by **(cons apple (cons banana (cons cherry nil)))** will be the list **(1 2 3)**. However, there is a shorter way to get the same result, namely, to write **(list apple banana cherry)**. It returns the same value. Similarly, if the atoms **sn** and **dn** are bound to **a** and **b** respectively, then execution of the command **(list sn dn)** will return **(a b)**. The function **list** is an unusual function in that it takes any number of arguments at all—even none, so that **(list)** returns the value of **nil**!

Now let us tackle the problem of what value we want the function called **carry-one-disk** to return. It has two parameters that represent needles, and

ideally we'd like it to return a single atom made out of those needles' names, such as **ab** or **12**. For the moment, it'll be easier if we assume that the needle names are the numbers 1, 2, and 3. In this case, to make the number 12 out of 1 and 2, it suffices to do a little bit of arithmetic: multiply the first by 10 and add on the second. Here is the Lisp for that:

```
(def carry-one-disk (lambda (sn dn) (plus (times 10 sn) dn)))
```

On the other hand, if the needle names are nonnumeric atoms, then we can use a standard Lisp function called **concat**, which takes the values of its arguments (any number, as with **list**) and concatenates them all so as to make one big atom. Thus **(concat 'con 'cat 'e 'nate)** returns the atom **concatenate**. In such a case, we could write:

```
(def carry-one-disk (lambda (sn dn) (concat sn dn)))
```

Either way, we have solved the "bottom line" half of the **move-tower** problem.

The other half of the problem is what the recursive part of **move-tower** will return. Well, that is pretty simple. We simply would like it to return a sandwich-like list in which the values of the two recursive calls flank the value of the single call to **carry-one-disk**. So we can modify our previous recursive definition very slightly, by adding the word **list**:

```
(def move-tower (lambda (n sn dn hn)
   (cond ((eq n 1) (carry-one-disk sn dn))
         (t (list (move-tower (sub1 n) sn hn dn)
                  (carry-one-disk sn dn)
                  (move-tower (sub1 n) hn dn sn))))))
```

Now let's conduct a little exchange with the Lisp genie:

```
→ (move-tower 4 'a 'b 'c)
(((ac ab cb) ac (ba bc ac)) ab ((cb ca ba) cb (ac ab cb)))
→
```

Smashing! It actually works! Isn't that pretty? In last month's column, this list was called **brahma.**

* * *

Suppose we wished to suppress all the inner parentheses, so that just a long uninterrupted sequence of atoms would be printed out. For instance, we would get **(ac ab cb ac ba bc ac ab cb ca ba cb ac ab cb)** instead of the intricacy of **brahma**. This would be slightly less informative, but it would be more impressive in its opaqueness. In this case, we would not want to

use the function **list** to make our sandwich of three values, but would have to use some other function that removed the parentheses from the two flanking recursive values.

This is a case where the Lisp function **append** comes in handy. It splices any number of lists together, dropping their outermost parentheses as it does so. Thus **(append '(a (b)) '(c) nil '(d e))** yields the five-element list **(a (b) c d e)** rather than the four-element list **((a (b)) (c) nil (d e))**, which would be yielded if **list** rather than **append** appeared in the function position. Using **append** and a slightly modified version of **carry-one-disk** to work with it, we can formulate a final definition of **move-tower** that does what we want:

```
(def move-tower (lambda (n sn dn hn)
   (cond ((eq n 1) (carry-one-disk sn dn))
         (t (append (move-tower (sub1 n) sn hn dn)
                    (carry-one-disk sn dn)
                    (move-tower (sub1 n) hn dn sn))))))

(def carry-one-disk (lambda (sn dn) (list (concat sn dn))))
```

To test this out, I asked the Lisp genie to solve a 9-high Tower of Brahma puzzle. Here is what it shot back at me, virtually instantaneously:

```
(ab ac bc ab ca cb ab ac bc ba ca bc ab ac bc ab ca cb ab ca bc ba ca cb
 ab ac bc ab ca cb ab ac bc ba ca bc ab ac bc ba ca cb ab ca bc ba ca bc
 ab ac bc ab ca cb ab ac bc ba ca bc ab ac bc ab ca cb ab ca bc ba ca cb
 ab ac bc ab ca cb ab ca bc ba ca bc ab ac bc ba ca cb ab ca bc ba ca cb
 ab ac bc ab ca cb ab ac bc ba ca bc ab ac bc ab ca cb ab ca bc ba ca cb
 ab ac bc ab ca cb ab ac bc ba ca bc ab ac bc ba ca cb ab ca bc ba ca bc
 ab ac bc ab ca cb ab ac bc ba ca bc ab ac bc ba ca cb ab ca bc ba ca cb
 ab ac bc ab ca cb ab ca bc ba ca bc ab ac bc ba ca cb ab ca bc ba ca bc
 ab ac bc ab ca cb ab ac bc ba ca bc ab ac bc ab ca cb ab ca bc ba ca cb
 ab ac bc ab ca cb ab ac bc ba ca bc ab ac bc ba ca cb ab ca bc ba ca bc
 ab ac bc ab ca cb ab ac bc ba ca bc ab ac bc ab ca cb ab ca bc ba ca cb
 ab ac bc ab ca cb ab ca bc ba ca bc ab ac bc ba ca cb ab ca bc ba ca cb
 ab ac bc ab ca cb ab ac bc ba ca bc ab ac bc ab ca cb ab ca bc ba ca cb
 ab ac bc ab ca cb ab ca bc ba ca bc ab ac bc ba ca cb ab ca bc ba ca bc
 ab ac bc ab ca cb ab ac bc ba ca bc ab ac bc ba ca cb ab ca bc ba ca cb
 ab ac bc ab ca cb ab ca bc ba ca bc ab ac bc ba ca cb ab ca bc ba ca cb
 ab ac bc ab ca cb ab ac bc ba ca bc ab ac bc ab ca cb ab ca bc ba ca cb
 ab ac bc ab ca cb ab ac bc ba ca bc ab ac bc ba ca cb ab ca bc ba ca bc
 ab ac bc ab ca cb ab ac bc ba ca bc ab ac bc ab ca cb ab ca bc ba ca cb
 ab ac bc ab ca cb ab ca bc ba ca bc ab ac bc ba ca cb ab ca bc ba ca cb
 ab ac bc ab ca cb ab ac bc ba ca bc ab ac bc ab ca cb ab ca bc ba ca cb
 ab ac bc ab ca cb ab)
```

Now that's the kind of genie I like!

* * *

Congratulations! You have just been through a rather sophisticated and brain-taxing example of recursion. Now let us take a look at a recursion that offers us a different kind of challenge. This recursion comes from an offhand remark I made last column. I used the odd variable name **tato**, mentioning that it is a recursive acronym standing for **tato (and tato only)**. Using this fact you can expand **tato** any number of times. The sole principle is that each occurrence of **tato** on a given level is replaced by the two-part phrase **tato (and tato only)** to make the next level. Here is a short table:

```
n=0: tato
n=1: tato
       (and tato only)
n=2: tato
       (and tato only)
        (and tato (and tato only) only)
n=3: tato
       (and tato only)
        (and tato (and tato only) only)
         (and tato (and tato only) (and tato (and tato only) only) only)
```

For us the challenge is to write a Lisp function that returns what **tato** becomes after **n** recursive expansions, for any **n**. Irrelevant that for **n** much bigger than 3 the whole thing gets ridiculously large. We're theoreticians!

There is only one problem. Any Lisp function must return a single Lisp structure (an atom or a list) as its value; however, the entries in our table do not satisfy this criterion. For instance, the one for **n** = 2 consists of one atom followed by two lists. To fix this, we can turn each of the entries in the table into a list by enclosing it in one outermost pair of parentheses. Now our goal is consistent with Lisp. How do we attain it?

Recursive thinking tells us that the bottom line, or embryonic case, occurs when **n** = 0, and that otherwise, the **n**th line is made from the line before it by replacing the atom **tato,** wherever it occurs, by the list **(tato (and tato only))**, only without its outermost parentheses. We can write this up right away.

```
(def tato-expansion (lambda (n)
 (cond ((eq n 0) '(tato))
       (t (replace 'tato '(tato (and tato only)) (tato-expansion (sub1 n)))))))
```

The only thing is, we have not specified what we mean by **replace**. We must be very careful in defining how to carry out our **replace** operation. Look at any of the lines of the **tato** table, and you will see that it contains

one element more than the preceding line. Why is this? Because the atom **tato** gets replaced each time by a two-element list whose parentheses, as I pointed out earlier, are dropped during the act of replacement. It's this parenthesis-dropping that is the sticky point. A less tricky example of such parenthesis-dropping replacement than the recursive one involving **tato** would be this: **(replace 'a '(1 2 3) '(a b a))**, whose value should be **(1 2 3 b 1 2 3)** rather than **((1 2 3) b (1 2 3))**. Rather than exact substitution of a list for an atom, this kind of replacement involves splicing or appending a list inside a longer list.

Let's try to specify in Lisp—using recursion, as usual—just what we mean by **replace**-ing all occurrences of the atom **atm** by a list called **lyst**, inside a long list called **longlist**. This is a good puzzle for you to try. A hint: See how the answer for argument **(a b a)** is built out of the answer for argument **(b a)**. Also look at other simple cases like that, moving back down toward the embryonic case.

* * *

The embryonic case occurs when **longlist** is **nil**. Then, of course, nothing happens so our answer should be **nil**.

The recursive case involves building a more complex answer from a simpler one assumed given. We can fall back on our **(a b a)** example for this. We can build the complex answer **(1 2 3 b 1 2 3)** out of the simpler answer **(b 1 2 3)** by appending **(1 2 3)** onto it. On the other hand, we could consider **(b 1 2 3)** itself to be a complex answer built from the simpler answer **(1 2 3)** by **cons**ing **b** onto it. Why does one involve **append**ing and the other involve **cons**ing? Simple: Because the first case involves the atom **a**, which *does* get replaced, while the second involves the atom **b**, which does *not* get replaced. This observation allows us to attempt to write down an attempt at a recursive definition of **replace**, as follows:

```
(def replace (lambda (atm lyst longlist)
   (cond ((null longlist) nil)
         ((eq (car longlist) atm)
          (append lyst (replace atm lyst (cdr longlist))))
         (t (cons (car longlist) (replace atm lyst (cdr longlist)))))))
```

As you can see, there is an embryonic case (where longlist equals **nil**), and then one recursive case featuring **append** and one recursive case featuring **cons.** Now let's try out this definition on a new example.

```
→ (replace 'a '(1 2 3) '(a (a) b a))
(1 2 3 (a) b 1 2 3)
→
```

Whoops! It *almost* worked, except that one of the occurrences of **a** was completely missed. This means that in our definition of **replace**, we must

have overlooked some eventuality. Indeed, if you go back, you will see that an unwarranted assumption slipped in right under our noses—namely, that the elements of **longlist** are always atoms. We ignored the possibility that **longlist** might contain sublists. And what to do in such a case? Answer: Do the replacement inside those sublists as well. And inside sublists of sublists, too—and so on. Can you figure out a way to fix up the ailing definition?

* * *

We've seen a recursion before in which all parts on all levels of a structure needed to be explored; it was the function **atomcount** last chapter, in which we did a simultaneous recursion on both **car** and **cdr**. The recursive line ran **(plus (atomcount (car s)) (atomcount (cdr s)))**. Here it will be quite analogous. We'll have a recursive line featuring *two* calls on **replace**, one involving the **car** of **longlist** and one involving the **cdr**, instead of just one involving the **cdr**. And this makes perfect sense, once you think about it. Suppose you wanted to replace all the unicorns in Europe by porpuquines. One way to achieve this nefarious goal would be to split Europe into two pieces: Portugal (Europe's **car**), and all the rest (its **cdr**). After replacing all the unicorns in Portugal by porpuquines, and also all the unicorns in the rest of Europe by porpuquines, finally you would recombine the two new pieces into a reunified Europe (this is supposed to suggest a **cons** operation). Of course, to carry out this dastardly operation on Portugal, an analogous splitting and rejoining would have to take place—and so on. This suggests that our recursive line will look like this:

```
(cons (replace 'unicorn '(porpuquine) (car geographical-unit))
      (replace 'unicorn '(porpuquine) (cdr geographical-unit)))
```

or, more elegantly and more generally,

```
(cons (replace atm lyst (car longlist)) (replace atm lyst (cdr longlist)))
```

This **cons** line will cover the case where **longlist**'s **car** is nonatomic, as well as the case where it is atomic but not equal to **atm**. In order to make this work, we need to augment the embryonic case slightly: we'll say that when **longlist** is not a list but an atom, then **replace** has no effect on **longlist** at all. Conveniently, this subsumes the earlier **null** line, so we can drop that one. If we put all this together, we come up with a new, improved definition:

```
(def replace (lambda (atm lyst longlist)
   (cond ((atom longlist) longlist)
         ((eq (car longlist) atm)
          (append lyst (replace atm lyst (cdr longlist))))
         (t (cons (replace atm lyst (car longlist))
                  (replace atm lyst (cdr longlist)))))))
```

Now when we say **(tato-expansion 2)** to the Lisp genie, it will print out for us the list **(tato (and tato only) (and tato (and tato only) only))**.

* * *

Well, well. Isn't this a magnificent accomplishment? If it seems less than magnificent, perhaps we can carry it a step further. A recursive acronym—one containing a letter standing for the acronym itself—can be amusing, but what of *mutually* recursive acronyms? This could mean, for instance, two acronyms, each of which contains a letter standing for the other acronym. An example would be the pair of acronyms **NOODLES** and **LINGUINI**, standing for:

NOODLES (oodles of delicious LINGUINI), elegantly served

and

luscious itty-bitty NOODLES gotten usually in naples, italy

respectively. Notice, incidentally, that **NOODLES** is not only indirectly but also directly recursive. There's nothing wrong with that.

In general, the notion of mutual recursion means a system of arbitrarily many interwoven structures, each of which is defined in terms of one or more members of the system (possibly including itself). If we are speaking of a family of mutually recursive acronyms, then this means a collection of words, letters in any one of which can stand for any word in the family.

I have to admit that this specific notion of mutually recursive acronyms is not particularly useful in any practical sense. However, it is quite useful as a droll example of a very common abstract phenomenon. Who has not at some time mused about the inevitable circularity of dictionary definitions? Anyone can see that all words eventually are defined in terms of some fundamental set that is not further reducible, but simply goes round and round endlessly. You can amuse yourself by looking up the definition of a common word in a dictionary and replacing the main words in it by *their* definitions. I once carried this process out for "love" (defined as "A strong affection for or attachment or devotion to a person or persons"), substituting for "strong", "affection", "attachment", "devotion", and "person", and coming up with this concoction:

> A morally powerful mental state or tendency, having strength of character or will for, or affectionate regard, or loyalty, faithfulness, or deep affection to, a human being or beings, especially as distinguished from a thing or lower animal.

But not being satisfied with that, I carried the whole process one step further. This was my result:

> A set of circumstances or attributes characterizing a person or thing at a given time in, with, or by the conscious or unconscious together as a unit full of or having a specific ability or capacity in a manner relating to, dealing with, or capable of making the distinction between right and wrong in conduct, or an inclination to move or act in a particular direction or way, having the state or quality of being strong in moral strength, self-discipline, or fortitude, or the act or process of volition for, or consideration, attention, or concern full of fond or tender feeling for, or the quality, state, or instance of being faithful to, those persons or ideals that one is under obligation to defend or support, or the condition, quality or state of being worthy of trust, or a strongly felt fond or tender feeling to a creature or creatures of or characteristic of a person or persons, that lives or exists, or is assumed to do so, particularly as separated or marked off by differences from that which is conceived, spoken of, or referred to as existing as an individual entity, or from any living organism inferior in rank, dignity, or authority, typically capable of moving about but not of making its own food by photosynthesis.

Isn't it romantic? It certainly makes "love" ever more mysterious. Stuart Chase, in his lucid classic on semantics, *The Tyranny of Words,* does a similar exercise for "mind" and shows its opacity equally well. But of course concrete words as well as abstract ones get caught in this vortex of confusion. My favorite example is one I discovered while looking through a French dictionary many years ago. It defined the verb *clocher* ("to limp") as *marcher en boitant* ("to walk while hobbling", roughly), and *boiter* ("to hobble") as *clocher en marchant* ("to limp while walking"). This eager learner of French was helped precious little by that particular pair of definitions.

* * *

But let us return to mutually recursive acronyms. I put quite a bit of effort into working out a family of them, and to my surprise, they wound up dealing mostly (though by no means exclusively!) with Italian food. It all began when, inspired by **tato**, I chose the similar word **tomato** and then decided to use its plural, coming up with this meaning for **tomatoes**:

TOMATOES on MACARONI (and TOMATOES only), exquisitely SPICED.

The capitalized words here are those that are also acronyms. Here is the rest of my mutually recursive family:

MACARONI:
MACARONI and CHEESE (a REPAST of Naples, Italy)

REPAST:
rather extraordinary PASTA and SAUCE, typical

CHEESE:
cheddar, havarti, emmenthaler (especially SHARP emmenthaler)

SHARP:
strong, hearty, and rather pungent

SPICED:
sweetly pickled in CHEESE ENDIVE dressing

ENDIVE:
egg NOODLES, dipped in vinegar eggnog

NOODLES:
NOODLES (oodles of delicious LINGUINI), elegantly served

LINGUINI:
LAMBCHOPS (including NOODLES), gotten usually in Northern Italy

PASTA:
PASTA and SAUCE (that's ALL!)

ALL!:
a luscious lunch

SAUCE:
SHAD and unusual COFFEE (eccellente!)

SHAD:
SPAGHETTI, heated al dente

SPAGHETTI:
standard PASTA, always good, hot especially (twist, then ingest)

COFFEE:
choice of fine flavors, especially ESPRESSO

ESPRESSO:
excellent, strong, powerful, rich, ESPRESSO, suppressing sleep outrageously

BASTA!:
belly all stuffed (tummy ache!)

LAMBCHOPS:
LASAGNE and meat balls, casually heaped onto PASTA SAUCE

LASAGNE:
LINGUINI and SAUCE and GARLIC (NOODLES everywhere!)

RHUBARB:
RAVIOLI, heated under butter and RHUBARB (BASTA!)

RAVIOLI:
RIGATONI and vongole in oil, lavishly introduced

RIGATONI:
rich Italian GNOCCHI and TOMATOES (or NOODLES instead)

GNOCCHI:
GARLIC NOODLES over crisp CHEESE, heated immediately

GARLIC:
green and red LASAGNE in CHEESE

Any gourmet can see that little attempt has been made to have each term defined by its corresponding phrase; it is simply associated more or less arbitrarily with the phrase.

Now what happens if we begin to expand some word—say, **pasta**? At first we get simply **PASTA and SAUCE (that's ALL!)**. The next stage yields **PASTA and SAUCE (that's ALL!) and SHAD and unusual COFFEE (eccellente!) (that's a luscious lunch)**. We could obviously go on expanding acronyms forever—or at least until we filled the universe up to its very brim with mouth-watering descriptions of Italian food. But what if we were less ambitious, and wanted merely to fill half a page or so with such a description? How might we find a way to halt this seemingly bottomless recursion in midcourse?

Well, of course, the key word here is "bottomless", and the answer it implies is: Put in a mechanism to allow the recursion to bottom out. The bottomlessness comes from the fact that at every stage, every acronym is allowed to expand, that is, to spawn further acronyms. So what if, instead, we kept tight control of the spawning process, being generous in the first few "generations" and gradually letting fewer and fewer acronyms spawn progeny as the generations got later? This would be similar to a redwood tree in a forest, which begins with a single "branch" (its trunk), and that branch spawns "progeny", namely, the first generation of smaller branches, and they in turn spawn ever more progeny—but eventually a natural "bottoming out" occurs as a consequence of the fact that teeny twigs simply cannot branch further. (Somehow, trees seem to have gotten their wires crossed, since for them, bottoming out generally takes place at the top.)

If this process were completely regular, then all redwood trees would look exactly alike, and one could well agree with former Governor Reagan's memorable dictum, "If you've seen one redwood tree, then you've seen them all." Unfortunately, though, redwood trees (and some other things as well) are trickier than Governor Reagan realized, and we have to learn to deal with a great variety of different things that all go by the same name. The variety is caused by the introduction of randomness into the choices as to whether to branch or not to branch, what angle to branch at, what size branch to grow, and so on.

Similar remarks apply to the "trees" of mutually recursive acronyms. If in expanding **rhubarb** we always made exactly the same control decisions about which acronyms to expand when, then there would be one and only one type of **rhubarb** expansion, so that here too, it would make sense to say "If you've seen one **rhubarb**, you've seen them all." But if we allow some randomness to enter the decision-making about spawning, then we can get many varieties of **rhubarb**, all bearing some telltale resemblance to one another, but at a much more elusive level of perception.

How can we do this? The ideal concept to bring to bear here is that of the *random-number generator*, which serves as the computational equivalent of a coin flip or throw of dice. We'll let all decisions about whether or not to expand a given acronym depend on the outcome of such a virtual coin flip. At early stages of expansion, we'll set things up so that the coin will be very likely to come up heads (*do* expand); at later stages, it will be increasingly likely to come up tails (*don't* expand). The Lisp function **rand** will be employed for this. It takes no arguments, and each time it is called, it returns a new real number located somewhere between 0 and 1, unpredictably. (This is an exaggeration—it is actually 100 percent predictable if you know how it is computed; but since the algorithm is rather obscure, for most purposes and to most observers the behavior of this function will be so erratic as to count as totally random. The story of random-number generation is itself quite a fascinating one, and would be an entire article in itself.)

If we want an event to happen with a probability of 60 percent, first we ask **rand** for a value. If that value turns out to be 0.6 or below, we go ahead, and if not, we do not. Since over a long period of time, **rand** sprays its outputs uniformly over the interval between 0 and 1, we will indeed get the go-ahead 60 percent of the time.

* * *

So much for random decisions. How will we get an acronym to expand when told to? This is not too hard. Suppose we let each acronym be a Lisp function, as in the following example:

```
(def tomatoes (lambda ()
   '(tomatoes on macaroni (and tomatoes only), exquisitely spiced)))
```

The function **tomatoes** takes no arguments, and simply returns the list of words that it expands into. Nothing could be simpler.

Now suppose we have a variable called **acronym** whose value is some particular acronym—but we don't know which one. How could we get that acronym to expand? The way we've set it up, that acronym must act as a function call. In order for any atom to invoke a function, it must be the **car** of a list, as in the examples **(plus 2 2)**, **(rand)**, and **(rhubarb)**. Now if we were

to write **(acronym)**, then the literal atom **acronym** would be taken by the genie as a function name. But that would be a misunderstanding. It's certainly not the atom **acronym** that we want to make serve as a function name, but its *value*, be it **macaroni**, **cheese**, or what-have-you.

To do this, we employ a little trick. If the value of the atom **acronym** is **rhubarb** and if I write **(list acronym)**, then the value the Lisp genie will return to me will be the list **(rhubarb)**. However, the genie will simply see this as an inert piece of Lispstuff rather than as a little command that I would like to have executed. It cannot read my mind. So how do I get it to perform the desired operation? Answer: I remember the function called **eval**, which makes the genie look upon a given data structure as a wish to be executed. In this case, I need merely say **(eval (list acronym))** and I will get the list **(ravioli, heated under butter and rhubarb (basta!))**. And had **acronym** had a different value, then the genie would have handed me a different list.

We now have just about enough ideas to build a function capable of expanding mutually recursive acronyms into long but finite phrases whose sizes and structures are controlled by many "flips" of the **rand** coin. Instead of stepping you through the construction of this function, I shall simply display it below, and let you peruse it. It is modeled very closely on the earlier function **replace**.

```
(def expand (lambda (phrase probability)
   (cond ((atom phrase) phrase)
         ((is-acronym (car phrase))
          (cond ((lessp (rand) probability)
                 (append
                  (expand (eval (list (car phrase))) (lower probability))
                  (expand (cdr phrase) probability)))
                (t
                 (cons (car phrase) (expand (cdr phrase) probability)))))
         (t (cons (expand (car phrase) (lower probability))
                  (expand (cdr phrase) probability))))))
```

Note that **expand** has two parameters. One represents the phrase to expand, the other represents the probability of expanding any acronyms that are top-level members of the given phrase. (Thus the value of the atom **probability** will always be a real number between 0 and 1.) As in the redwood-tree example, the expansion probability should decrease as the calls get increasingly recursive. That is why lines that call for expansion of **(car phrase)** do so with a *lowered* probability. To be exact, we can define the function **lower** as follows:

```
(def lower (lambda (x) (times x 0.8)))
```

Thus each time an acronym expands, its progeny are only 0.8 times as likely to expand as it was. This means that sufficiently deeply nested acronyms have a vanishingly small probability of spawning further progeny. You could use any reducing factor; there is nothing sacred about 0.8, except that it seems to yield pretty good results for me.

The only remaining undescribed function inside the definition above is **is-acronym.** Its name is pretty self-explanatory. First the function tests to see if its argument is an atom; if not, it returns **nil**. If its argument *is* an atom, it goes on to see if that atom has a function definition—in particular, a definition with the form of an acronym. If so, **is-acronym** returns the value **t**; otherwise it returns **nil**. Precisely how to accomplish this depends on your specific variety of Lisp, which is why I have not shown it explicitly. In Franz Lisp, it is a one-liner.

You may have noticed that there are two **cond** clauses in close proximity that begin with **t.** How come one "otherwise" follows so closely on the heels of another one? Well, actually they belong to different **cond**'s, one nested inside the other. The first **t** (belonging to the inner **cond**) applies to a case where we know we are dealing with an acronym but where our random coin, instead of coming down heads, has come down tails (which amounts to a decision not to expand); the second **t** (belonging to the outer **cond**) applies to a case where we have discovered we are simply not dealing with an acronym at all.

The inner logic of **expand**, when scrutinized carefully, makes perfect sense. On the other hand, no matter how carefully you scrutinize it, the output produced by **expand** using this *famiglia* of acronyms remains quite silly. Here is an example:

(rich italian green and red linguini and shad and unusual choice of fine flavors, especially excellent, strong, powerful, rich, espresso, suppressing sleep outrageously (eccellente!) and green and red lasagne in cheese (noodles everywhere!) in cheddar, havarti, emmenthaler (especially sharp emmenthaler) noodles (oodles of delicious linguini), elegantly served (oodles of delicious linguini), elegantly served (oodles of delicious linguini and sauce and garlic (noodles (oodles of delicious linguini), elegantly served everywhere!) and meat balls, casually heaped onto pasta and sauce (that's all!) and sauce (that's a luscious lunch) sauce (including noodles (oodles of delicious linguini), elegantly served), gotten usually in Northern Italy), elegantly served over crisp cheese, heated immediately and tomatoes on macaroni and cheese (a repast of Naples, Italy) (and tomatoes only), exquisitely sweetly pickled in cheese endive dressing (or noodles instead) and vongole in oil, lavishly introduced, heated under butter and rich italian gnocchi and tomatoes (or noodles instead) and vongole in oil, lavishly introduced, heated under butter and rigatoni and vongole in oil, lavishly introduced, heated under butter and ravioli, heated under butter and rich italian garlic noodles over crisp cheese, heated immediately and tomatoes (or noodles instead) and vongole in oil, lavishly introduced, heated under butter and ravioli, heated under butter and

rhubarb (basta!) (basta!) (basta!) (basta!) (belly all stuffed (tummy ache!)) (basta!))

Oh, the glories of recursive spaghetti! As you can see, Lisp is hardly the computer language to learn if you want to lose weight. Can you figure out which acronym this gastronomical monstrosity grew out of?

* * *

The **expand** function exploits one of the most powerful features of Lisp —that is, the ability of a Lisp program to take data structures it has created and treat them as pieces of code (that is, give them to the Lisp genie as commands). Here it was done in a most rudimentary way. An atom was wrapped in parentheses and the resulting minuscule list was then evaluated, or **eval**ed, as Lispers' jargon has it. The work involved in manufacturing the data structure was next to nothing, in this case, but in other cases elaborate pieces of structure can be "**cons**ed up", then handed to the Lisp genie for **eval**ing. Such pieces of code might be new function definitions, or any number of other things. The main idea is that in Lisp, one has the ability to "elevate" an inert, information-containing data structure to the level of "animate agent", where it becomes a manipulator of inert structures itself. This *program-data cycle,* or loop, can continue on and on, with structures reaching out, twisting back, and indirectly modifying themselves or related structures.

Certain types of inert, or passive, information-containing data structures are sometimes referred to as *declarative knowledge*—"declarative" because they often have a form abstractly resembling that of a declarative sentence, and "knowledge" because they encode facts about the world in some way, accessible by looking in an index in somewhat the way "book-learned" facts are accessible to a person. By contrast, animate, or active, pieces of code are referred to as *procedural knowledge*—"procedural" since they define sequences of actions ("procedures") that actually manipulate data structures, and "knowledge" since they can be viewed as embodying the program's set of skills, something like a human's unconscious skills that were once learned through long, rote drill sessions. (Sometimes these contrasting knowledge types are referred to as "knowledge that" and "knowledge how".)

This distinction should remind biologists of that between genes—relatively inert structures inside the cell—and enzymes, which are anything but inert. Enzymes are the animate agents of the cell; they transform and manipulate all the inert structures in indescribably sophisticated ways. Moreover, Lisp's loop of program and data should remind biologists of the way that genes dictate the form of enzymes, and enzymes manipulate genes (among other things). Thus Lisp's procedural-declarative program-data loop provides a primitive but very useful and tangible example of one of the most fundamental patterns at the base of life: the ability of passive structures

to control their own destiny, by creating and regulating active structures whose form they dictate.

* * *

We have been talking all along about the Lisp genie as a mysterious given agent, without asking where it is to be found or what makes it work. It turns out that one of Lisp's most exciting properties is the great ease with which one can describe the Lisp genie's complete nature in Lisp itself. That is, the Lisp interpreter can be easily written down in Lisp. Of course, if there is no prior Lisp interpreter to run it, it might seem like an absurd and pointless exercise, a bit like having a description in flowery English telling foreigners how best to learn English. But it is not so silly as that makes it sound.

In the first place, if you know enough English, you can "bootstrap" your way further into English; there is a point beyond which explanations written in English about English are indeed quite useful. What's more, that point is not too terribly far beyond the beginning level. Therefore, all you need to acquire first, and autonomously, is a "kernel"; then you can begin to lift yourself by your own bootstraps. For children, it is an exciting thing when, in reading, they begin to learn new phrases all by themselves, simply by running into them several times in succession. Their vocabulary begins to grow by leaps and bounds. So it is once there is a Lisp kernel in a system; the rest of the Lisp interpreter can be—and usually is—written in Lisp.

The fact that one can can easily write the Lisp interpreter in Lisp is no mere fluke depending on some peculiarly introverted fact about Lisp. The reason it is easy is that Lisp lends itself to writing interpreters for all sorts of languages. This means that Lisp can serve as a basis on which one can build other languages.

To put it more vividly, suppose you have designed on paper a new language called "Flumsy". If you really know how Flumsy should work, then it should not be too hard for you to write an interpreter for it in Lisp. Once implemented, your Flumsy interpreter then becomes, in essence, an intermediary genie to whom you can give wishes in Flumsy and who will in turn communicate those wishes to the Lisp genie in Lisp. Of course, all the mechanisms allowing the Flumsy genie to talk to the Lisp genie are themselves being carried out by the Lisp genie. So is this a mere façade? Is talking Flumsy really just a way of talking Lisp in disguise?

Well, when the U.S. arms negotiators talk to their Soviet counterparts through an interpreter, are they really just speaking Russian in disguise? Or is the crux of the matter whether the interpreter's native language was English or Russian, upon which the other was learned as a second tongue? And suppose you find out that in fact, the interpreter's native language was Lithuanian, that she learned English only as an adolescent and then learned Russian by taking high-school classes in English? Will you then feel that when she speaks Russian, she is actually speaking English in disguise, or worse, that she is actually speaking Lithuanian, doubly disguised?

Analogously, you might discover that the Lisp interpreter is in fact written in Pascal or some other language. And then someone could strip off the Pascal façade as well, revealing to you that the truth of the matter is that all instructions are really being executed in *machine language,* so that you are fooling yourself completely if you think that the machine is talking Flumsy, Lisp, Pascal, or any higher-level language at all!

* * *

When one interpreter runs on top of another one, there is always the question of *what level one chooses not to look below.* I personally seldom think about what underlies the Lisp interpreter, so that when I am dealing with the Lisp system, I feel as if I am talking to "someone" whose "native language" is Lisp. Similarly, when dealing with people, I seldom think about what their brains are composed of; I don't reduce them in my mind to piles of patterned neuron firings. It is natural to my perceptual system to recognize them at a certain level and not to look below that level.

If someone were to write a program that could deal in Chinese with simple questions and answers about restaurant visits, and if that program were in turn written in another language—say, the hypothetical language "SEARLE" (for "Simulated East-Asian Restaurant-Lingo Expert"), I could choose to view the system either as genuinely speaking Chinese (assuming it gave a creditable and not too slow performance), or as genuinely speaking SEARLE. I can shift my point of view at will. The one I adopt is governed mostly by pragmatic factors, such as which subject area I am currently more interested in at the moment (Chinese restaurants, or how grammars work), how closely matched the program's speed is to that of my own brain, and —not least—whether I happen to be more fluent in Chinese or in SEARLE. If to me, Chinese is a mere bunch of "squiggles and squoggles", I may opt for the SEARLE viewpoint; if on the other hand, SEARLE is a mere bunch of confusing technical gibberish, I will probably opt for the Chinese viewpoint. And if I find out that the SEARLE interpreter is in turn implemented in the Flumsy language, whose interpreter is written in Lisp, then I have two more points of view to choose from. And so on.

With interpreters stacked on interpreters, however, things become rapidly very inefficient. It is like running a motor off power created through a series of electric generators, each one being run off power coming from the preceding one: one loses a good deal at each stage. With generators there is usually no need for a long cascade, but with interpreters it is often the only way to go. If there is no machine whose machine language is Lisp, then you build a Lisp interpreter for whatever machine you have available, and run Lisp that way. And Flumsy and SEARLE, if you wish to have them at your disposal, are then built on top of this *virtual Lisp machine.* This indirectness can be annoyingly inefficient, causing your new "virtual Flumsy machine" or "virtual SEARLE machine" to run dozens of times more slowly than you would like.

* * *

Important hardware developments have taken place in the last several years, and now machines are available that are based at the hardware level on Lisp. This means that they "speak" Lisp in a somewhat deeper sense—let us say, "more fluently"—than virtual Lisp machines do. It also means that when you are on such a machine, you are "swimming" in a Lisp environment. A Lisp environment goes considerably beyond what I have described so far, for it is more than just a language for writing programs. It includes an editing program, with which one can create and modify one's programs (and text as well), a debugging program, with which one can easily localize one's errors and correct them, and many other features, all designed to be compatible with each other and with an overarching "Lisp philosophy".

Such machines, although still expensive and somewhat experimental, are rapidly becoming cheaper and more reliable. They are put out by various new companies such as LMI (Lisp Machine, Inc.), Symbolics, Inc., both of Cambridge, Massachusetts, and older companies such as Xerox. Lisp is also available on most personal computers—you need merely look at any issue of any of the many small-computer magazines to find ads for Lisp.

Why, in conclusion, is Lisp popular in artificial intelligence? There is no single answer, nor even a simple answer. Here is an attempt at a summary:

(1) Lisp is elegant and simple.
(2) Lisp is centered on the idea of lists and their manipulation—and lists are extremely flexible, fluid data structures.
(3) Lisp code can easily be manufactured in Lisp, and run.
(4) Interpreters for new languages can easily be built and experimented with in Lisp.
(5) "Swimming" in a Lisp-like environment feels natural for many people.
(6) The "recursive spirit" permeates Lisp.

Perhaps it is this last rather intangible statement that gets best at it. For some reason, many people in artificial intelligence seem to have a deep sense that recursivity, in *some* form or other, is connected with the "trick" of intelligence. This is a hunch, an intuition, a vaguely felt and slightly mystical belief, one that I certainly share—but whether it will pay off in the long run remains to be seen.

Post Scriptum.

In March of 1977, I met the great AI pioneer Marvin Minsky for the first time. It was an unforgettable experience. One of the most memorable remarks he made to me was this one: "Gödel should just have thought up

Lisp; it would have made the proof of his theorem much easier." I knew exactly what Minsky meant by that, I could see a grain of truth in it, and moreover I knew it had been made with tongue semi in cheek. Still, something about this remark drove me crazy. It made me itch to say a million things at once, and thus left me practically speechless. Finally today, after my seven-year itch, I will say some of the things I would have loved to say then.

What Minsky meant, paraphrased, is this: "Probably the hardest part of Gödel's proof was to figure out how to get a mathematical system to talk about itself. This took several strokes of genius. But Lisp can talk about itself, at least in the sense Gödel needed, *directly*. So why didn't he just invent Lisp? Then the rest would have been a piece of cake." An obvious retort is that to invent Lisp out of the blue would have taken a larger number of strokes of genius. Minsky, of course, knew this, and at bottom, his remark was clearly just a way of making this very point in a facetious way.

Still, it was clear that Minsky felt there was some serious content to the remark, as well. (And I have heard him make the same remark since then, so I know it was not just a throwaway quip.) There was the implicit question, "Why didn't Gödel invent the idea of *direct* self-reference, as in Lisp?" And this, it seemed to me, missed a crucial point about Gödel's work, which is that it showed that self-reference can crop up even where it is totally unexpected and unwelcome. The power of Gödel's result was that it obliterated the hopes for completeness of an *already known* system, namely Russell and Whitehead's *Principia Mathematica*; to have destroyed similar hopes for some newly concocted system, Lisp-like or not, would have been far less significant (or, to be more accurate, such a result's significance would have been far harder for people to grasp, even if it were equally significant).

Moreover, Gödel's construction revealed in a crystal-clear way that the line between "direct" and "indirect" self-reference (indeed, between direct and indirect *reference*, and that's even more important!) is completely blurry, because his construction pinpoints the essential role played by *isomorphism* (another name for coding) in the establishment of reference and meaning. Gödel's work is, to me, the most beautiful possible demonstration of how meaning emerges from and only from isomorphism, and of how any notion of "direct" meaning (*i.e.*, codeless meaning) is incoherent. In brief, it shows that *semantics is an emergent quality of complex syntax*, which harks back to my earlier remark in the *Post Scriptum* to Chapter 1, namely: "Content is fancy form." So the serious question implicit in Minsky's joke seemed to me to rest on a confusion about this aspect of the nature of meaning.

* * *

Now let me explain this in more detail. Part I of Gödel's insight was to realize that via a code, a number can represent a mathematical symbol 11 *e.g.*, the integer eleven can represent the left parenthesis, and the integer

thirteen the right parenthesis13. The analogue of this in human languages is the recognition that certain orally produced screeches or manually produced scratches (such as "word", "say", "language", "sentence", "reference", "grammar", "meaning", and so on) can stand for elements of language itself (as distinguished from screeches or scratches such as "cow" and "splash", which stand for extralinguistic parts of the universe). Here we have pieces *of* language talking *about* language. Maybe it doesn't seem strange to you, but put yourself, if you can, back in the shoes of barefoot cave people who had barely gotten out of the grunt stage. How amazingly magical it must have felt to the beings in whose minds such powerful concepts as *words about words* first sparked! In some sense, human consciousness began then and there.

But a language can't get very far in terms of self-reference if it can talk only about isolated symbols. Part II of Gödel's insight was to figure out how the system (and here I mean *Principia Mathematica* "and", as Gödel's paper's title says, "related systems") could talk about *lists* of symbols, and even lists of lists of symbols, and so on. In the analogy to human language, making this step is like the jump from an ability to talk about people's one-word utterances ("Paul Revere said 'Land!'.") to the ability to talk about arbitrarily long utterances, and nested ones at that ("'Douglas Hofstadter wrote, "Paul Revere said, 'Land!'.".'").

Gödel found a way to have some integers stand for *single* symbols and others stand for *lists* of symbols, usually called *strings*. An example will help. Suppose the integer 1 stands for the symbol '0', and as I mentioned earlier, 11 and 13 for parentheses. Thus to encode the string "(0)" would require you to combine the integers 11, 1, and 13 somehow into a single integer. Gödel chose the integer 7500000000000—not capriciously, of course! This integer can be viewed as the product of the three integers 2048, 3, and 1220703125, which in turn are, respectively: 2^{11}, 3^1, and 5^{13}. In other words, the three-symbol string whose symbols *individually* are coded for by 11 and 1 and 13 is coded for *in toto* by the single integer $2^{11}3^15^{13}$. Now 2, 3, and 5 are of course the first three primes, and if you want to encode a longer string, you use as many primes as you need, in increasing order. This simple scheme allows you to code strings of arbitrary length into large integers, and moreover—since large integers can be exponents just as easily as small ones can—it allows for recursive coding. In other words, strings can contain the integer codes for other strings, and this can go on indefinitely. An example: the list of strings "0", "(0)", and "((0))" is coded into the stupendously large integer

$$2^{2^1}3^{2^{11}3^15^{13}}5^{2^{11}3^{11}5^17^{13}11^{13}}$$

The proverbial "astute reader" might well have noticed a possible ambiguity: How can you tell if an integer is to be decomposed via prime factorization into other integers or to be left alone and interpreted as a code

for an atomic symbol? Gödel's simple but ingenious solution was to have all atomic symbols be represented by odd integers. How does that solve the matter? Easy: You know you should not factorize odd integers, and conversely, you should factorize even ones, and then do the same with all the exponents you get when you do so. Eventually, you will bottom out in a bunch of odd integers representing atomic symbols, and you will know which ones are grouped together to form larger chunks and how those chunks are nested.

With this beautifully polished scheme for encoding strings inside integers and thereby inside mathematical systems, Gödel had discovered a way of getting such a system to talk—in code—about itself. He had snuck self-reference into systems that were presumed to be as incapable of self-reference as are pencils of writing on themselves or garbage cans of containing themselves, and he had done so as wittily as the Greeks snuck a boatload of unacceptable soldiers into Troy, "encoded" as one single large acceptable structure.

Historically, the importance of Gödel's work was that it revealed a plethora of unexpected self-references (via his code, to be sure, but that fact in no way diminishes their effect) within the supposedly impregnable walls of Russell and Whitehead's Troy, *Principia Mathematica*. Now in Lisp, it's possible to construct and manipulate pieces of Lisp programs. The idea of *quoted code* is one of those deep ideas that make Lisp so appealing to AI people. Okay, but—when you have a system constructed expressly to have self-referential potential, the fact that it *has* self-referential structures will amaze no one. What is amazing and wonderful is when self-reference pops up inside the very fortress constructed expressly to keep it out! The repercussions of that are enormous.

One of the clear consequences of Gödel's revelation of this self-referential potential inside mathematical systems was that the same potential exists within any similar formalism, including computer languages. That is simply because computers can do all the standard arithmetic operations—at least in theory—with integers of unlimited size, and so coded representations of programs are being manipulated any time you are manipulating certain large integers. Of course, which program is being manipulated depends on what code you use. It was only after Gödel's work had been absorbed by a couple of generations of mathematicians, logicians, and computer people that the desirability of inserting the concept of quotation *directly* into a formal language became obvious. To be quite emphatic about it, however, this does not enhance the language's potential in any way, except that it makes certain constructions easier and more transparent. It was for this reason of transparency that Minsky made his remark.

Oh yes, I agree, Gödel's proof would have been easier, but by the time Gödel dreamt it up, it would have long since been discovered (and called "Snoddberger's proof") had Gödel been in a mindset where inventing Lisp

was natural. I'm all for counterfactuals, but I think one should be careful to slip things realistically.

* * *

After this diatribe, you will think I am crazy when I turn around and tell you: Gödel *did* invent Lisp! I am not trying to take anything away from John McCarthy, but if you look carefully at what Gödel did in his 1931 article, written roughly 15 years before the birth of computers and 27 years before the official birth of Lisp, you will see that he anticipated nearly all the central ideas of Lisp. We have already been through the fact that the central data structure of Lisp, the list, was at the core of Gödel's work. The crucial need to be able to distinguish between atoms and lists—something that modern-day implementors of Lisp systems have to worry about—was recognized and cleverly resolved by Gödel, in his odd-even distinction. The idea of quoting is, in essence, that of the Gödel code. And finally, what about recursive functions, the heart and soul of Lisp programming technique? That idea, too, is an indispensable part of Gödel's paper! This is the astounding truth.

The heart of Gödel's construction is a chain of 46 definitions of functions, each new one building on previous ones in a dizzying spire ascending toward one celestial goal: the definition of one very complex function of a single integer, which, given any input, either returns the value 1 or goes into an infinite loop. It returns 1 whenever the input integer is the Gödel number —the code number—of a theorem of *Principia Mathematica,* and it loops otherwise. This is Gödel's 46th function, which he named "Bew", short for "beweisbar", meaning "provable" in German.

If we could calculate the value of function 46 swiftly for any input, it would resolve for us any true-false question of mathematics that a full axiomatic system could resolve. All we would need to do is to write down the statement in question in the language of *Principia Mathematica,* code the resulting formula into its Gödel number (the most mechanical of tasks), and then call function 46 on that number. A result of 1 means *true,* looping forever means *false.* Do I hear the astute reader protesting again? All right, then: If we want to avoid any chance of having to wait forever to find out the answer, we can encode the *negation* of the statement in question into its Gödel number as well, and also call function 46 on this second number. We'll let the two calculations proceed simultaneously, and see which one says '1'. Now, as long as *Principia Mathematica* has either the statement or its negation as a theorem, one of the two calls on function 46 will return 1, while the other will presumably spin on unto eternity.

How does function 46 work? Oh, easy—by calling function 45 many times. And how does function 45 work? Well, it calls functions 44 and others, and they call previously defined functions, some of which call themselves (recursion!), and so on and so forth—all of these complex calls eventually bottoming out in calls to absolutely trivial functions such as "S", the

successor function, which returns the value 18 when you feed it the integer 17. In short, the evaluation of a high-numbered function in Gödel's paper sets in motion the calling of one subroutine after another in a hierarchical chain of command, in *precisely* the same way as my function **expand** called numerous lower-level functions which called others, and so on. Gödel's remarkable series of 46 function definitions is, in my book, the world's first serious computer program—and *it is in Lisp.* (The Norwegian mathematician Thoralf Skolem was the inventor of the study of recursive functions *theoretically,* but Gödel was the first person to use recursive functions *practically,* to build up functions of great complexity.)

It was for all these reasons that Minsky's pseudo-joke struck my intellectual funnybone. One answer I wanted to make was: "I disagree: Gödel *shouldn't* have and *couldn't* have invented Lisp, because his work was a necessary precursor to the invention of Lisp, and anyway, he was out to destroy *PM,* not Lisp." Another answer was: "Your innuendo is wrong, because any type of reference has to be grounded in a code, and Gödel's way of doing it involved no more coding machinery grounding its referential capacity than any other efficient way would have." The final answer I badly wanted to blurt out was: "No, no, no—Gödel *did* invent Lisp!" You see why I was tongue-tied?

* * *

One reason I mention all this about Gödel is that I wish to make some historical and philosophical points. There is another reason, however, and that is to point out that the ideas in Lisp are intimately related to the basic questions of metamathematics and metalogic, and these, translated into a more machine-oriented perspective, are none other than the basic questions of computability—perhaps the deepest questions of computer science. Michael Levin has even written an introduction to mathematical logic using Lisp, rather than a more traditional system, as its underlying formal system. For this type of reason, Lisp occupies a very special place inside computer science, and is not likely to go away for a very long time.

However . . . (you were waiting for this, weren't you?), there is a vast gulf between the issues that Lisp makes clear and easy, and the issues that confront one who would try to understand and model the human mind. The way I see it is in terms of *grain size.* To me, the thought that Lisp itself might be "more conducive" to good AI ideas than any other computer language is quite preposterous. In fact, such claims remind me of certain wonderfully romantic but woefully antic claims I have heard about the Hopi language. The typical one runs something like this: "Einstein should just have invented Hopi; then the discovery of his theory of relativity would have been much easier." The basis for this viewpoint is that Hopi, it is said, lacks terms for "absolute time" in it. Supposedly, Hopi (or a language with similar properties) would therefore be the ideal language in which to speak of relativity, since absolute time is abandoned in relativity.

This kind of claim was first put forth by the outstanding American linguist Edward Sapir, was later polished by his student Benjamin Whorf, and is usually known these days as the Sapir-Whorf hypothesis. (I have already made reference to this view in Chapters 7 and 8 on sexist language and imagery.) To state the Sapir-Whorf thesis explicitly: Language controls thought. A milder version of it would say: Language exerts a powerful influence upon thought.

In the case of computer languages, the Sapir-Whorf thesis would have to be interpreted as asserting that programmers in language *X* can think only in terms that language *X* furnishes them, and no others. Therefore, they are strapped in to certain ways of seeing the "world", and are prevented from seeing many ideas that programmers in language *L* can easily see. At least this is what Sapir-Whorf would have you believe. I will have none of it!

I do use Lisp, I do think it is very convenient and natural in many ways, I do advocate that anyone seriously interested in AI learn Lisp well; all this is true, but I do not think that deep study of Lisp is the royal road to AI any more than I think that deep study of bricks is the royal road to understanding architecture. Indeed, I would suggest that the raw materials to be found in Lisp are to AI what raw materials are to architecture: convenient building blocks out of which far larger structures are made.

It would be ridiculous for anyone to hope to acquire a deep understanding of what AI is all about without first having a clear, precise understanding of what computers are all about. I know of no shorter cut to that latter goal than the study of Lisp, and that is one reason Lisp is so good for AI students. Beginners in Lisp encounter, and are in a good position to understand, fundamental issues in computer science that even some advanced programmers in other languages may not have encountered or thought about. Such concepts as lists, recursion, side effects, quoting and evaluating pieces of code, and many others that I did not have the space to present in my three columns, are truly central to the understanding of the potential of computing machinery. Moreover, without languages that allow people to deal with such concepts directly, it would be next to impossible to make programs of subtlety, grace, and multi-level complexity. Therefore I advocate Lisp very strongly.

It would similarly be next to impossible to build structures of subtlety, grace, and multi-level complexity such as the Golden Gate Bridge and the Empire State Building out of bricks or stone. Until the use of steel as an architectural substrate was established, such constructions were unthinkable. Now we are in a position to erect buildings that use steel in even more sophisticated ways. But steel itself is not the source of great architects' inspiration; it is simply a liberator. Being a super-expert on steel may be of some use to an architect, but I would guess that being quite knowledgeable will suffice. After all, buildings are not just scaled-up girders. And so it is with Lisp and AI. Lisp is not the "language of thought" or the "language of the brain"—not by any stretch of the imagination. Lisp is,

however, a liberator. Being a super-expert on Lisp may be of some use to a person interested in computer models of mentality, but being quite knowledgeable will suffice. After all, minds are not just scaled-up Lisp functions.

Let me switch analogies. Is it possible for a novelist to conceive of plot ideas, characters, intrigues, emotions, and so on, without being channeled by her own or his own native language? Are the events that take place in, say, *Anna Karenina* specifically determined by the nature of the Russian language and the words that it furnished to Tolstoy? If that were the case, then of course the novel would be incomprehensible to people who do not know the Russian language. It would simply make no sense at all. But that is not even remotely the case. English-language readers have read that novel with great pleasure and have just as fully fathomed its psychological twists and turns as have Russian-language readers. The reason is that Tolstoy's mind was concerned with concepts that float far above the grain size of *any* human language. To think otherwise is to reduce Tolstoy to a mere syntactician, is to see Tolstoy as pushed around by low-level quirks and local flukes of his own language.

Now please understand, I am not by any means asserting that Tolstoy transcended his own culture and times; certainly he belongs to a particular era and a particular set of circumstances, and those facts flavor what he wrote. But "flavor" is the right word here. The essence of what he did—the meat of it, to prolong the "flavor" metaphor—is universal, and has to do with the fact that Tolstoy had profoundly tasted the human condition, had felt the pangs of many conflicting emotions in all sorts of ways. *That's* where the power of his writing comes from, not from the language he happened to inherit (otherwise, why wouldn't all Russians be great novelists?); *that's* why his novels survive translation not only into other languages (so they reach other cultures), but also into other eras, with different sensibilities. If Tolstoy manages to reach further into the human psyche than most other writers do, it is not the Russian language that deserves the credit, but Tolstoy's acute sensitivity and empathy for people.

The analogous statement could be made about AI programs and AI researchers. One could even mechanically substitute "AI program" for "novel", "Lisp" for "Russian", and—well, I have to admit that I would be hard pressed to come up with "the Tolstoy of AI". Oh, well. My point is simply that good AI researchers are not in any sense slaves to any language. Their ideas are as far removed from Lisp (or whatever language they program in, be it Lisp itself, a "super-Lisp" (such as *n*-Lisp for any value of *n*), Prolog, Smalltalk, and so on) as they are from English or from their favorite computer's hardware design. As an example, the AI program that has probably inspired me more than any other is the one called Hearsay-II, a speech-understanding program developed at Carnegie-Mellon University in the mid-1970's by a team headed up by D. Raj Reddy and Lee Erman. That program was written not in Lisp but in a language called SAIL, very

different in spirit from Lisp. Nonetheless, it could easily have been written in Lisp. The reason it doesn't matter is simply that the scientific questions of how the mind works are on a totally different level from the statements of any computer language. The ideas transcend the language.

* * *

To some, I may seem here to be flirting dangerously with an anti-mechanistic mysticism, but I hasten to say that that is far from the case. Quite the contrary. Still, I can see why people might at first suspect me of putting forth such a view. A programmer's instinct says that you can cumulatively build a system, encapsulating all the complexity of one layer into a few functions, then building the next layer up by exploiting the efficient and compact functions defined in the preceding layer. This hierarchical mode of buildup would seem to allow you to make arbitrarily complex actions be represented at the top level by very simple function calls. In other words, the functions at the top of the pyramid are like "cognitive" events, and as you move down the hierarchy of lower-level functions, you get increasingly many ever-dumber subroutines, until you bottom out in a myriad calls on trivial, "subcognitive" ones. All this sounds very biological —even tantalizingly close to being an entire resolution of the mind-brain problem. In fact, for a clear spelling-out of just that position, see Daniel Dennett's book *Brainstorms,* or perhaps worse, see parts of my own *Gödel, Escher, Bach*!

Yes, although I don't like to admit it, I too have been seduced by this recursive vision of mechanical mentality, resembling nothing so much as an army, with its millions of unconscious robot privates carrying out the desires of its top-level cognitive general, as conveyed to them by large numbers of obedient and semi-bright intermediaries. Probably my own strongest espousal of this view is found in Chapter X of *GEB,* where a subheading blared out, loud and clear, "AI Advances Are Language Advances". I was arguing there, in essence, for the orthodox AI position that if we could just find the right "superlanguage"—a language presumably several levels above Lisp, but built on top of it as Flumsy or SEARLE are built on top of it—then all would be peaches and cream. We would be able to program in the legendary "language of thought". AI programs would practically write themselves. Why, there would be so much intelligence *in the language itself* that we could just sit back and give the sketchiest of hints, and the computer would go off and do our tacit bidding!

This, in the opinion of my current self, is a crazy vision, and my reasons for thinking so are presented in Chapter 26, "Waking Up from the Boolean Dream". I am relieved that I spent a lot more time in *GEB* knocking down that orthodox vision of AI rather than propping it up. I argued then, as I still do now, that the top-level behavior of the overall system must emerge statistically from a myriad independent pieces, whose actions are almost as likely to cancel each other out as to be in phase with each other. This picture,

most forcefully presented in the dialogue *Prelude . . . Ant Fugue* and surrounding chapters, was the dominant view in that book, and it continues to be my view. In this book, it is put forth in Chapters 25 and 26.

In anticipation of those chapters, you might just ponder the following question. Why is it the case that, after all these millennia of using language, we haven't developed a single word for common remarks such as "Could you come over here and take a look at this?" Why isn't that thought expressed by some minuscule epigrammatic utterance such as "Cycohatalat"? Why are we not building new layer upon new layer of sophistication, so that each new generation can say things that no previous generation could have even conceived of? Of course, in some domains we are doing just that. The phrase "Lisp interpreter" is one that requires a great deal of spelling out for novices, but once it is understood, it is a very useful shorthand for getting across an extremely powerful set of ideas. All through science and other aspects of life, we are adding words and phrases. Acronyms such as "radar", "laser", "NATO", and "OPEC", as well as sets of initials such as "NYC", "ICBM", "MIT", "DNA", and "PC", are all very common and very wordlike.

Indeed, language does grow, but nonetheless, despite what might be considered an exponential explosion in the number of terms we have at our disposal, nobody writes a novel in one word. Nobody even writes a novel in a hundred words. As a matter of fact, novels these days are no shorter than they were 200 years ago. Nor are textbooks getting shorter. No matter how big an idea can be packed into a single word, the ideas that people want to put into novels and textbooks are on a totally different scale from that. Obviously, this is not claiming that language *cannot express* the ideas of a novel; it is simply saying that it takes a heap o' language to do so, no matter how the language is built. That is the issue of grain size that I alluded to before, and I feel that it is a deep and subtle issue that will come up more often as theoretical AI becomes more sophisticated.

* * *

Those interested in the Sapir-Whorf thesis might be interested to learn of Novelflo, a new pseudo-natural language invented by Rhoda Spark of Golden, Colorado. Novelflo is intended as a hypothetical extension of, or perhaps successor to, English. In particular, it is a language designed expressly for streamlining the writing of novels (or poetry). You write your novel in Novelflo in a tiny fraction of the number of words it would take in full English; then you feed it into Spark's copyrighted "Expandatron" program, which expands your concise Novelflo input into beautiful, flowing streams of powerful and evocative English prose. Or poetry, for that matter —you simply set some parameters defining the desired "shape" of the poem (sonnet, limerick, etc.; free verse or rhyme; and similar decisions), and out comes a beautifully polished poem in mere seconds. Some of Spark's advertised features for Novelflo are:

* Plot Enhancement Mechanisms (PEM's)
* Default Inheritance Assumptions
* Automatic Character Verification (checks for consistency of each character's character)
* Automatic Plot Verification (checks to be sure plot isn't self-contradictory—an indispensable tool for any novel writer)
* SVP's (Stereotype Violation Mechanisms)—allow you to override default assumptions with maximal ease)
* Sarcasm Facilitators (allows sarcasm to be easily constructed)
* VuSwap (so you can shift point of view effortlessly)
* AEP's (Atmosphere Evocation Phrases)—conjure up, in a few words, whole scenes, feelings, moods, that otherwise would take many pages if not full chapters

This entire *Post Scriptum,* incidentally, was written in Novelflo (it was my first attempt at using Spark's language), and before being expanded, it was only 114 words long! I must admit, it took me over 80 hours to compose those nuggets of ideas, but Spark assures me that one gets used to Novelflo quickly, and hours become minutes.

Spark is now hard at work on the successor to Novelflo, to be called Supernovelflo. The accompanying expansion program will be called, naturally enough, the Superexpandatron. Spark claims this advance will allow a further factor of compression of up to 100. (She did not inform me how the times for writing a given passage in Supernovelflo and Novelflo would compare.) Thus this whole *P.S.*—yes, *all* of it—would be a mere three words long, when written in Supernovelflo. It would run as follows (so she tells me):

SP 91pahC TM-foH

Now I'd call that jus-telegant!