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-lb 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- lc. 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 c played in the previous 63-disk maneuver. Figure 19-ld 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

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 do making use of hn:

If n = 1, then just carry that one disk directly from an to dn; otherwise, do the following three steps:

(1) move a tower of height n-1 from sin 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 do 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 do hn)

(cond ((eq n 1) (carry-one-disk sn dn)) (t (move-tower (subl n) sn hn dn) (carry-one-disk sn dn) (move-tower (subl n) hn do 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 (subl n) sn do 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

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 be ab ca cb ab or equally well 1213 23 12 3132 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 be 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 be) 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 do hn)

(cond ((eq n 1) (carry-one-disk sn dn))

(t (list (move-tower (subl n) sn hn dn) (carry-one-disk sn dn) (move-tower (subl n) hn do sn))))))

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

-> (move-tower 4 'a 'b 'c)

(((ac ab cb) ac (ba be 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 be 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 do hn)

(cond ((eq n 1) (carry-one-disk sn dn))

(t (append (move-tower (subl n) sn hn dn) (carry-one-disk sn dn) (move-tower (subl n) hn do 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 c be ab ca cb ab ac be be ca be ab ac be ab ca cb ab ca be be ca cb ab ac be ab ca cb ab ac be ba ca be ab ac be ba ca cb ab ca be ba ca be ab ac be ab ca cb ab ac be ba ca be ab ac be ab ca cb ab ca be ba ca cb ab ac be ab ca cb ab ca be ba ca be ab ac be ba ca cb ab ca be ba ca cb ab ac be ab ca cb ab ac be ba ca be ab ac be ab ca cb ab ca be be ca cb ab ac be ab ca cb ab ac be be ca be ab ac be ba ca cb ab ca be be ca be ab ac be ab ca cb ab ac be ba ca be ab ac be ba ca cb ab ca be be ca cb ab ac be ab ca cb ab ca be be ca be ab ac be be ca cb ab ca be be ca be ab ac be ab ca cb ab ac be be ca be ab ac be ab ca cb ab ca be be ca cb ab ac be ab ca cb ab ac be ba ca be ab ac be ba ca cb ab ca be ba ca be ab ac be Ab ca cb ab ac be ba ca be ab ac be ab ca cb ab ca be ba ca cb ab ac be ab ca cb ab ca be be ca be ab ac be be ca cb ab ca be be ca cb ab ac be ab ca cb ab ac be ba ca be ab ac be ab ca cb ab ca be be ca cb ab ac be ab ca cb' ab ca be ba ca be ab ac be ba ca cb ab ca be ba ca be ab ac be ab ca cb ab ac be be ca be ab ac be be ca cb ab ca be be ca cb ab ac be ab ca cb ab ca be ba ca be ab ac be ba ca cb ab ca be be ca cb ab ac be ab ca cb ab ac be be ca be ab ac be ab ca cb ab ca be ba ca cb ab ac be ab ca cb ab ac be be ca be ab ac be ba ca cb ab ca be ba ca be ab ac be ab ca cb ab ac be ba ca be ab ac be ab ca cb ab ca be ba ca cb ab ac be ab ca cb ab ca be ba ca be ab ac be ba ca cb ab ca be ba ca cb ab ac be ab ca cb ab ac be be ca be ab ac be ab ca cb ab ca be be ca cb ab ac be 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=l: 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, W embryonic case, occurs when n=0, and that otherwise, the nth' 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 (sub 1 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 (12 3 b 1 2 3) rather than ((1 2 3) b (12 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 (12 3 b 1 2 3) out of the simpler answer (b 1 2 3) by appending (12 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 consing b onto it. Why does one involve appending and the other involve consing? 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 month, 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 acronymone 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 boiler ("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 BASTAI:

belly all stuffed (tummy achel) 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 in expanding tomatoes 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 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 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 ((Iessp (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 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 (eccellentel) 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 sill) 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 (bastal) (basta! (bastal) (bastal) (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 evaled, 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 "consed up then handed to the Lisp genie for evaling. 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 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 facade? 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 facade 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 senselet 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, 1 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: "Godel 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 Godel'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 Godel 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 Godel invent the idea of direct self-reference, as in Lisp?" And this, it seemed to me, missed a crucial point about Godel's work, which is that it showed that self-reference can crop up even where it is totally unexpected and unwelcome. The power of Godel'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, Godel'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. Godel'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 Godel'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 parenthesis 13. 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 Godel's insight was to figure out how the system (and here I mean Principia Mathematica "and", as Godel'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 v Land!".") to the ability to talk about arbitrarily long utterances, and nested ones at that ("'Douglas Hofstadter wrote,"Paul Revere said, v Land¡. "." ').

Godel 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. Godel 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", 3', and 5's. In other words, the three-symbol string whose symbols individually are coded for by 1 1 and 1 and 13 is coded for in toto by the single integer 2"3'5'3. 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

221321131513521131151713111 3

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

446

for an atomic symbol? Godel'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, Godel 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 Godel'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 Al 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 Godel'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 Godel'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, Godel's proof would have been easier, but by the time Godel dreamt it up, it would have long since been discovered (and called "Snoddberger's proof") had Godel 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: Godel did invent Lisp! I am not trying to take anything away from John McCarthy, but if you look carefully at what Godel 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 Godel' 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

Godel, in his odd-even distinction. The idea of quoting is, in essence, that of the Godel code. And finally, what about recursive functions, the heart and soul of Lisp programming technique? That idea, too, is an indispensable part of Godel' s paper! This is the astounding truth.

The heart of Godel' 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 Godel number -the code number-of a theorem of Principia Mathematica, and it loops otherwise. This is Godel' 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 Godel 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 Godel 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 v l'. 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 Godel'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. Godel'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 Godel 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: Godel 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 Godel'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-Godel did invent Lisp!" You see why I was tongue-tied?

One reason I mention all this about Godel 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 Al 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 Tanguage 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 channelled 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 experienced 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 Godel, 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 Al 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 Al 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 Al 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 91 pahC TM-foH

Now I'd call that jus-telegant!