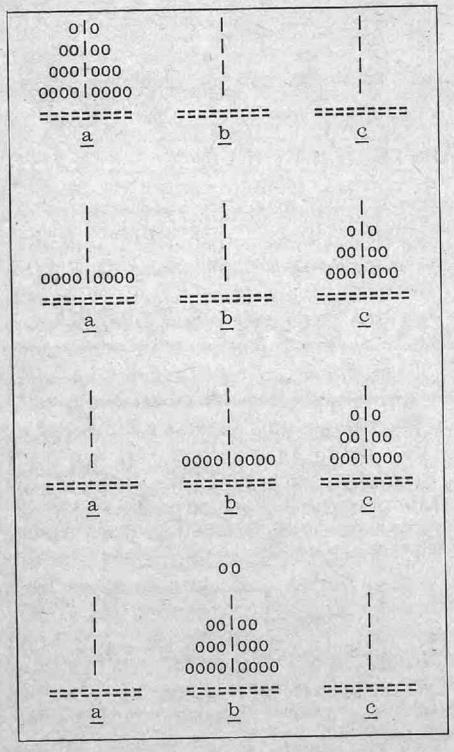
METAMAGICAL THEMAS

In which a discourse on the language Lisp concludes with a gargantuan Italian feast

by Douglas R. Hofstadter

Last month I described Édouard Lucas's Tower of Brahma puzzle, where 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. The disks must be picked up and moved one at a time, the only other constraint being that no disk may ever be placed on a smaller one. The problem I posed for readers was to come up with a recursive description, expressed as a function in the computer language Lisp, of how to accomplish this goal (and thereby, according to the priests of Brahma, end the world).

I pointed out that the recursion is evident enough: in order 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 recapitulate the idea, it is this. Suppose the 64-disk tower of Brahma starts out on needle *a*. At the top in the illustration on this page is a schematic picture representing all 64 disks by a mere 4. First of all, relying on your presumed 63-disk-moving ability, transfer 63 disks from needle *a* to needle *c*, using needle *b* as your "helping needle." How the setup looks is shown second from the top in the illustration. (Note that 4 plays the role of 64 in my original picture, so that 3 plays the role of 63, but for some reason 1 does not play the role of 61. Isn't that peculiar?) Now, simply pick up the one remaining *a* disk—the biggest disk of all—and plunk it down on needle *b*, as is shown third from the top.



The Tower of Brahma puzzle with four disks

Now you can see how easy it will be to finish up: simply reexploit your 63-disk ability 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 needle *c* played in the preceding 63-disk maneuver. A split second before the last disk is put in place the situation is as it is shown at the bottom in the illustration. Why before it is in place? Why not after? Because the entire world then turns to dust, and it is 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 may seem, but there is nothing magical about such an assumption. After all, in order to move 63 you 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 admit that you have to keep track of where you are in the process, which may be a bit tedious, but it is merely bookkeeping. In principle you now could actually carry out the entire process—if you were bent on seeing the world end!

As our first approximation of a Lisp function let us write an English description of the method. Let us 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 the steps labeled 1 and 3 are the two recursive calls; skirting paradox, they seem to call on the very ability they are helping to define. The saving feature is that they involve n − 1 disks instead of n. Note that in step 1 hn plays the "destination" role while dn plays the "helper" role. And in step 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 in the course of the transfers. That is the beauty of the puzzle and in a way it is also the beauty of recursion.

How do we make the transition from English to Lisp? It is 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. Then how can the Lisp genie know which needle is to play which role at each stage? The answer is that the information is conveyed *positionally*. There are, in this function definition, four parameters: one integer and three "dummy needles." The first of the three needles is the source, the second the destination and 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 step 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 step 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 defini-

tion. Hence it would have been a gross blunder to have written, for instance, "(move-tower (sub1 n) sn dn hn)" as Lisp for step 1, because that contains no indication that hn and dn must switch roles in that step.

An important question remains. What happens when the friendly Lisp genie comes to a step 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 it 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, to carry that object to another specific location and then to release it. In these days of industrial robots there is nothing science-fictional about it. In the absence of a mechanical arm and hand to move physical disks, however, what could it mean?

One obvious and closely related possibility is for there to 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. That would amount to simulating the puzzle graphically, which can be done with varying degrees of realism.

Suppose, however, we do not have fancy graphics hardware or software at our disposition. Suppose all we want is to create a printed recipe telling us how to move our soft, organic, human hands in order 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 to 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, in contrast, we are concerned only with the value 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 order to set groups apart in Lisp, rather than using wider spaces we use parentheses. Hence our goal for n = 3 might be to produce the sandwichlike list "((ab ac bc) ab (ca cb ab))". One way to produce a list out of its components is to use "cons" repeatedly. Therefore if the values of the atoms "apple", "banana" and "cherry" are 1, 2 and 3, then the value returned by "(cons apple (cons banana (cons cherry nil)))" will be the list "(1 2 3)". There is, however, 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 respectively to "a" and "b", 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 should like it to return a single atom made out of those needles' names, such as "ab" or "12". For the moment it will 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 bit of arithmetic: multiply the first number 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 non-numeric atoms, 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 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 easy. We simply want it to return a sandwichlike list in which the values of the two recursive calls flank the value of the single call to carry-one-disk. And 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 us 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 wanted 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. It would be slightly less informative but 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 would remove the parentheses from the two flanking recursive values.

This is a case where the standard Lisp function "append" comes in handy. It splices any number of lists together, dropping their outermost parentheses

```
(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 ba 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 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 cb
 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 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 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)
```

The Lisp genie fires back the moves to solve a 9-high Tower of Brahma puzzle

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" had 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. What it shot back at me, almost instantaneously, is at the top of the preceding page.

We 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 in the 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. How it goes is shown in the illustration below. For us the challenge is to write a Lisp function that returns the value of "tato" after n recursive expansions, for any n. It is irrelevant that for n much bigger than 3 the whole thing gets ridiculously large. We are theoreticians!

There is only one problem: any Lisp function must return a single Lisp structure (an atom or a list) as its value, and the entries in our table do not satisfy this criterion. For instance, the entry for n = 2 consists of one atom followed by two lists. To fix this we can turn each entry 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 equals 0, and that otherwise the nth line is made from the line before it by replacing the atom "tato", wherever it occurs, with the list "(tato (and tato only))", but without its outermost parentheses. We can write this up:

```
(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 that 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 line of the "tato" table and you will see that it contains one more element 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 above, are dropped in the act of replacement. It is 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 "(replace 'a '(1 2 3) '(a b a))", whose value should be "(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 inside a long list.

Let us try to specify in Lisp—using recursion as usual—just what we mean by "replacing" 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. Let me give you 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.

```
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)
```

A short table of recursive expansions of the Lisp variable designated "tato"

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

The recursive case involves building a more complex answer out of a simpler one assumed to be given. We can fall back on our "(a b a)" example for this. We built the complex answer "(1 2 3 b 1 2 3)" out of the simpler answer "(b 1 2 3)" by appending "(1 2 3)" to it. On the other hand, we could consider "(b 1 2 3)" itself to be a complex answer built out of the simpler answer "(1 2 3)" by consing "b" onto it. Why does one involve appending and the other involve consing? Simply because the first case involves the atom "a", which *does* get replaced, whereas the second involves the atom "b", which does *not* get replaced. This observation enables us to attempt to write down 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 us 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 nose, namely that the elements of longlist are always atoms. We ignored the possibility that longlist might contain sublists. What to do in such a case? The 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 definition?

We have seen a recursion before where 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 shall have a recur-

sive line featuring *two* calls on "replace", one involving the car of longlist and one involving the cdr, instead of just one involving the cdr. This makes perfect sense when you think about it. Suppose you wanted to replace all the unicorns in Europe with porpuquines. One way to achieve this nefarious goal would be to split Europe into two pieces: Portugal (Europe's car) and the remainder (Europe's cdr). After replacing all the unicorns in Portugal with porpuquines, and also all the unicorns in the rest of Europe with porpuquines, you would finally 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. That suggests 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 (ear 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 shall say that when longlist is not a list but an atom, "replace" has no effect on longlist at all. Conveniently this subsumes the earlier "null" line, and so we can drop that one. If we put all of 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 about *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 respectively for:

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

"luscious itty-bitty NOODLES got
usually in Naples, Italy"

Notice, incidentally, that "NOODLES" is not only indirectly recursive but also directly so. There is 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, this means a collection of words in any one of which letters 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. It is nonetheless 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."

Not being satisfied with that, I carried the entire process a 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. 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* (roughly "to walk while hobbling") 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 (although 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 within the phrases 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, Emmentaler
(particularly SHARP Emmentaler)

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:
LAMB CHOPS
(including NOODLES),
got 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
particularly (twist, then ingest)

COFFEE:
choice of fine flavors,
particularly ESPRESSO

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

BASTA!:
belly all stuffed (tummy ache!)

LAMB CHOPS:
LASAGNE and meatballs,
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.

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 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?

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. 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? It 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 comes as a consequence of the fact that teeny twigs simply cannot branch further. (Somehow in the course of evolution trees seem to have got their wires crossed, since for them bottoming out generally takes place at the top.)

If this process were completely regular, all redwood trees would look exactly alike and one could agree with former Governor Reagan's memorable dictum "If you've seen one redwood tree, you've seen them all." Unfortunately redwood trees (and maybe some other things) are trickier than Governor Reagan realized, and we have to learn to deal with a variety of things that go by the same name. The variety is caused by the introduction of randomness into the choices of 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 "tomatoes" we always made exactly the same control decisions about which acronyms to expand when, 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, 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 it? 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 a dice throw. We shall 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 shall 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 (no expansion). 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 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 for most observers the behavior of the function will be so erratic that it counts as being totally random. The story of random-number generation is quite a fascinating one and would be an article in itself.)

If we want an event to happen with a probability of 60 percent, first we ask rand for a value. If the value turns out to be .6 or below, we go ahead, otherwise we do not. Since over a long period of time rand sprays its outputs uniformly over the interval between 0 and 1, we shall indeed get the go-ahead 60 percent of the time.

So much for random decisions. How do we get an acronym to expand when it is told to? That 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 it expands into. Nothing could be simpler.

Now suppose we have a variable called "acronym" whose value is some particular acronym—but we do not know which one. How could we get that acronym to expand? The way we have set it up the 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)". If we were to write "(acronym)", the literal atom "acronym" would be taken by the Lisp genie

as a function name. That, however, would be a misunderstanding. It is 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.

In order to accomplish this we resort to a little trick. If the value of the atom "acronym" is "rhubarb" and if I write "(list acronym)", the value the genie will return to me will be the list "(rhubarb)". The genie, however, will simply see this as an inert piece of Lispstuff rather than as a little command I should like to have executed. It cannot read my mind. How do I get it to perform the desired operation? The answer is that I remember the function called "eval", which makes the genie look on a given data structure as a wish to be executed. In this case I need merely say "(eval (list acronym))" and I shall get the list "(ravioli, heated under butter and rhubarb (basta!))". And if the acronym had had a different value, 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 determined by many "flips" of the "rand" coin. Instead of stepping you through the construction of this function I shall simply display it 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 parameter represents the phrase to be expanded, 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. This is why lines that call for the 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 .8)))
```

Thus each time an acronym expands, its progeny are only .8 times as likely to expand as it was. This means acronyms nested deep enough have a vanishingly small probability of spawning further progeny. You could use any reducing factor; there is nothing sacred about .8 except that it seems to yield 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 it is not, the function returns nil. If the argument *is* an atom, the function goes on to see whether that atom has a function definition, in particular a definition with the form of an acronym. If the atom has such a definition, is-acronym returns the value t; otherwise it returns nil. Precisely how this is done depends on your specific variety of Lisp, which is why I have not shown it explicitly. In Franz Lisp only one line is needed.

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? Actually they belong to different conds, 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 it is 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, particularly excellent, strong, powerful, rich espresso, suppressing sleep outrageously (eccellente!) and green and red lasagne in cheese (noodles everywhere!) in cheddar, havarti, Emmentaler (particularly sharp Emmentaler) 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 meatballs, 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), got 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!))

Can you figure out which acronym this gastronomical monstrosity grew out of? As you can see, Lisp is hardly the computer language to learn if you want to lose weight. One final example of the glories of recursive spaghetti is given herewith, expanded from a different starting point:

(macaroni and cheese (a rather extraordinary pasta and sauce, typical of Naples, Italy) and cheddar, havarti, Emmentaler (particularly sharp Emmentaler) (a rather extraordinary pasta and shad and unusual coffee (eccellente!) (that's a luscious lunch) and sauce (that's all!) and shad and unusual choice of fine flavors, particularly espresso (eccellente!) (that's all!) and sauce (that's a luscious lunch) and spaghetti, heated al dente and unusual choice of fine flavors, particularly excellent, strong, powerful, rich espresso, suppressing sleep outrageously (eccellente!), typical of Naples, Italy))

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'd", as Lispers' jargon has it. The work involved in manufacturing the data structure was next to nothing in this instance, but in other instances elaborate pieces of structure can be "consed 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 an "animate agent," where it becomes a manipulator of inert structures itself. This program-data cycle, or loop, can con-

tinue on and on, with structures reaching out, twisting back and indirectly modifying themselves or structures related to them.

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, accessible by looking in an index in somewhat the way "book-learned" facts are accessible to a human being. In contrast, animate, or active, pieces of code are referred to as "procedural knowledge"—"procedural" because they define sequences of actions ("procedures") that actually manipulate data structures, and "knowledge" because they can be viewed as embodying the program's set of skills, something like a human being'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 the distinction between genes, which are 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 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 that one can describe with ease the Lisp genie's complete nature in Lisp itself. In other words, the Lisp interpreter can be easily written down in Lisp. Of course, if there is no Lisp interpreter to run *that* Lisp interpreter, 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. It is not, however, as 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 is more, that point is not too 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, as they read, they begin to learn new phrases all by themselves, simply by encountering 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 written in Lisp and usually is.

The fact that one can easily write the Lisp interpreter in Lisp is no mere fluke of some peculiarly introverted fact about Lisp. The reason it is easy is that, because of its program-data loop, Lisp lends itself to writing interpreters for all kinds of computer languages. This means 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, it should not be too hard for you to write an interpreter for it in Lisp. Once your Flumsy interpreter is implemented, it becomes in essence a new genie to which you can give wishes in Flumsy and that will in turn communicate those wishes to the Lisp genie in Lisp. Of course, all the mechanisms allowing the Flumsy "meta-genie" to talk with the Lisp genie are themselves being carried out by the Lisp genie. 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 with their Russian 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, on which the other language was built as a second one? And suppose you find out that actually the interpreter's native language was Lithuanian, that she learned English only as an adolescent and then learned Russian by taking high school classes taught in English? Will you then think 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 find that the Lisp interpreter is actually written in Pascal or some other computer language. And then someone could strip off the Pascal facade as well, revealing to you that all instructions are really being executed in *machine language,* so that you are fooling yourself completely if you think the machine is talking Flumsy, Lisp, Pascal or any other higher-level computer language!

When one interpreter runs on top of another, 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 with "someone" whose "native language" is Lisp. Similarly, when I am dealing with people, I seldom think about what their brains are composed of; I do not reduce them mentally 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 written in 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 (Chinese restaurants or how grammars work), how closely matched the given level'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 expressions, I shall probably opt for the Chinese viewpoint. And if I find out that the SEARLE system is in turn implemented on top of a Lisp system, then I have yet a third point of view to choose. And so on.

With interpreters stacked on interpreters, however, things rapidly become very inefficient. It is like running a motor on power coming from a series of electric generators, each generator being run on power coming from the preceding one: a good deal of power is lost 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 or SEARLE, if you want to have it at your disposal, is then built on top of this "virtual Lisp machine." Such indirectness can be annoyingly inefficient, causing your new virtual Flumsy machine or virtual SEARLE machine to run dozens of times slower than you would like.

There have been important hardware developments in the past several years, and now machines are available that are based on Lisp at the hardware level. This means that they "speak" Lisp in a somewhat deeper sense—let us say "more fluently"—than virtual Lisp machines. It also means that when you are working on such a machine, you are "swimming" in a Lisp environment. A Lisp environment goes considerably beyond what I have described so far, because 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 one another and with an overarching "Lisp philosophy."

Such machines, although they are still expensive and somewhat developmental, are rapidly becoming cheaper and more available. They are put out by various new companies such as LMI (LISP Machine, Inc.), Symbolics, Inc., both of Cambridge, Mass., and older companies such as Xerox. Lisp is also available on most personal computers; you need only look at any issue of the current microcomputer magazines to find advertisements for it.

Why, in conclusion, is Lisp popular in artificial intelligence? There is no single answer, or even a simple answer. Would it be valid to say Lisp is "the language of thought"? Certainly not. AI people might have said so 10 or 20 years ago, but few would be so bold today. One reason Lisp has remained so popular is, as I wrote in my first Lisp column, that Lisp is crisp. Indeed, Lisp can be so elegant that a good Lisp function can please a Lisper as a poem pleases a lover of poetry. My colleague Dan Friedman is in fact compiling a book titled *Lisp Poems,* and he urges interested readers to send him their most elegant efforts. His address is Computer Science Department, Indiana University, Bloomington, Ind. 47405.

What properties of Lisp, then, have made it central to AI research? I would say the following ones are nearly exhaustive:

1. Lisp is crisp.
2. Lisp is interactive.
3. Lisp is centered on the idea of lists and their manipulation, and lists are extremely flexible, fluid data structures.
4. Lisp code, having the same form as Lisp data, can easily be manufactured in Lisp and run.
5. Interpreters for new languages can easily be built and experimented with in Lisp.
6. "Swimming" in a Lisplike environment feels natural for many people.
7. Lisp is permeated by the "recursive spirit."

Perhaps it is this last rather intangible statement that gets at it the best. 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. It is a hunch, an intuition, a vaguely felt and slightly mystical belief, and one that I certainly share, but whether it will pay off in the long run remains to be seen.