

Pulitzer Prize-Winner

20th-anniversary Edition: With a new preface by the author



GÖDEL, ESCHER, BACH:

||||| *an Eternal Golden Braid* |||||

DOUGLAS R. HOFSTADTER

A metaphorical fugue on minds and machines in the spirit of Lewis Carroll

Recursive Structures and Processes

What Is Recursion?

WHAT IS RECURSION? It is what was illustrated in the Dialogue *Little Harmonic Labyrinth*: nesting, and variations on nesting. The concept is very general. (Stories inside stories, movies inside movies, paintings inside paintings, Russian dolls inside Russian dolls (even parenthetical comments inside parenthetical comments!)—these are just a few of the charms of recursion.) However, you should be aware that the meaning of “recursive” in this Chapter is only faintly related to its meaning in Chapter III. The relation should be clear by the end of this Chapter.

Sometimes recursion seems to brush paradox very closely. For example, there are *recursive definitions*. Such a definition may give the casual viewer the impression that something is being defined in terms of *itself*. That would be circular and lead to infinite regress, if not to paradox proper. Actually, a recursive definition (when properly formulated) never leads to infinite regress or paradox. This is because a recursive definition never defines something in terms of itself, but always in terms of *simpler versions* of itself. What I mean by this will become clearer shortly, when I show some examples of recursive definitions.

One of the most common ways in which recursion appears in daily life is when you postpone completing a task in favor of a simpler task, often of the same type. Here is a good example. An executive has a fancy telephone and receives many calls on it. He is talking to A when B calls. To A he says, “Would you mind holding for a moment?” Of course he doesn’t really care if A minds; he just pushes a button, and switches to B. Now C calls. The same deferment happens to B. This could go on indefinitely, but let us not get too bogged down in our enthusiasm. So let’s say the call with C terminates. Then our executive “pops” back up to B, and continues. Meanwhile, A is sitting at the other end of the line, drumming his fingernails against some table, and listening to some horrible Muzak piped through the phone lines to placate him . . . Now the easiest case is if the call with B simply terminates, and the executive returns to A finally. But it *could* happen that after the conversation with B is resumed, a new caller—D—calls. B is once again pushed onto the stack of waiting callers, and D is taken care of. After D is done, back to B, then back to A. This executive is hopelessly mechanical, to be sure—but we are illustrating recursion in its most precise form.

Pushing, Popping, and Stacks

In the preceding example, I have introduced some basic terminology of recursion—at least as seen through the eyes of computer scientists. The terms are *push*, *pop*, and *stack* (or *push-down stack*, to be precise) and they are all related. They were introduced in the late 1950's as part of IPL, one of the first languages for Artificial Intelligence. You have already encountered “push” and “pop” in the Dialogue. But I will spell things out anyway. To *push* means to suspend operations on the task you're currently working on, without forgetting where you are—and to take up a new task. The new task is usually said to be “on a lower level” than the earlier task. To *pop* is the reverse—it means to close operations on one level, and to resume operations exactly where you left off, one level higher.

But how do you remember exactly where you were on each different level? The answer is, you store the relevant information in a *stack*. So a stack is just a table telling you such things as (1) where you were in each unfinished task (jargon: the “return address”), (2) what the relevant facts to know were at the points of interruption (jargon: the “variable bindings”). When you pop back up to resume some task, it is the stack which restores your context, so you don't feel lost. In the telephone-call example, the stack tells you *who* is waiting on each different level, and *where* you were in the conversation when it was interrupted.

By the way, the terms “push”, “pop”, and “stack” all come from the visual image of cafeteria trays in a stack. There is usually some sort of spring underneath which tends to keep the topmost tray at a constant height, more or less. So when you push a tray onto the stack, it sinks a little—and when you remove a tray from the stack, the stack pops up a little.

One more example from daily life. When you listen to a news report on the radio, oftentimes it happens that they switch you to some foreign correspondent. “We now switch you to Sally Swumpley in Peafog, England.” Now Sally has got a tape of some local reporter interviewing someone, so after giving a bit of background, she plays it. “I'm Nigel Cadwallader, here on scene just outside of Peafog, where the great robbery took place, and I'm talking with . . .” Now you are three levels down. It may turn out that the interviewee also plays a tape of some conversation. It is not too uncommon to go down three levels in real news reports, and surprisingly enough, we scarcely have any awareness of the suspension. It is all kept track of quite easily by our subconscious mind. Probably the reason it is so easy is that each level is extremely different in flavor from each other level. If they were all similar, we would get confused in no time flat.

An example of a more complex recursion is, of course, our Dialogue. There, Achilles and the Tortoise appeared on all the different levels. Sometimes they were reading a story in which they appeared as characters. That is when your mind may get a little hazy on what's going on, and you have to concentrate carefully to get things straight. “Let's see, the *real* Achilles and Tortoise are still up there in Goodfortune's helicopter, but the

secondary ones are in some Escher picture—and then they found this book and are reading in it, so it's the *tertiary* Achilles and Tortoise who are wandering around inside the grooves of the *Little Harmonic Labyrinth*. No, wait a minute—I left out one level somewhere . . .” You have to have a conscious mental stack like this in order to keep track of the recursion in the Dialogue. (See Fig. 26.)

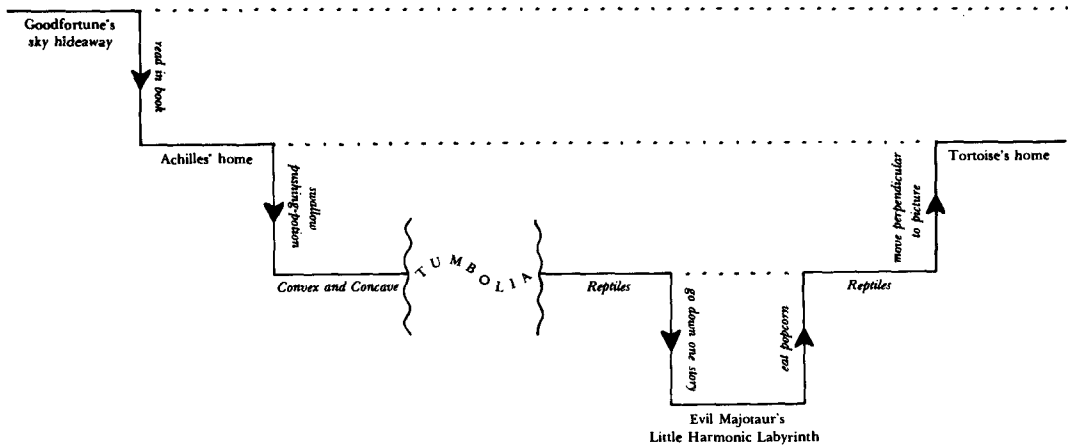


FIGURE 26. Diagram of the structure of the Dialogue Little Harmonic Labyrinth. Vertical descents are “pushes”; rises are “pops”. Notice the similarity of this diagram to the indentation pattern of the Dialogue. From the diagram it is clear that the initial tension—Goodfortune’s threat—never was resolved; Achilles and the Tortoise were just left dangling in the sky. Some readers might agonize over this unpopped push, while others might not bat an eyelash. In the story, Bach’s musical labyrinth likewise was cut off too soon—but Achilles didn’t even notice anything funny. Only the Tortoise was aware of the more global dangling tension.

Stacks in Music

While we’re talking about the *Little Harmonic Labyrinth*, we should discuss something which is hinted at, if not stated explicitly in the Dialogue: that we hear music recursively—in particular, that we maintain a mental stack of keys, and that each new modulation pushes a new key onto the stack. The implication is further that we want to hear that sequence of keys retraced in reverse order—popping the pushed keys off the stack, one by one, until the tonic is reached. This is an exaggeration. There is a grain of truth to it, however.

Any reasonably musical person automatically maintains a shallow stack with two keys. In that “short stack”, the true tonic key is held, and also the most immediate “pseudotonic” (the key the composer is pretending to be in). In other words, the most global key and the most local key. That way, the listener knows when the true tonic is regained, and feels a strong sense of “relief”. The listener can also distinguish (unlike Achilles) between a *local* easing of tension—for example a resolution into the pseudotonic—

and a *global* resolution. In fact, a pseudoresolution should heighten the global tension, not relieve it, because it is a piece of irony—just like Achilles' rescue from his perilous perch on the swinging lamp, when all the while you know he and the Tortoise are really awaiting their dire fates at the knife of Monsieur Goodfortune.

Since tension and resolution are the heart and soul of music, there are many, many examples. But let us just look at a couple in Bach. Bach wrote many pieces in an "*AABB*" form—that is, where there are two halves, and each one is repeated. Let's take the *gigue* from the French Suite no. 5, which is quite typical of the form. Its tonic key is G, and we hear a gay dancing melody which establishes the key of G strongly. Soon, however, a modulation in the *A*-section leads to the closely related key of D (the dominant). When the *A*-section ends, we are in the key of D. In fact, it sounds as if the piece has ended in the key of D! (Or at least it might sound that way to Achilles.) But then a strange thing happens—we abruptly jump back to the beginning, back to G, and rehear the same transition into D. But then a strange thing happens—we abruptly jump back to the beginning, back to G, and rehear the same transition into D.

Then comes the *B*-section. With the inversion of the theme for our melody, we begin in D as if that had always been the tonic—but we modulate back to G after all, which means that we pop back into the tonic, and the *B*-section ends properly. Then that funny repetition takes place, jerking us without warning back into D, and letting us return to G once more. Then that funny repetition takes place, jerking us without warning back into D, and letting us return to G once more.

The psychological effect of all this key shifting—some jerky, some smooth—is very difficult to describe. It is part of the magic of music that we can automatically make sense of these shifts. Or perhaps it is the magic of Bach that he can write pieces with this kind of structure which have such a natural grace to them that we are not aware of exactly what is happening.

The original *Little Harmonic Labyrinth* is a piece by Bach in which he tries to lose you in a labyrinth of quick key changes. Pretty soon you are so disoriented that you don't have any sense of direction left—you don't know where the true tonic is, unless you have perfect pitch, or like Theseus, have a friend like Ariadne who gives you a thread that allows you to retrace your steps. In this case, the thread would be a written score. This piece—another example is the *Endlessly Rising Canon*—goes to show that, as music listeners, we don't have very reliable deep stacks.

Recursion in Language

Our mental stacking power is perhaps slightly stronger in language. The grammatical structure of all languages involves setting up quite elaborate push-down stacks, though, to be sure, the difficulty of understanding a sentence increases sharply with the number of pushes onto the stack. The proverbial German phenomenon of the "verb-at-the-end", about which

droll tales of absentminded professors who would begin a sentence, ramble on for an entire lecture, and then finish up by rattling off a string of verbs by which their audience, for whom the stack had long since lost its coherence, would be totally nonplussed, are told, is an excellent example of linguistic pushing and popping. The confusion among the audience that out-of-order popping from the stack onto which the professor's verbs had been pushed, is amusing to imagine, could engender. But in normal spoken German, such deep stacks almost never occur—in fact, native speakers of German often unconsciously violate certain conventions which force the verb to go to the end, in order to avoid the mental effort of keeping track of the stack. Every language has constructions which involve stacks, though usually of a less spectacular nature than German. But there are always ways of rephrasing sentences so that the depth of stacking is minimal.

Recursive Transition Networks

The syntactical structure of sentences affords a good place to present a way of describing recursive structures and processes: the *Recursive Transition Network* (RTN). An RTN is a diagram showing various paths which can be followed to accomplish a particular task. Each path consists of a number of *nodes*, or little boxes with words in them, joined by *arcs*, or lines with arrows. The overall name for the RTN is written separately at the left, and the first and last nodes have the words *begin* and *end* in them. All the other nodes contain either very short explicit directions to perform, or else names of other RTN's. Each time you hit a node, you are to carry out the directions inside it, or to jump to the RTN named inside it, and carry it out.

Let's take a sample RTN, called **ORNATE NOUN**, which tells how to construct a certain type of English noun phrase. (See Fig. 27a.) If we traverse **ORNATE NOUN** purely horizontally, we *begin*, then we create an **ARTICLE**, an **ADJECTIVE**, and a **NOUN**, then we *end*. For instance, "the silly shampoo" or "a thankless brunch". But the arcs show other possibilities, such as skipping the article, or repeating the adjective. Thus we could construct "milk", or "big red blue green sneezes", etc.

When you hit the node **NOUN**, you are asking the unknown black box called **NOUN** to fetch any noun for you from its storehouse of nouns. This is known as a *procedure call*, in computer science terminology. It means you temporarily give control to a *procedure* (here, **NOUN**) which (1) does its thing (produces a noun) and then (2) hands control back to you. In the above RTN, there are calls on three such procedures: **ARTICLE**, **ADJECTIVE**, and **NOUN**. Now the RTN **ORNATE NOUN** could itself be called from some other RTN—for instance an RTN called **SENTENCE**. In this case, **ORNATE NOUN** would produce a phrase such as "the silly shampoo" and then return to the place inside **SENTENCE** from which it had been called. It is quite reminiscent of the way in which you resume where you left off in nested telephone calls or nested news reports.

However, despite calling this a "recursive transition network", we have

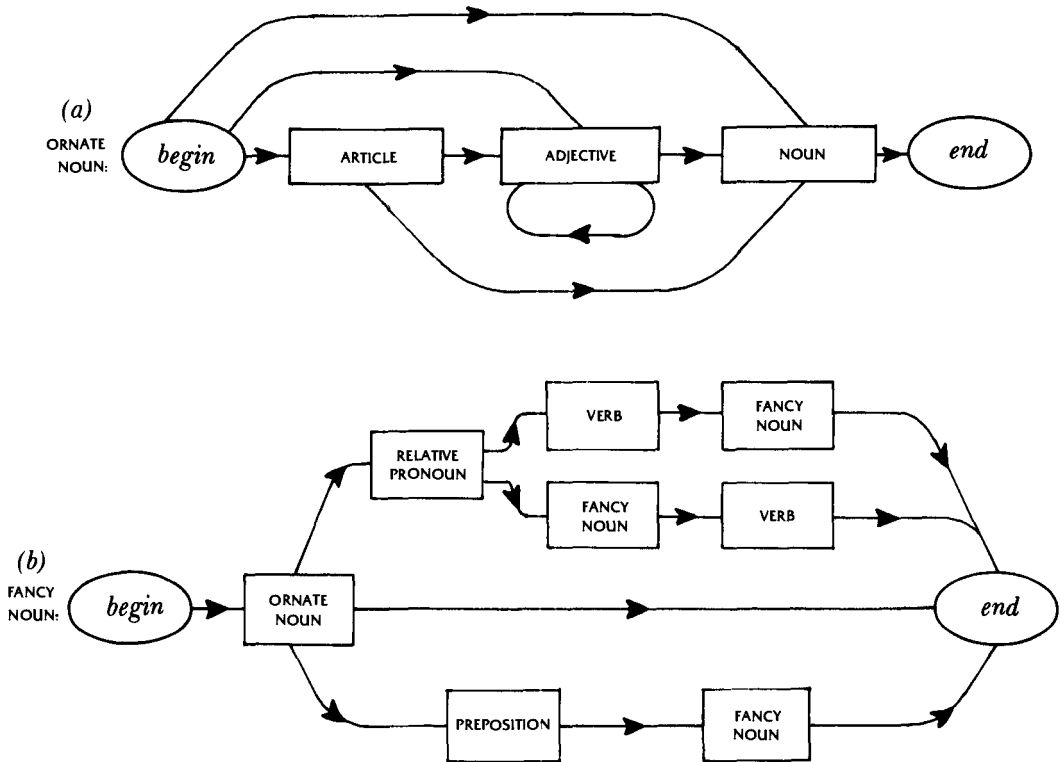


FIGURE 27. Recursive Transition Networks for ORNATE NOUN and FANCY NOUN.

not exhibited any true recursion so far. Things get recursive—and seemingly circular—when you go to an RTN such as the one in Figure 27b, for FANCY NOUN. As you can see, every possible pathway in FANCY NOUN involves a call on ORNATE NOUN, so there is no way to avoid getting a noun of some sort or other. And it is possible to be no more ornate than that, coming out merely with “milk” or “big red blue green sneezes”. But three of the pathways involve *recursive* calls on FANCY NOUN itself. It certainly looks as if something is being defined in terms of itself. Is that what is happening, or not?

The answer is “yes, but benignly”. Suppose that, in the procedure SENTENCE, there is a node which calls FANCY NOUN, and we hit that node. This means that we commit to memory (*viz.*, the stack) the location of that node inside SENTENCE, so we’ll know where to return to—then we transfer our attention to the procedure FANCY NOUN. Now we must choose a pathway to take, in order to generate a FANCY NOUN. Suppose we choose the lower of the upper pathways—the one whose calling sequence goes:

ORNATE NOUN; RELATIVE PRONOUN; FANCY NOUN; VERB.

So we spit out an **ORNATE NOUN**: “*the strange bagels*”; a **RELATIVE PRO-NOUN**: “*that*”; and now we are suddenly asked for a **FANCY NOUN**. But we are in the middle of **FANCY NOUN**! Yes, but remember our executive who was in the middle of one phone call when he got another one. He merely stored the old phone call’s status on a stack, and began the new one as if nothing were unusual. So we shall do the same.

We first write down in our stack the node we are at in the outer call on **FANCY NOUN**, so that we have a “return address”; then we jump to the beginning of **FANCY NOUN** as if nothing were unusual. Now we have to choose a pathway again. For variety’s sake, let’s choose the lower pathway: **ORNATE NOUN**; **PREPOSITION**; **FANCY NOUN**. That means we produce an **ORNATE NOUN** (say “*the purple cow*”), then a **PREPOSITION** (say “*without*”), and once again, we hit the recursion. So we hang onto our hats, and descend one more level. To avoid complexity, let’s assume that this time, the pathway we take is the direct one—just **ORNATE NOUN**. For example, we might get “*horns*”. We hit the node **END** in this call on **FANCY NOUN**, which amounts to popping out, and so we go to our stack to find the return address. It tells us that we were in the middle of executing **FANCY NOUN** one level up—and so we resume there. This yields “*the purple cow without horns*”. On this level, too, we hit **END**, and so we pop up once more, this time finding ourselves in need of a **VERB**—so let’s choose “*gobbled*”. This ends the highest-level call on **FANCY NOUN**, with the result that the phrase

“the strange bagels that the purple cow without horns gobbled”

will get passed upwards to the patient **SENTENCE**, as we pop for the last time.

As you see, we didn’t get into any infinite regress. The reason is that at least one pathway inside the RTN **FANCY NOUN** does *not* involve any recursive calls on **FANCY NOUN** itself. Of course, we could have perversely insisted on always choosing the bottom pathway inside **FANCY NOUN**, and then we would never have gotten finished, just as the acronym “**GOD**” never got fully expanded. But if the pathways are chosen at random, then an infinite regress of that sort will not happen.

“Bottoming Out” and Heterarchies

This is the crucial fact which distinguishes recursive definitions from circular ones. There is always some part of the definition which avoids self-reference, so that the action of constructing an object which satisfies the definition will eventually “bottom out”.

Now there are more oblique ways of achieving recursivity in RTN’s than by self-calling. There is the analogue of Escher’s *Drawing Hands* (Fig. 135), where each of two procedures calls the other, but not itself. For example, we could have an RTN named **CLAUSE**, which calls **FANCY NOUN** whenever it needs an object for a transitive verb, and conversely, the upper path of **FANCY NOUN** could call **RELATIVE PRONOUN** and then **CLAUSE**

whenever it wants a relative clause. This is an example of *indirect* recursion. It is reminiscent also of the two-step version of the Epimenides paradox.

Needless to say, there can be a trio of procedures which call one another, cyclically—and so on. There can be a whole family of RTN's which are all tangled up, calling each other and themselves like crazy. A program which has such a structure in which there is no single “highest level”, or “monitor”, is called a *heterarchy* (as distinguished from a hierarchy). The term is due, I believe, to Warren McCulloch, one of the first cyberneticists, and a reverent student of brains and minds.

Expanding Nodes

One graphic way of thinking about RTN's is this. Whenever you are moving along some pathway and you hit a node which calls on an RTN, you “expand” that node, which means to replace it by a very small copy of the RTN it calls (see Fig. 28). Then you proceed into the very small RTN!

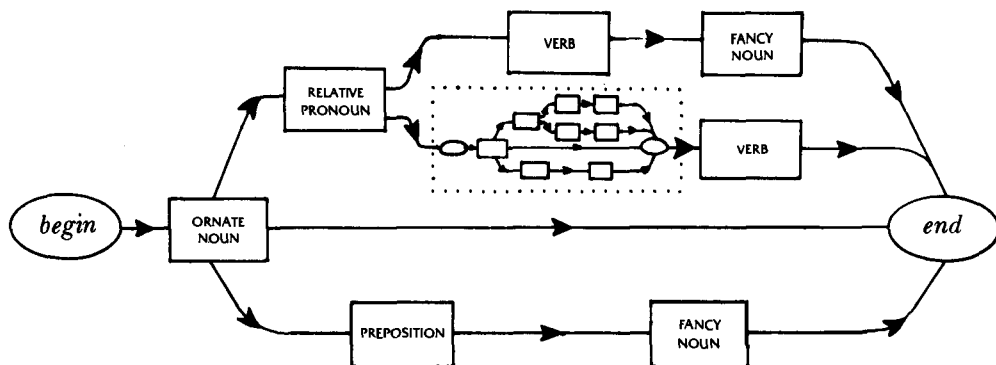


FIGURE 28. The FANCY NOUN RTN with one node recursively expanded.

When you pop out of it, you are automatically in the right place in the big one. While in the small one, you may wind up constructing even more miniature RTN's. But by expanding nodes only when you come across them, you avoid the need to make an infinite diagram, even when an RTN calls itself.

Expanding a node is a little like replacing a letter in an acronym by the word it stands for. The “GOD” acronym is recursive but has the defect—or advantage—that you must repeatedly expand the ‘G’; thus it never bottoms out. When an RTN is implemented as a real computer program, however, it always has at least one pathway which avoids recursivity (direct or indirect) so that infinite regress is not created. Even the most heterarchical program structure bottoms out—otherwise it couldn't run! It would just be constantly expanding node after node, but never performing any action.

Diagram G and Recursive Sequences

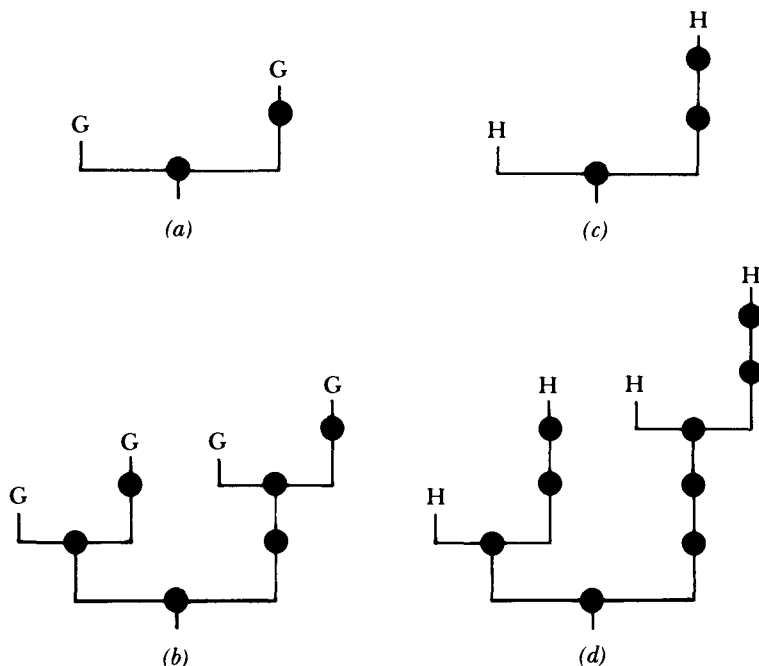
Infinite geometrical structures can be defined in just this way—that is, by expanding node after node. For example, let us define an infinite diagram called “Diagram G”. To do so, we shall use an implicit representation. In two nodes, we shall write merely the letter ‘G’, which, however, will stand for an entire copy of Diagram G. In Figure 29a, Diagram G is portrayed implicitly. Now if we wish to see Diagram G more explicitly, we expand each of the two G’s—that is, we *replace them by the same diagram*, only reduced in scale (see Fig. 29b). This “second-order” version of Diagram G gives us an inkling of what the final, impossible-to-realize Diagram G really looks like. In Figure 30 is shown a larger portion of Diagram G, where all the nodes have been numbered from the bottom up, and from left to right. Two extra nodes—numbers 1 and 2—have been inserted at the bottom.

This infinite *tree* has some very curious mathematical properties. Running up its right-hand edge is the famous sequence of *Fibonacci numbers*:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ...

discovered around the year 1202 by Leonardo of Pisa, son of Bonaccio, ergo “Filius Bonacci”, or “Fibonacci” for short. These numbers are best

FIGURE 29. (a) Diagram G, unexpanded. (c) Diagram H, unexpanded.
(b) Diagram G, expanded once. (d) Diagram H, expanded once.



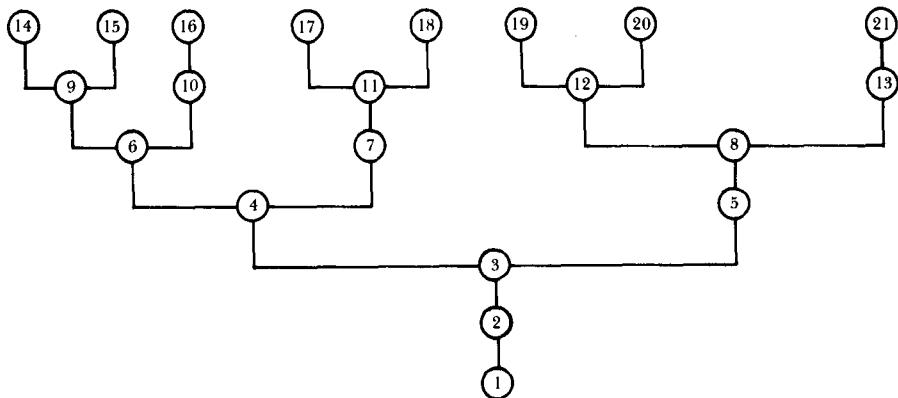


FIGURE 30. Diagram G, further expanded and with numbered nodes.

defined recursively by the pair of formulas

$$\text{FIBO}(n) = \text{FIBO}(n-1) + \text{FIBO}(n-2) \quad \text{for } n > 2$$

$$\text{FIBO}(1) = \text{FIBO}(2) = 1$$

Notice how new Fibonacci numbers are defined in terms of previous Fibonacci numbers. We could represent this pair of formulas in an RTN (see Fig. 31).

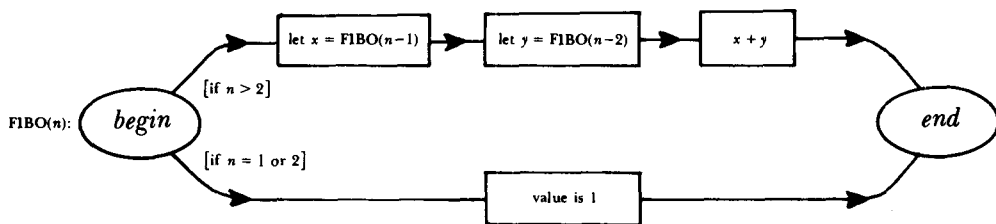


FIGURE 31. An RTN for Fibonacci numbers.

Thus you can calculate $\text{FIBO}(15)$ by a sequence of recursive calls on the procedure defined by the RTN above. This recursive definition bottoms out when you hit $\text{FIBO}(1)$ or $\text{FIBO}(2)$ (which are given explicitly) after you have worked your way backwards through descending values of n . It is slightly awkward to work your way backwards, when you could just as well work your way forwards, starting with $\text{FIBO}(1)$ and $\text{FIBO}(2)$ and always adding the most recent two values, until you reach $\text{FIBO}(15)$. That way you don't need to keep track of a stack.

Now Diagram G has some even more surprising properties than this. Its entire structure can be coded up in a single recursive definition, as follows:

$$G(n) = n - G(G(n-1)) \quad \text{for } n > 0$$

$$G(0) = 0$$

How does this function $G(n)$ code for the tree-structure? Quite simply, if you construct a tree by placing $G(n)$ below n , for all values of n , you will recreate Diagram G. In fact, that is how I discovered Diagram G in the first place. I was investigating the *function* G , and in trying to calculate its values quickly, I conceived of displaying the values I already knew in a tree. To my surprise, the tree turned out to have this extremely orderly recursive geometrical description.

What is more wonderful is that if you make the analogous tree for a function $H(n)$ defined with one more nesting than G —

$$H(n) = n - H(H(H(n-1))) \quad \text{for } n > 0$$

$$H(0) = 0$$

—then the associated “Diagram H” is defined implicitly as shown in Figure 29c. The right-hand trunk contains one more node; that is the only difference. The first recursive expansion of Diagram H is shown in Figure 29d. And so it goes, for any degree of nesting. There is a beautiful regularity to the recursive geometrical structures, which corresponds precisely to the recursive algebraic definitions.

A problem for curious readers is: suppose you flip Diagram G around as if in a mirror, and label the nodes of the new tree so they increase from left to right. Can you find a recursive *algebraic* definition for this “flip-tree”? What about for the “flip” of the H-tree? Etc.?

Another pleasing problem involves a pair of recursively intertwined functions $F(n)$ and $M(n)$ —“married” functions, you might say—defined this way:

$$\left. \begin{aligned} F(n) &= n - M(F(n-1)) \\ M(n) &= n - F(M(n-1)) \end{aligned} \right\} \quad \text{for } n > 0$$

$$F(0) = 1, \quad \text{and} \quad M(0) = 0.$$

The RTN’s for these two functions call each other and themselves as well. The problem is simply to discover the recursive structures of Diagram F and Diagram M. They are quite elegant and simple.

A Chaotic Sequence


One last example of recursion in number theory leads to a small mystery. Consider the following recursive definition of a function:


$$Q(n) = Q(n - Q(n-1)) + Q(n - Q(n-2)) \quad \text{for } n > 2$$

$$Q(1) = Q(2) = 1.$$

It is reminiscent of the Fibonacci definition in that each new value is a sum of two previous values—but not of the *immediately* previous two values. Instead, the two immediately previous values tell *how far to count back* to obtain the numbers to be added to make the new value! The first 17 Q-numbers run as follows:

1, 1, 2, 3, 3, 4, 5, 5, 6, 6, 6, 8, 8, 8, 10, 9, 10, ...

$\uparrow \quad \uparrow$
 $5 + 6 = 11$

 new term


 how far to move
to the left

To obtain the next one, move leftwards (from the three dots) respectively 10 and 9 terms; you will hit a 5 and a 6, shown by the arrows. Their sum—11—yields the new value: Q(18). This is the strange process by which the list of known Q-numbers is used to extend itself. The resulting sequence is, to put it mildly, erratic. The further out you go, the less sense it seems to make. This is one of those very peculiar cases where what seems to be a somewhat natural definition leads to extremely puzzling behavior: chaos produced in a very orderly manner. One is naturally led to wonder whether the apparent chaos conceals some subtle regularity. Of course, by definition, there is regularity, but what is of interest is whether there is another way of characterizing this sequence—and with luck, a nonrecursive way.

Two Striking Recursive Graphs

The marvels of recursion in mathematics are innumerable, and it is not my purpose to present them all. However, there are a couple of particularly striking examples from my own experience which I feel are worth presenting. They are both graphs. One came up in the course of some number-theoretical investigations. The other came up in the course of my Ph.D. thesis work, in solid state physics. What is truly fascinating is that the graphs are closely related.

The first one (Fig. 32) is a graph of a function which I call $\text{INT}(x)$. It is plotted here for x between 0 and 1. For x between any other pair of integers n and $n + 1$, you just find $\text{INT}(x - n)$, then add n back. The structure of the plot is quite jumpy, as you can see. It consists of an infinite number of curved pieces, which get smaller and smaller towards the corners—and incidentally, less and less curved. Now if you look closely at each such piece, you will find that it is actually a copy of the full graph, merely curved! The implications are wild. One of them is that the graph of INT consists of nothing but copies of itself, nested down infinitely deeply. If you pick up any piece of the graph, no matter how small, you are holding a complete copy of the whole graph—in fact, infinitely many copies of it!

The fact that INT consists of nothing but copies of itself might make you think it is too ephemeral to exist. Its definition sounds too circular.

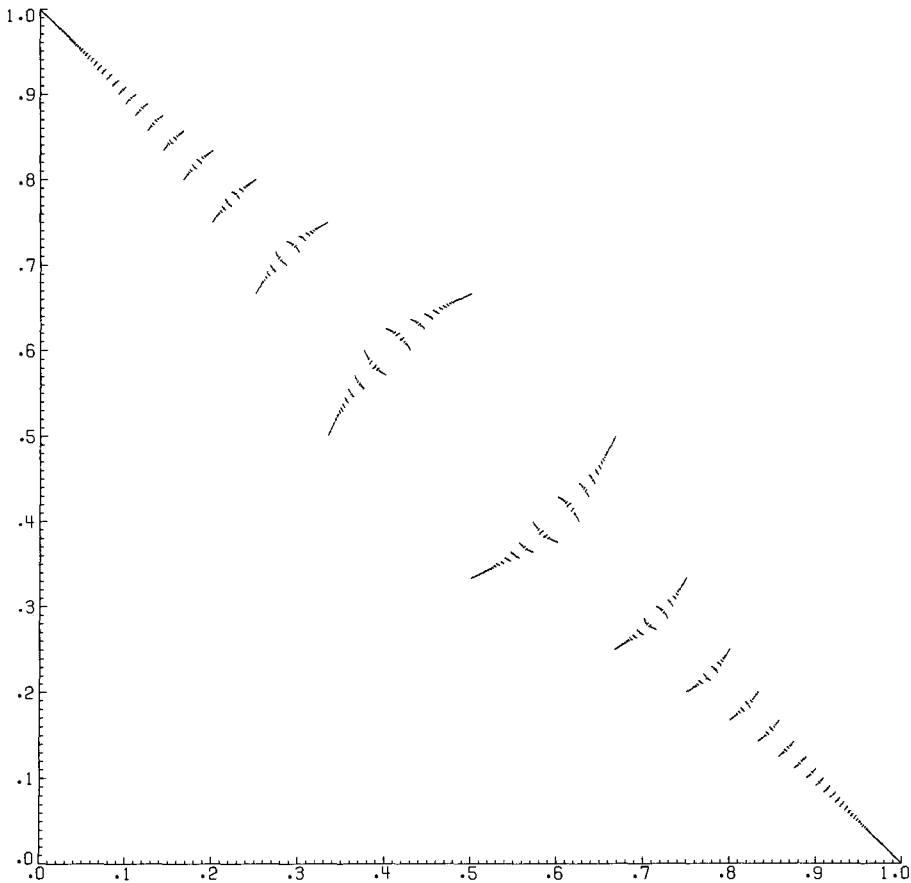


FIGURE 32. Graph of the function $\text{INT}(x)$. There is a jump discontinuity at every rational value of x .

How does it ever get off the ground? That is a very interesting matter. The main thing to notice is that, to describe INT to someone who hasn't seen it, it will not suffice merely to say, "It consists of copies of itself." The other half of the story—the nonrecursive half—tells *where* those copies lie inside the square, and *how* they have been deformed, relative to the full-size graph. Only the combination of these two aspects of INT will specify the structure of INT . It is exactly as in the definition of Fibonacci numbers, where you need two lines—one to define the *recursion*, the other to define the *bottom* (i.e., the values at the beginning). To be very concrete, if you make one of the bottom values 3 instead of 1, you will produce a completely different sequence, known as the *Lucas sequence*:

$$\begin{array}{ccccccccccc}
 1, & 3, & 4, & 7, & 11, & 18, & 29, & 47, & 76, & 123, & \dots \\
 \underbrace{\hspace{1.5cm}} & & & & & & & & & & \\
 \text{the "bottom"} & & & & & & 29 + 47 = 76 & & & & \\
 & & & & & & \text{same recursive rule} & & & & \\
 & & & & & & \text{as for the Fibonacci numbers} & & & &
 \end{array}$$

What corresponds to the *bottom* in the definition of INT is a picture (Fig. 33a) composed of many boxes, showing *where* the copies go, and *how* they are distorted. I call it the “skeleton” of INT. To construct INT from its skeleton, you do the following. First, for each box of the skeleton, you do two operations: (1) put a small curved copy of the skeleton inside the box, using the curved line inside it as a guide; (2) erase the containing box and its curved line. Once this has been done for each box of the original skeleton, you are left with many “baby” skeletons in place of one big one. Next you repeat the process one level down, with all the baby skeletons. Then again, again, and again . . . What you approach in the limit is an exact graph of INT, though you never get there. By nesting the skeleton inside itself over and over again, you gradually construct the graph of INT “from out of nothing”. But in fact the “nothing” was not nothing—it was a picture.

To see this even more dramatically, imagine keeping the recursive part of the definition of INT, but changing the initial picture, the skeleton. A variant skeleton is shown in Figure 33b, again with boxes which get smaller and smaller as they trail off to the four corners. If you nest this second skeleton inside itself over and over again, you will create the key graph from my Ph.D. thesis, which I call *Gplot* (Fig. 34). (In fact, some complicated distortion of each copy is needed as well—but nesting is the basic idea.) *Gplot* is thus a member of the INT-family. It is a distant relative, because its skeleton is quite different from—and considerably more complex than—that of INT. However, the recursive part of the definition is identical, and therein lies the family tie.

I should not keep you too much in the dark about the origin of these beautiful graphs. INT—standing for “interchange”—comes from a problem involving “Eta-sequences”, which are related to continued fractions. The basic idea behind INT is that plus and minus signs are interchanged in a certain kind of continued fraction. As a consequence, $\text{INT}(\text{INT}(x)) = x$. INT has the property that if x is rational, so is $\text{INT}(x)$; if x is quadratic, so is $\text{INT}(x)$. I do not know if this trend holds for higher algebraic degrees. Another lovely feature of INT is that at all rational values of x , it has a jump discontinuity, but at all irrational values of x , it is continuous.

Gplot comes from a highly idealized version of the question, “What are the allowed energies of electrons in a crystal in a magnetic field?” This problem is interesting because it is a cross between two very simple and fundamental physical situations: an electron in a perfect crystal, and an electron in a homogeneous magnetic field. These two simpler problems are both well understood, and their characteristic solutions seem almost incompatible with each other. Therefore, it is of quite some interest to see how nature manages to reconcile the two. As it happens, the crystal-without-magnetic-field situation and the magnetic-field-without-crystal situation do have one feature in common: in each of them, the electron behaves periodically in time. It turns out that when the two situations are combined, the ratio of their two time periods is the key parameter. In fact, that ratio holds all the information about the distribution of allowed electron energies—but it only gives up its secret upon being expanded into a continued fraction.

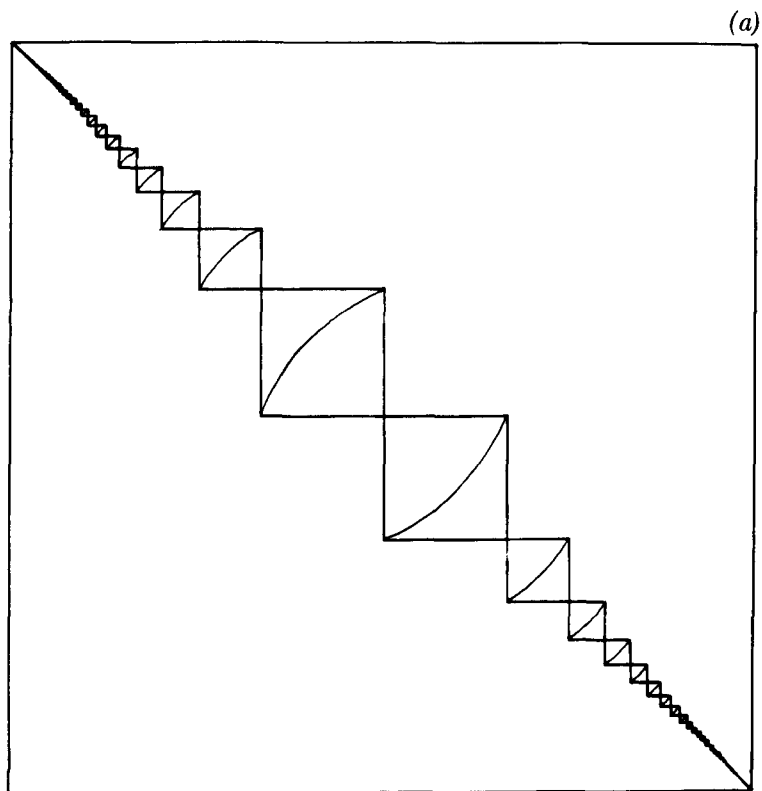
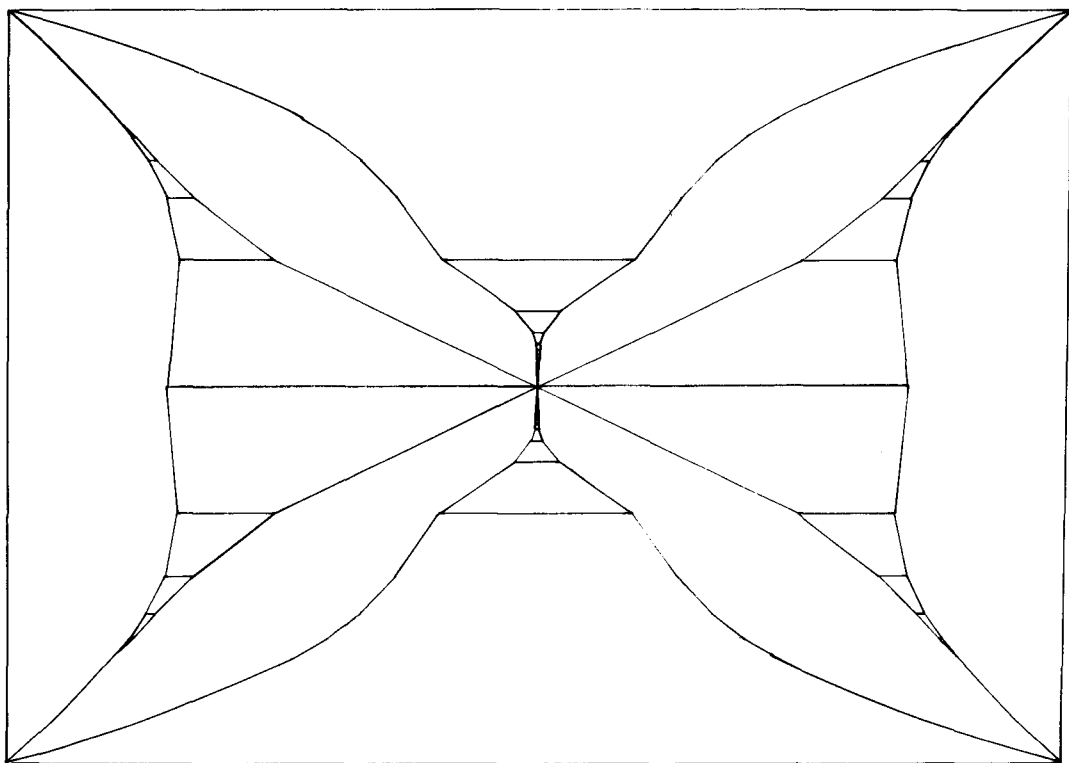


FIGURE 33(a) *The skeleton from which INT can be constructed by recursive substitutions.*
 (b) *The skeleton from which Gplot can be constructed by recursive substitutions.*



Gplot shows that distribution. The horizontal axis represents energy, and the vertical axis represents the above-mentioned ratio of time periods, which we can call " α ". At the bottom, α is zero, and at the top α is unity. When α is zero, there is no magnetic field. Each of the line segments making up Gplot is an "energy band"—that is, it represents allowed values of energy. The empty swaths traversing Gplot on all different size scales are therefore regions of forbidden energy. One of the most startling properties of Gplot is that when α is rational (say p/q in lowest terms), there are exactly q such bands (though when q is even, two of them "kiss" in the middle). And when α is irrational, the bands shrink to points, of which there are infinitely many, very sparsely distributed in a so-called "Cantor set"—another recursively defined entity which springs up in topology.

You might well wonder whether such an intricate structure would ever show up in an experiment. Frankly, I would be the most surprised person in the world if Gplot came out of any experiment. The physicality of Gplot lies in the fact that it points the way to the proper mathematical treatment of less idealized problems of this sort. In other words, Gplot is purely a contribution to theoretical physics, not a hint to experimentalists as to what to expect to see! An agnostic friend of mine once was so struck by Gplot's infinitely many infinities that he called it "a picture of God", which I don't think is blasphemous at all.

Recursion at the Lowest Level of Matter

We have seen recursion in the grammars of languages, we have seen recursive geometrical trees which grow upwards forever, and we have seen one way in which recursion enters the theory of solid state physics. Now we are going to see yet another way in which the whole world is built out of recursion. This has to do with the structure of elementary particles: electrons, protons, neutrons, and the tiny quanta of electromagnetic radiation called "photons". We are going to see that particles are—in a certain sense which can only be defined rigorously in relativistic quantum mechanics—nested inside each other in a way which can be described recursively, perhaps even by some sort of "grammar".

We begin with the observation that if particles didn't interact with each other, things would be incredibly simple. Physicists would like such a world because then they could calculate the behavior of all particles easily (if physicists in such a world existed, which is a doubtful proposition). Particles without interactions are called *bare particles*, and they are purely hypothetical creations; they don't exist.

Now when you "turn on" the interactions, then particles get tangled up together in the way that functions F and M are tangled together, or married people are tangled together. These real particles are said to be *renormalized*—an ugly but intriguing term. What happens is that no particle can even be defined without referring to all other particles, whose definitions in turn depend on the first particles, etc. Round and round, in a never-ending loop.

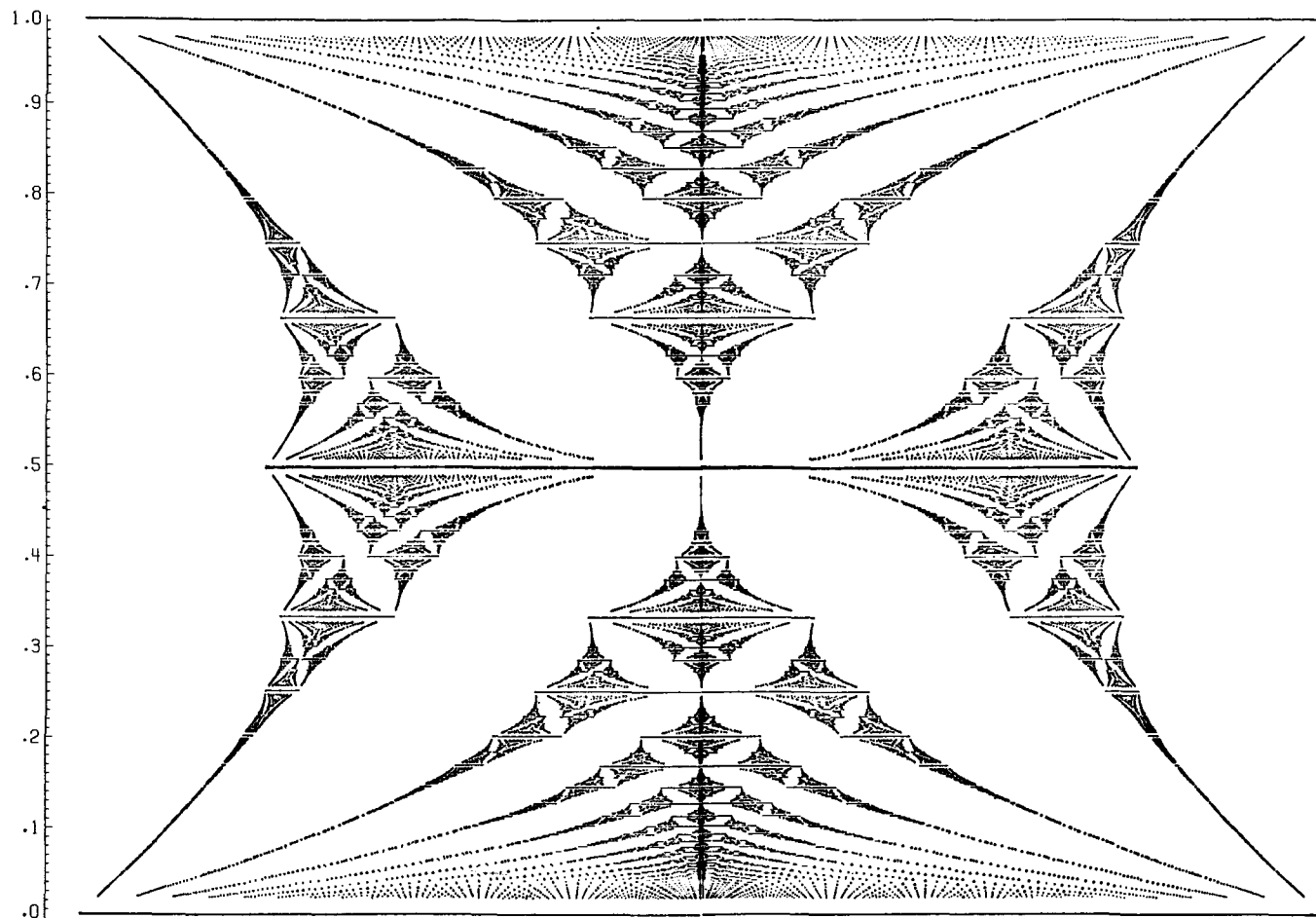


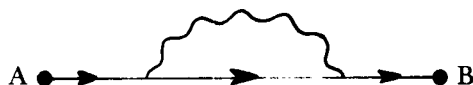
FIGURE 34. Gplot: a recursive graph showing energy bands for electrons in an idealized crystal in a magnetic field. α , representing magnetic field strength, runs vertically from 0 to 1. Energy runs horizontally. The horizontal line segments are bands of allowed electron energies.

Let us be a little more concrete, now. Let's limit ourselves to only two kinds of particles: *electrons* and *photons*. We'll also have to throw in the electron's antiparticle, the *positron*. (Photons are their own antiparticles.) Imagine first a dull world where a bare electron wishes to propagate from point A to point B, as Zeno did in my *Three-Part Invention*. A physicist would draw a picture like this:

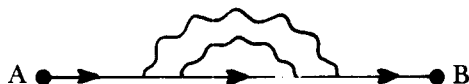


There is a mathematical expression which corresponds to this line and its endpoints, and it is easy to write down. With it, a physicist can understand the behavior of the bare electron in this trajectory.

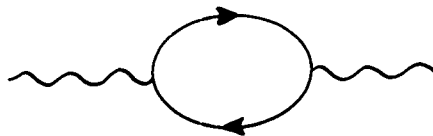
Now let us “turn on” the electromagnetic interaction, whereby electrons and photons interact. Although there are no photons in the scene, there will nevertheless be profound consequences even for this simple trajectory. In particular, our electron now becomes capable of emitting and then reabsorbing *virtual photons*—photons which flicker in and out of existence before they can be seen. Let us show one such process:



Now as our electron propagates, it may emit and reabsorb one photon after another, or it may even nest them, as shown below:



The mathematical expressions corresponding to these diagrams—called “Feynman diagrams”—are easy to write down, but they are harder to calculate than that for the bare electron. But what really complicates matters is that a photon (real or virtual) can decay for a brief moment into an electron-positron pair. Then these two annihilate each other, and, as if by magic, the original photon reappears. This sort of process is shown below:



The electron has a right-pointing arrow, while the positron's arrow points leftwards.

As you might have anticipated, these virtual processes can be nested inside each other to arbitrary depth. This can give rise to some very complicated-looking drawings, such as the one in Figure 35. In that Feynman diagram, a single electron enters on the left at A, does some amazing acrobatics, and then a single electron emerges on the right at B. To an outsider who can't see the inner mess, it looks as if one electron has peacefully sailed from A to B. In the diagram, you can see how electron lines can get arbitrarily embellished, and so can the photon lines. This diagram would be ferociously hard to calculate.

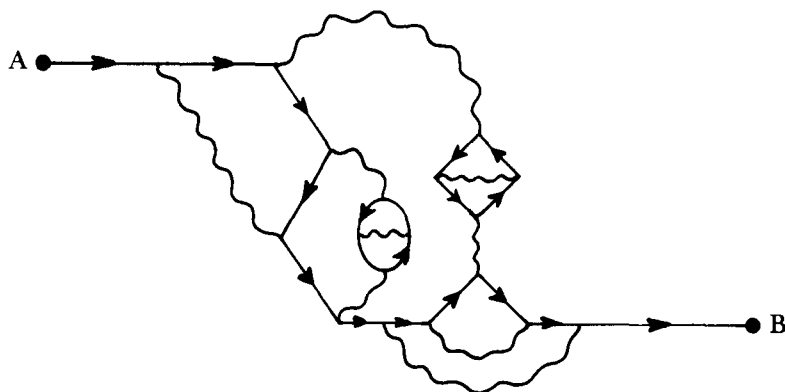


FIGURE 35. A Feynman diagram showing the propagation of a renormalized electron from A to B. In this diagram, time increases to the right. Therefore, in the segments where the electron's arrow points leftwards, it is moving "backwards in time". A more intuitive way to say this is that an antielectron (positron) is moving forwards in time. Photons are their own antiparticles; hence their lines have no need of arrows.

There is a sort of "grammar" to these diagrams, that only allows certain pictures to be realized in nature. For instance, the one below is impossible:



You might say it is not a "well-formed" Feynman diagram. The grammar is a result of basic laws of physics, such as conservation of energy, conservation of electric charge, and so on. And, like the grammars of human languages, this grammar has a recursive structure, in that it allows deep nestings of structures inside each other. It would be possible to draw up a set of recursive transition networks defining the "grammar" of the electromagnetic interaction.

When bare electrons and bare photons are allowed to interact in these arbitrarily tangled ways, the result is *renormalized* electrons and photons. Thus, to understand how a real, physical electron propagates from A to B,

the physicist has to be able to take a sort of average of all the infinitely many different possible drawings which involve virtual particles. This is Zeno with a vengeance!

Thus the point is that a physical particle—a renormalized particle—involves (1) a bare particle and (2) a huge tangle of virtual particles, inextricably wound together in a recursive mess. Every real particle's existence therefore involves the existence of infinitely many other particles, contained in a virtual “cloud” which surrounds it as it propagates. And each of the virtual particles in the cloud, of course, also drags along its own virtual cloud, and so on ad infinitum.

Particle physicists have found that this complexity is too much to handle, and in order to understand the behavior of electrons and photons, they use approximations which neglect all but fairly simple Feynman diagrams. Fortunately, the more complex a diagram, the less important its contribution. There is no known way of summing up all of the infinitely many possible diagrams, to get an expression for the behavior of a fully renormalized, physical electron. But by considering roughly the simplest hundred diagrams for certain processes, physicists have been able to predict one value (the so-called *g*-factor of the muon) to nine decimal places—correctly!

Renormalization takes place not only among electrons and photons. Whenever any types of particle interact together, physicists use the ideas of renormalization to understand the phenomena. Thus protons and neutrons, neutrinos, pi-mesons, quarks—all the beasts in the subnuclear zoo—they all have bare and renormalized versions in physical theories. And from billions of these bubbles within bubbles are all the beasts and baubles of the world composed.

Copies and Sameness

Let us now consider Gplot once again. You will remember that in the Introduction, we spoke of different varieties of canons. Each type of canon exploited some manner of taking an original theme and copying it by an isomorphism, or information-preserving transformation. Sometimes the copies were upside down, sometimes backwards, sometimes shrunk or expanded . . . In Gplot we have all those types of transformation, and more. The mappings between the full Gplot and the “copies” of itself inside itself involve size changes, skewings, reflections, and more. And yet there remains a sort of skeletal identity, which the eye can pick up with a bit of effort, particularly after it has practiced with INT.

Escher took the idea of an object's parts being copies of the object itself and made it into a print: his woodcut *Fishes and Scales* (Fig. 36). Of course these fishes and scales are the same only when seen on a sufficiently abstract plane. Now everyone knows that a fish's scales aren't really small copies of the fish; and a fish's cells aren't small copies of the fish; however, a fish's DNA, sitting inside each and every one of the fish's cells, is a very convo-



FIGURE 36. Fish and Scales, by M. C. Escher (woodcut, 1959).

luted “copy” of the entire fish—and so there is more than a grain of truth to the Escher picture.

What is there that is the “same” about all butterflies? The mapping from one butterfly to another does not map cell onto cell; rather, it maps functional part onto functional part, and this may be partially on a macroscopic scale, partially on a microscopic scale. The exact proportions of parts are not preserved; just the functional relationships between parts. That is the type of isomorphism which links all butterflies in Escher’s wood engraving *Butterflies* (Fig. 37) to each other. The same goes for the more abstract butterflies of Gplot, which are all linked to each other by mathematical mappings that carry functional part onto functional part, but totally ignore exact line proportions, angles, and so on.

Taking this exploration of sameness to a yet higher plane of abstraction, we might well ask, “What is there that is the ‘same’ about all Escher drawings?” It would be quite ludicrous to attempt to map them piece by piece onto each other. The amazing thing is that even a tiny section of an



FIGURE 37. Butterflies, by M. C. Escher (wood-engraving, 1950).

Escher drawing or a Bach piece gives it away. Just as a fish's DNA is contained inside every tiny bit of the fish, so a creator's "signature" is contained inside every tiny section of his creations. We don't know what to call it but "style"—a vague and elusive word.

We keep on running up against "sameness-in-differentness", and the question

When are two things the same?

It will recur over and over again in this book. We shall come at it from all sorts of skew angles, and in the end, we shall see how deeply this simple question is connected with the nature of intelligence.

That this issue arose in the Chapter on recursion is no accident, for recursion is a domain where "sameness-in-differentness" plays a central role. Recursion is based on the "same" thing happening on several differ-

ent levels at once. But the events on different levels *aren't* exactly the same—rather, we find some invariant feature in them, despite many ways in which they differ. For example, in the *Little Harmonic Labyrinth*, all the stories on different levels are quite unrelated—their “sameness” resides in only two facts: (1) they are stories, and (2) they involve the Tortoise and Achilles. Other than that, they are radically different from each other.

Programming and Recursion: Modularity, Loops, Procedures

One of the essential skills in computer programming is to perceive when two processes are the same in this extended sense, for that leads to *modularization*—the breaking-up of a task into natural subtasks. For instance, one might want a sequence of many similar operations to be carried out one after another. Instead of writing them all out, one can write a *loop*, which tells the computer to perform a fixed set of operations and then loop back and perform them again, over and over, until some condition is satisfied. Now the *body* of the loop—the fixed set of instructions to be repeated—need not actually be completely fixed. It may vary in some predictable way.

An example is the most simple-minded test for the primality of a natural number N , in which you begin by trying to divide N by 2, then by 3, 4, 5, etc. until $N - 1$. If N has survived all these tests without being divisible, it's prime. Notice that each step in the loop is similar to, but not the same as, each other step. Notice also that the number of steps varies with N —hence a loop of fixed length could never work as a general test for primality. There are two criteria for “aborting” the loop: (1) if some number divides N exactly, quit with answer “NO”; (2) if $N - 1$ is reached as a test divisor and N survives, quit with answer “YES”.

The general idea of loops, then, is this: perform some series of related steps over and over, and abort the process when specific conditions are met. Now sometimes, the maximum number of steps in a loop will be known in advance; other times, you just begin, and wait until it is aborted. The second type of loop—which I call a *free* loop—is dangerous, because the criterion for abortion may never occur, leaving the computer in a so-called “infinite loop”. This distinction between *bounded loops* and *free loops* is one of the most important concepts in all of computer science, and we shall devote an entire Chapter to it: “BlooP and FlooP and Gloop”.

Now loops may be nested inside each other. For instance, suppose that we wish to test all the numbers between 1 and 5000 for primality. We can write a second loop which uses the above-described test over and over, starting with $N = 1$ and finishing with $N = 5000$. So our program will have a “loop-the-loop” structure. Such program structures are typical—in fact they are deemed to be good programming style. This kind of nested loop also occurs in assembly instructions for commonplace items, and in such activities as knitting or crocheting—in which very small loops are

repeated several times in larger loops, which in turn are carried out repeatedly . . . While the result of a low-level loop might be no more than couple of stitches, the result of a high-level loop might be a substantial portion of a piece of clothing.

In music, too, nested loops often occur—as, for instance, when a scale (a small loop) is played several times in a row, perhaps displaced in pitch each new time. For example, the last movements of both the Prokofiev fifth piano concerto and the Rachmaninoff second symphony contain extended passages in which fast, medium, and slow scale-loops are played simultaneously by different groups of instruments, to great effect. The Prokofiev-scales go up; the Rachmaninoff-scales, down. Take your pick.

A more general notion than loop is that of *subroutine*, or *procedure*, which we have already discussed somewhat. The basic idea here is that a group of operations are lumped together and considered a single unit with a name—such as the procedure ORNATE NOUN. As we saw in RTN's, procedures can call each other by name, and thereby express very concisely sequences of operations which are to be carried out. This is the essence of modularity in programming. Modularity exists, of course, in hi-fi systems, furniture, living cells, human society—wherever there is hierarchical organization.

More often than not, one wants a procedure which will act variably, according to context. Such a procedure can either be given a way of peering out at what is stored in memory and selecting its actions accordingly, or it can be explicitly fed a list of *parameters* which guide its choice of what actions to take. Sometimes both of these methods are used. In RTN-terminology, choosing the sequence of actions to carry out amounts to *choosing which pathway to follow*. An RTN which has been souped up with parameters and conditions that control the choice of pathways inside it is called an *Augmented Transition Network* (ATN). A place where you might prefer ATN's to RTN's is in producing sensible—as distinguished from nonsensical—English sentences out of raw words, according to a grammar represented in a set of ATN's. The parameters and conditions would allow you to insert various semantic constraints, so that random juxtapositions like “a thankless brunch” would be prohibited. More on this in Chapter XVIII, however.

Recursion in Chess Programs

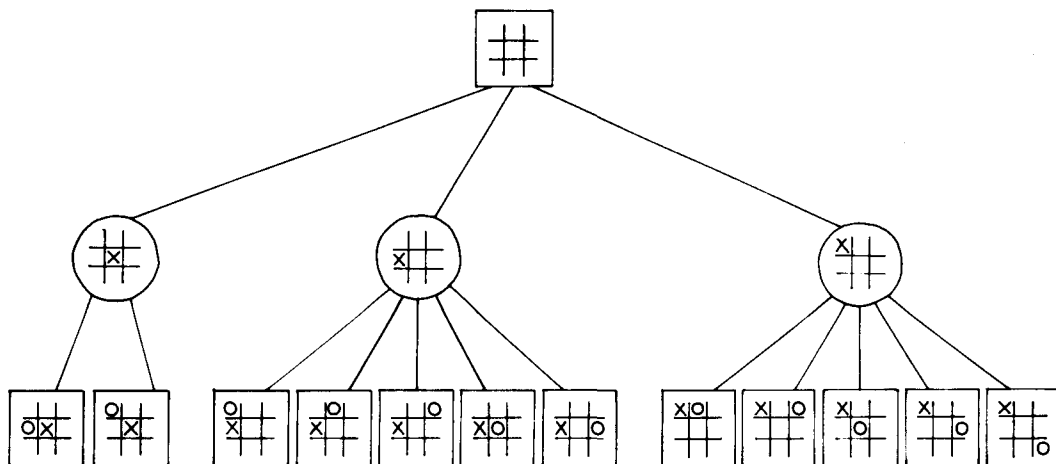
A classic example of a recursive procedure with parameters is one for choosing the “best” move in chess. The best move would seem to be the one which leaves your opponent in the toughest situation. Therefore, a test for goodness of a move is simply this: pretend you've made the move, and now evaluate the board from the point of view of your opponent. But how does your opponent evaluate the position? Well, he looks for *his* best move. That is, he mentally runs through all possible moves and evaluates them from what he thinks is *your* point of view, hoping they will look bad to you. But

notice that we have now defined “best move” recursively, simply using the maxim that what is best for one side is worst for the other. The recursive procedure which looks for the best move operates by trying a move, and then *calling on itself in the role of opponent!* As such, it tries another move, and calls on itself in the role of its opponent’s opponent—that is, itself.

This recursion can go several levels deep—but it’s got to bottom out somewhere! How do you evaluate a board position *without* looking ahead? There are a number of useful criteria for this purpose, such as simply the number of pieces on each side, the number and type of pieces under attack, the control of the center, and so on. By using this kind of evaluation at the bottom, the recursive move-generator can pop back upwards and give an evaluation at the top level of each different move. One of the parameters in the self-calling, then, must tell how many moves to look ahead. The outermost call on the procedure will use some externally set value for this parameter. Thereafter, each time the procedure recursively calls itself, it must decrease this look-ahead parameter by 1. That way, when the parameter reaches zero, the procedure will follow the alternate pathway—the non-recursive evaluation.

In this kind of game-playing program, each move investigated causes the generation of a so-called “look-ahead tree”, with the move itself as trunk, responses as main branches, counter-responses as subsidiary branches, and so on. In Figure 38 I have shown a simple look-ahead tree, depicting the start of a tic-tac-toe game. There is an art to figuring out how to avoid exploring every branch of a look-ahead tree out to its tip. In chess trees, people—not computers—seem to excel at this art; it is known that top-level players look ahead relatively little, compared to most chess programs—yet the people are far better! In the early days of computer chess, people used to estimate that it would be ten years until a computer (or

FIGURE 38. The branching tree of moves and counter moves at the start of a game of tic-tac-toe.



program) was world champion. But after ten years had passed, it seemed that the day a computer would become world champion was still more than ten years away . . . This is just one more piece of evidence for the rather recursive

Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law.

Recursion and Unpredictability

Now what is the connection between the recursive processes of this Chapter, and the recursive sets of the preceding Chapter? The answer involves the notion of a *recursively enumerable set*. For a set to be r.e. means that it can be generated from a set of starting points (axioms), by the repeated application of rules of inference. Thus, the set grows and grows, each new element being compounded somehow out of previous elements, in a sort of “mathematical snowball”. But this is the essence of recursion—something being defined in terms of simpler versions of itself, instead of explicitly. The Fibonacci numbers and the Lucas numbers are perfect examples of r.e. sets—snowballing from two elements by a recursive rule into infinite sets. It is just a matter of convention to call an r.e. set whose complement is also r.e. “recursive”.

Recursive enumeration is a process in which new things emerge from old things by fixed rules. There seem to be many surprises in such processes—for example the unpredictability of the Q-sequence. It might seem that recursively defined sequences of that type possess some sort of inherently increasing complexity of behavior, so that the further out you go, the less predictable they get. This kind of thought carried a little further suggests that suitably complicated recursive systems might be strong enough to break out of any predetermined patterns. And isn't this one of the defining properties of intelligence? Instead of just considering programs composed of procedures which can recursively *call* themselves, why not get really sophisticated, and invent programs which can *modify* themselves—programs which can act on programs, extending them, improving them, generalizing them, fixing them, and so on? This kind of “tangled recursion” probably lies at the heart of intelligence.