Prelude . . .

Achilles and the Tortoise have come to the residence of their friend the Crab, to make the acquaintance of one of his friends, the Anteater. The introductions having been made, the four of them settle down to tea.

Tortoise: We have brought along a little something for you, Mr. Crab.

Crab: That's most kind of you. But you shouldn't have.

Tortoise: Just a token of our esteem. Achilles, would you like to give it to Mr. C?

Achilles: Surely. Best wishes, Mr. Crab. I hope you enjoy it.

(Achilles hands the Crab an elegantly wrapped present, square and very thin. The Crab begins unwrapping it.)

Anteater: I wonder what it could be.

Crab: We'll soon find out. *(Completes the unwrapping, and pulls out the gift.)* Two records! How exciting! But there's no label. Uh-oh—is this another of your "specials", Mr. T?

Tortoise: If you mean a phonograph-breaker, not this time. But it is in fact a custom-recorded item, the only one of its kind in the entire world. In fact, it's never even been heard before—except, of course, when Bach played it.

Crab: When Bach played it? What do you mean, exactly?

Achilles: Oh, you are going to be fabulously excited, Mr. Crab, when Mr. T tells you what these records in fact are.

Tortoise: Oh, you go ahead and tell him, Achilles.

Achilles: May I? Oh, boy! I'd better consult my notes, then. *(Pulls out a small filing card, and clears his voice.)* Ahem. Would you be interested in hearing about the remarkable new result in mathematics, to which your records owe their existence?

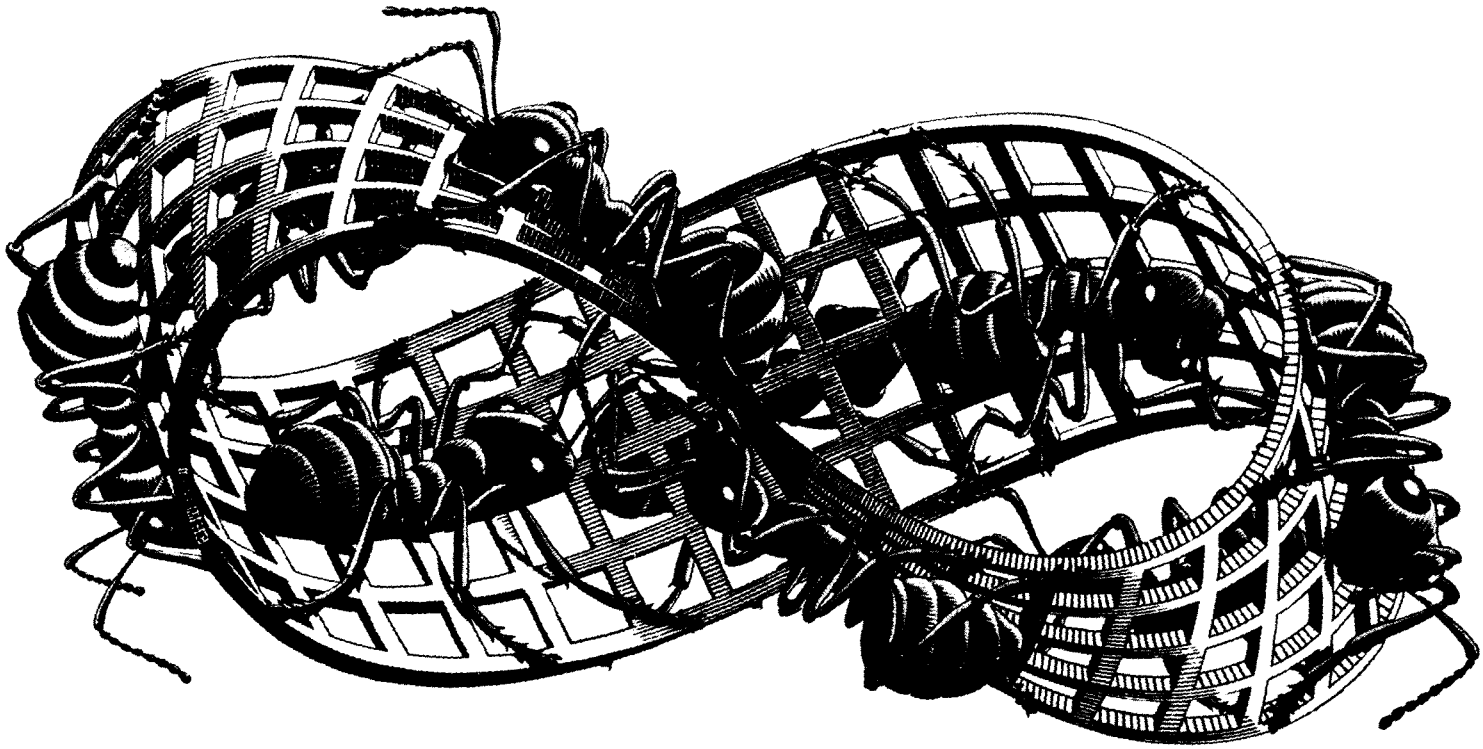
Crab: My records derive from some piece of mathematics? How curious! Well, now that you've provoked my interest, I must hear about it.

Achilles: Very well, then. *(Pauses for a moment to sip his tea, then resumes.)* Have you heard of Fermat's infamous "Last Theorem"?

Anteater: I'm not sure . . . It sounds strangely familiar, and yet I can't quite place it.

Achilles: It's a very simple idea. Pierre de Fermat, a lawyer by vocation but mathematician by avocation, had been reading in his copy of the classic text *Arithmetica* by Diophantus, and came across a page containing the equation

$$a^2 + b^2 = c^2$$



He immediately realized that this equation has infinitely many solutions a, b, c, and then wrote in the margin the following notorious comment:

> The equation
>
> $$a^n + b^n = c^n$$
>
> has solutions in positive integers a, b, c, and n only when n = 2 (and then there are infinitely many triplets a, b, c which satisfy the equation); but there are no solutions for n > 2. I have discovered a truly marvelous proof of this statement, which, unfortunately, this margin is too small to contain.

Ever since that day, some three hundred years ago, mathematicians have been vainly trying to do one of two things: either to prove Fermat's claim, and thereby vindicate Fermat's reputation, which, although very high, has been somewhat tarnished by skeptics who think he never really found the proof he claimed to have found—or else to refute the claim, by finding a counterexample: a set of four integers a, b, c, and n, with n > 2, which satisfy the equation. Until very recently, every attempt in either direction had met with failure. To be sure, the Theorem has been proven for many specific values of n—in particular, all n up to 125,000.

Anteater: Shouldn't it be called a "Conjecture" rather than a "Theorem", if it's never been given a proper proof?

Achilles: Strictly speaking, you're right, but tradition has kept it this way.

Crab: Has someone at last managed to resolve this celebrated question?

Achilles: Indeed! In fact, Mr. Tortoise has done so, and as usual, by a wizardly stroke. He has not only found a PROOF of Fermat's Last Theorem (thus justifying its name as well as vindicating Fermat), but also a COUNTEREXAMPLE, thus showing that the skeptics had good intuition!

Crab: Oh my gracious! That is a revolutionary discovery.

Anteater: But please don't leave us in suspense. What magical integers are they, that satisfy Fermat's equation? I'm especially curious about the value of n.

Achilles: Oh, horrors! I'm most embarrassed! Can you believe this? I left the values at home on a truly colossal piece of paper. Unfortunately it was too huge to bring along. I wish I had them here to show to you. If it's of any help to you, I do remember one thing—the value of n is the only positive integer which does not occur anywhere in the continued fraction for π.

Crab: Oh, what a shame that you don't have them here. But there's no reason to doubt what you have told us.

FIGURE 54. Möbius Strip II, *by M. C. Escher (woodcut, 1963).*

FIGURE 55. Pierre de Fermat.

Anteater: Anyway, who needs to see n written out decimally? Achilles has just told us how to find it. Well, Mr. T, please accept my hearty felicitations, on the occasion of your epoch-making discovery!

Tortoise: Thank you. But what I feel is more important than the result itself is the practical use to which my result immediately led.

Crab: I am dying to hear about it, since I always thought number theory was the Queen of Mathematics—the purest branch of mathematics—the one branch of mathematics which has NO applications!

Tortoise: You're not the only one with that belief, but in fact it is quite impossible to make a blanket statement about when or how some branch—or even some individual Theorem—of pure mathematics will have important repercussions outside of mathematics. It is quite unpredictable—and this case is a perfect example of that phenomenon.

Achilles: Mr. Tortoise's double-barreled result has created a breakthrough in the field of acoustico-retrieval!

Anteater: What is acoustico-retrieval?

Achilles: The name tells it all: it is the retrieval of acoustic information from extremely complex sources. A typical task of acoustico-retrieval is to reconstruct the sound which a rock made on plummeting into a lake from the ripples which spread out over the lake's surface.

Crab: Why, that sounds next to impossible!

Achilles: Not so. It is actually quite similar to what one's brain does, when it reconstructs the sound made in the vocal cords of another person from the vibrations transmitted by the eardrum to the fibers in the cochlea.

Crab: I see. But I still don't see where number theory enters the picture, or what this all has to do with my new records.

Achilles: Well, in the mathematics of acoustico-retrieval, there arise many questions which have to do with the number of solutions of certain Diophantine equations. Now Mr. T has been for years trying to find a way of reconstructing the sounds of Bach playing his harpsichord, which took place over two hundred years ago, from calculations involving the motions of all the molecules in the atmosphere at the present time.

Anteater: Surely that is impossible! They are irretrievably gone, gone forever!

Achilles: Thus think the naïve . . . But Mr. T has devoted many years to this problem, and came to the realization that the whole thing hinged on the number of solutions to the equation

$$a^n + b^n = c^n$$

in positive integers, with $n > 2$.

Tortoise: I could explain, of course, just how this equation arises, but I'm sure it would bore you.

Achilles: It turned out that acoustico-retrieval theory predicts that the Bach sounds can be retrieved from the motion of all the molecules in the atmosphere, provided that EITHER there exists at least one solution to the equation—

Crab: Amazing!

Anteater: Fantastic!

Tortoise: Who would have thought!

Achilles: I was about to say, "provided that there exists EITHER such a solution OR a proof that there are NO solutions!" And therefore, Mr. T, in careful fashion, set about working at both ends of the problem, simultaneously. As it turns out, the discovery of the counterexample was the key ingredient to finding the proof, so the one led directly to the other.

Crab: How could that be?

Tortoise: Well, you see, I had shown that the structural layout of any proof of Fermat's Last Theorem—if one existed—could be described by an elegant formula, which, it so happened, depended on the values of a solution to a certain equation. When I found this second equation, to my surprise it turned out to be the Fermat equation. An amusing accidental relationship between form and content. So when I found the counterexample, all I needed to do was to use those numbers as a blueprint for constructing my proof that there were no solutions to the equation. Remarkably simple, when you think about it. I can't imagine why no one had ever found the result before.

Achilles: As a result of this unanticipatedly rich mathematical success, Mr. T was able to carry out the acoustico-retrieval which he had so long dreamed of. And Mr. Crab's present here represents a palpable realization of all this abstract work.

Crab: Don't tell me it's a recording of Bach playing his own works for harpsichord!

Achilles: I'm sorry, but I have to, for that is indeed just what it is! This is a set of two records of Johann Sebastian Bach playing all of his *Well-Tempered Clavier.* Each record contains one of the two volumes of the *Well-Tempered Clavier;* that is to say, each record contains 24 preludes and fugues—one in each major and minor key.

Crab: Well, we must absolutely put one of these priceless records on, immediately! And how can I ever thank the two of you?

Tortoise: You have already thanked us plentifully, with this delicious tea which you have prepared.

(The Crab slides one of the records out of its jacket, and puts it on. The sound of an incredibly masterful harpsichordist fills the room, in the highest imaginable fidelity. One even hears—or is it one's imagination?—the soft sounds of Bach singing to himself as he plays . . .)

Crab: Would any of you like to follow along in the score? I happen to have a unique edition of the *Well-Tempered Clavier,* specially illuminated by a teacher of mine who happens also to be an unusually fine calligrapher.

Tortoise: I would very much enjoy that.

(The Crab goes to his elegant glass-enclosed wooden bookcase, opens the doors, and draws out two large volumes.)

Crab: Here you are, Mr. Tortoise. I've never really gotten to know all the beautiful illustrations in this edition. Perhaps your gift will provide the needed impetus for me to do so.

Tortoise: I do hope so.

Anteater: Have you ever noticed how in these pieces the prelude always sets the mood perfectly for the following fugue?

Crab: Yes. Although it may be hard to put it into words, there is always some subtle relation between the two. Even if the prelude and fugue do not have a common melodic subject, there is nevertheless always some intangible abstract quality which underlies both of them, binding them together very strongly.

Tortoise: And there is something very dramatic about the few moments of silent suspense hanging between prelude and fugue—that moment where the the theme of the fugue is about to ring out, in single tones, and then to join with itself in ever-increasingly complex levels of weird, exquisite harmony.

Achilles: I know just what you mean. There are so many preludes and fugues which I haven't yet gotten to know, and for me that fleeting interlude of silence is very exciting; it's a time when I try to second-guess old Bach. For example, I always wonder what the fugue's tempo will be: allegro, or adagio? Will it be in 6/8, or 4/4? Will it have three voices, or five—or four? And then, the first voice starts . . . Such an exquisite moment.

Crab: Ah, yes, well do I remember those long-gone days of my youth, the days when I thrilled to each new prelude and fugue, filled with the excitement of their novelty and beauty and the many unexpected surprises which they conceal.

Achilles: And now? Is that thrill all gone?

Crab: It's been supplanted by familiarity, as thrills always will be. But in that familiarity there is also a kind of depth, which has its own compensations. For instance, I find that there are always new surprises which I hadn't noticed before.

Achilles: Occurrences of the theme which you had overlooked?

Crab: Perhaps—especially when it is inverted and hidden among several other voices, or where it seems to come rushing up from the depths, out of nowhere. But there are also amazing modulations which it is marvelous to listen to over and over again, and wonder how old Bach dreamt them up.

Achilles: I am very glad to hear that there is something to look forward to, after I have been through the first flush of infatuation with the *Well-Tempered Clavier*—although it also makes me sad that this stage could not last forever and ever.

Crab: Oh, you needn't fear that your infatuation will totally die. One of the nice things about that sort of youthful thrill is that it can always be resuscitated, just when you thought it was finally dead. It just takes the right kind of triggering from the outside.

Achilles: Oh, really? Such as what?

Crab: Such as hearing it through the ears, so to speak, of someone to whom it is a totally new experience—someone such as you, Achilles. Somehow the excitement transmits itself, and I can feel thrilled again.

Achilles: That is intriguing. The thrill has remained dormant somewhere inside you, but by yourself, you aren't able to fish it up out of your subconscious.

Crab: Exactly. The potential of reliving the thrill is "coded", in some unknown way, in the structure of my brain, but I don't have the power to summon it up at will; I have to wait for chance circumstance to trigger it.

Achilles: I have a question about fugues which I feel a little embarrassed about asking, but as I am just a novice at fugue-listening, I was wondering if perhaps one of you seasoned fugue-listeners might help me in learning . . . ?

Tortoise: I'd certainly like to offer my own meager knowledge, if it might prove of some assistance.

Achilles: Oh, thank you. Let me come at the question from an angle. Are you familiar with the print called *Cube with Magic Ribbons,* by M. C. Escher?

Tortoise: In which there are circular bands having bubble-like distortions which, as soon as you've decided that they are bumps, seem to turn into dents—and vice versa?

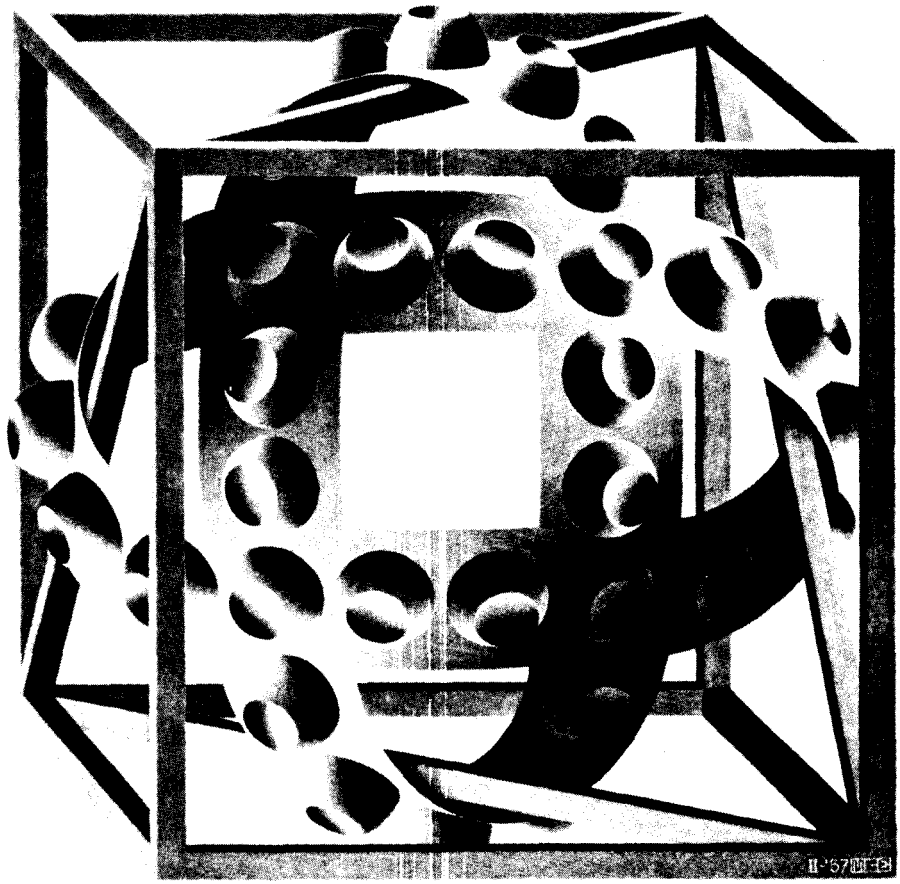


FIGURE 56. Cube with Magic Ribbons, *by M. C. Escher (lithograph, 1957).*

Achilles: Exactly.

Crab: I remember that picture. Those little bubbles always seem to flip back and forth between being concave and convex, depending on the direction that you approach them from. There's no way to see them simultaneously as concave AND convex—somehow one's brain doesn't allow that. There are two mutually exclusive "modes" in which one can perceive the bubbles.

Achilles: Just so. Well, I seem to have discovered two somewhat analogous modes in which I can listen to a fugue. The modes are these: either to follow one individual voice at a time, or to listen to the total effect of all of them together, without trying to disentangle one from another. I have tried out both of these modes, and, much to my frustration, each one of them shuts out the other. It's simply not in my power to follow the paths of individual voices and at the same time to hear the whole effect. I find that I flip back and forth between one mode and the other, more or less spontaneously and involuntarily.

Anteater: Just as when you look at the magic bands, eh?

Achilles: Yes. I was just wondering . . . does my description of these two modes of fugue-listening brand me unmistakably as a naïve, inexperienced listener, who couldn't even begin to grasp the deeper modes of perception which exist beyond his ken?

Tortoise: No, not at all, Achilles. I can only speak for myself, but I too find myself shifting back and forth from one mode to the other without exerting any conscious control over which mode should be dominant. I don't know if our other companions here have also experienced anything similar.

Crab: Most definitely. It's quite a tantalizing phenomenon, since you feel that the essence of the fugue is flitting about you, and you can't quite grasp all of it, because you can't quite make yourself function both ways at once.

Anteater: Fugues have that interesting property, that each of their voices is a piece of music in itself; and thus a fugue might be thought of as a collection of several distinct pieces of music, all based on one single theme, and all played simultaneously. And it is up to the listener (or his subconscious) to decide whether it should be perceived as a unit, or as a collection of independent parts, all of which harmonize.

Achilles: You say that the parts are "independent", yet that can't be literally true. There has to be some coordination between them, otherwise when they were put together one would just have an unsystematic clashing of tones—and that is as far from the truth as could be.

Anteater: A better way to state it might be this: if you listened to each voice on its own, you would find that it seemed to make sense all by itself. It could stand alone, and that is the sense in which I meant that it is independent. But you are quite right in pointing out that each of these individually meaningful lines fuses with the others in a highly nonrandom way, to make a graceful totality. The art of writing a beautiful fugue lies precisely in this ability, to manufacture several different lines, each one of which gives the illusion of having been written for its own beauty, and yet which when taken together form a whole, which does not feel forced in any way. Now, this dichotomy between hearing a fugue as a whole, and hearing its component voices, is a particular example of a very general dichotomy, which applies to many kinds of structures built up from lower levels.

Achilles: Oh, really? You mean that my two "modes" may have some more general type of applicability, in situations other than fugue-listening?

Anteater: Absolutely.

Achilles: I wonder how that could be. I guess it has to do with alternating between perceiving something as a whole, and perceiving it as a collection of parts. But the only place I have ever run into that dichotomy is in listening to fugues.

Tortoise: Oh, my, look at this! I just turned the page while following the music, and came across this magnificent illustration facing the first page of the fugue.

Crab: I have never seen that illustration before. Why don't you pass it 'round?

(The Tortoise passes the book around. Each of the foursome looks at it in a characteristic way—this one from afar, that one from close up, everyone tipping his head this way and that in puzzlement. Finally it has made the rounds, and returns to the Tortoise, who peers at it rather intently.)

Achilles: Well, I guess the prelude is just about over. I wonder if, as I listen to this fugue, I will gain any more insight into the question, "What is the right way to listen to a fugue: as a whole, or as the sum of its parts?"

TTortoise: Listen carefully, and you will!

(The prelude ends. There is a moment of silence; and . . .)

[*ATTACCA*]

CHAPTER X

Levels of Description, and Computer Systems

Levels of Description

GÖDEL'S STRING G, and a Bach fugue: they both have the property that they can be understood on different levels. We are all familiar with this kind of thing; and yet in some cases it confuses us, while in others we handle it without any difficulty at all. For example, we all know that we human beings are composed of an enormous number of cells (around twenty-five trillion), and therefore that everything we do could in principle be described in terms of cells. Or it could even be described on the level of molecules. Most of us accept this in a rather matter-of-fact way; we go to the doctor, who looks at us on lower levels than we think of ourselves. We read about DNA and "genetic engineering" and sip our coffee. We seem to have reconciled these two inconceivably different pictures of ourselves simply by disconnecting them from each other. We have almost no way to relate a microscopic description of ourselves to that which we feel ourselves to be, and hence it is possible to store separate representations of ourselves in quite separate "compartments" of our minds. Seldom do we have to flip back and forth between these two concepts of ourselves, wondering "How can these two totally different things be the same *me*?"

Or take a sequence of images on a television screen which shows Shirley MacLaine laughing. When we watch that sequence, we know that we are actually looking not at a woman, but at sets of flickering dots on a flat surface. We know it, but it is the furthest thing from our mind. We have these two wildly opposing representations of what is on the screen, but that does not confuse us. We can just shut one out, and pay attention to the other—which is what all of us do. Which one is "more real"? It depends on whether you're a human, a dog, a computer, or a television set.

Chunking and Chess Skill

One of the major problems of Artificial Intelligence research is to figure out how to bridge the gap between these two descriptions; how to construct a system which can accept one level of description, and produce the other. One way in which this gap enters Artificial Intelligence is well illustrated by the progress in knowledge about how to program a computer to play good chess. It used to be thought—in the 1950's and on into the 1960's—that the

trick to making a machine play well was to make the machine look further ahead into the branching network of possible sequences of play than any chess master can. However, as this goal gradually became attained, the level of computer chess did not have any sudden spurt, and surpass human experts. In fact, a human expert can quite soundly and confidently trounce the best chess programs of this day.

The reason for this had actually been in print for many years. In the 1940's, the Dutch psychologist Adriaan de Groot made studies of how chess novices and chess masters perceive a chess situation. Put in their starkest terms, his results imply that chess masters perceive the distribution of pieces in *chunks*. There is a higher-level description of the board than the straightforward "white pawn on K5, black rook on Q6" type of description, and the master somehow produces such a mental image of the board. This was proven by the high speed with which a master could reproduce an actual position taken from a game, compared with the novice's plodding reconstruction of the position, after both of them had had five-second glances at the board. Highly revealing was the fact that masters' mistakes involved placing whole *groups* of pieces in the wrong place, which left the game strategically almost the same, but to a novice's eyes, not at all the same. The clincher was to do the same experiment but with pieces randomly assigned to the squares on the board, instead of copied from actual games. The masters were found to be simply no better than the novices in reconstructing such random boards.

The conclusion is that in normal chess play, certain types of situation recur—certain patterns—and it is to those high-level patterns that the master is sensitive. He thinks *on a different level* from the novice; his set of concepts is different. Nearly everyone is surprised to find out that in actual play, a master rarely looks ahead any further than a novice does—and moreover, a master usually examines only a handful of possible moves! The trick is that his mode of perceiving the board is like a filter: he literally *does not see bad moves* when he looks at a chess situation—no more than chess amateurs see *illegal* moves when they look at a chess situation. Anyone who has played even a little chess has organized his perception so that diagonal rook-moves, forward captures by pawns, and so forth, are never brought to mind. Similarly, master-level players have built up higher levels of organization in the way they see the board; consequently, to them, bad moves are as unlikely to come to mind as illegal moves are, to most people. This might be called *implicit pruning* of the giant branching tree of possibilities. By contrast, *explicit pruning* would involve thinking of a move, and after superficial examination, deciding not to pursue examining it any further.

The distinction can apply just as well to other intellectual activities—for instance, doing mathematics. A gifted mathematician doesn't usually think up and try out all sorts of false pathways to the desired theorem, as less gifted people might do; rather, he just "smells" the promising paths, and takes them immediately.

Computer chess programs which rely on looking ahead have not been taught to think on a higher level; the strategy has just been to use brute

force look-ahead, hoping to crush all types of opposition. But it has not worked. Perhaps someday, a look-ahead program with enough brute force will indeed overcome the best human players—but that will be a small intellectual gain, compared to the revelation that intelligence depends *crucially* on the ability to create high-level descriptions of complex arrays, such as chess boards, television screens, printed pages, or paintings.

Similar Levels

Usually, we are not required to hold more than one level of understanding of a situation in our minds at once. Moreover, the different descriptions of a single system are usually so conceptually distant from each other that, as was mentioned earlier, there is no problem in maintaining them both; they are just maintained in separate mental compartments. What *is* confusing, though, is when a single system admits of two or more descriptions on different levels which nevertheless *resemble* each other in some way. Then we find it hard to avoid mixing levels when we think about the system, and can easily get totally lost.

Undoubtedly this happens when we think about our own psychology—for instance, when we try to understand people's motivations for various actions. There are many levels in the human mental structure—certainly it is a system which we do not understand very well yet. But there are hundreds of rival theories which tell why people act the way they do, each theory based on some underlying assumptions about how far down in this set of levels various kinds of psychological "forces" are found. Since at this time we use pretty much the same kind of language for all mental levels, this makes for much level-mixing and most certainly for hundreds of wrong theories. For instance, we talk of "drives"—for sex, for power, for fame, for love, etc., etc.—without knowing where these drives come from in the human mental structure. Without belaboring the point, I simply wish to say that our confusion about who we are is certainly related to the fact that we consist of a large set of levels, and we use overlapping language to describe ourselves on all of those levels.

Computer Systems

There is another place where many levels of description coexist for a single system, and where all the levels are conceptually quite close to one another. I am referring to computer systems. When a computer program is running, it can be viewed on a number of levels. On each level, the description is given in the language of computer science, which makes all the descriptions similar in some ways to each other—yet there are extremely important differences between the views one gets on the different levels. At the lowest level, the description can be so complicated that it is like the dot-description of a television picture. For some purposes, however, this is by far the most important view. At the highest level, the description is greatly *chunked*, and

takes on a completely different feel, despite the fact that many of the same concepts appear on the lowest and highest levels. The chunks on the high-level description are like the chess expert's chunks, and like the chunked description of the image on the screen: they summarize in capsule form a number of things which on lower levels are seen as separate. (See Fig. 57.) Now before things become too abstract, let us pass on to the

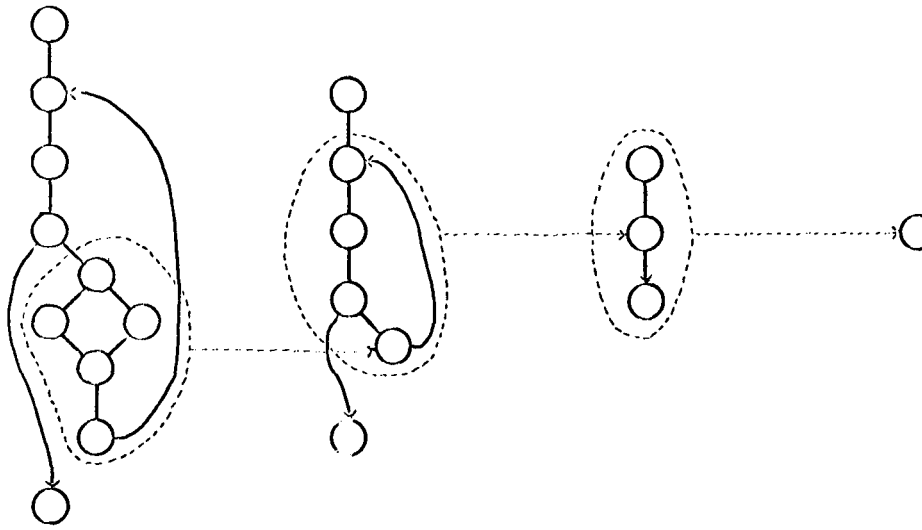
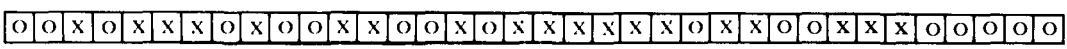


FIGURE 57. The idea of "chunking": a group of items is reperceived as a single "chunk". The chunk's boundary is a little like a cell membrane or a national border: it establishes a separate identity for the cluster within. According to context, one may wish to ignore the chunk's internal structure or to take it into account.

concrete facts about computers, beginning with a very quick skim of what a computer system is like on the lowest level. The lowest level? Well, not really, for I am not going to talk about elementary particles—but it is the lowest level which we wish to think about.

At the conceptual rock-bottom of a computer, we find a *memory*, a *central processing unit* (CPU), and some *input-output* (I/O) *devices*. Let us first describe the memory. It is divided up into distinct physical pieces, called *words*. For the sake of concreteness, let us say there are 65,536 words of memory (a typical number, being 2 to the 16th power). A word is further divided into what we shall consider the atoms of computer science—*bits*. The number of bits in a typical word might be around thirty-six. Physically, a bit is just a magnetic "switch" that can be in either of two positions.



--- a word of 36 bits ---

You could call the two positions "up" and "down", or "x" and "o", or "1" and "0" . . . The third is the usual convention. It is perfectly fine, but it has the possibly misleading effect of making people think that a computer, deep down, is storing *numbers.* This is not true. A set of thirty-six bits does not have to be thought of as a number any more than two bits has to be thought of as the price of an ice cream cone. Just as money can do various things depending on how you use it, so a word in memory can serve many functions. Sometimes, to be sure, those thirty-six bits will indeed represent a number in binary notation. Other times, they may represent thirty-six dots on a television screen. And other times, they may represent a few letters of text. How a word in memory is to be thought of depends entirely on the role that this word plays in the program which uses it. It may, of course, play more than one role—like a note in a canon.

Instructions and Data

There is one interpretation of a word which I haven't yet mentioned, and that is as an *instruction.* The words of memory contain not only data to be acted on, but also the program to act on the data. There exists a limited repertoire of operations which can be carried out by the central processing unit—the CPU—and part of a word, usually its first several bits—is interpretable as the name of the instruction-type which is to be carried out. What do the rest of the bits in a word-interpreted-as-instruction stand for? Most often, they tell which other words in memory are to be acted upon. In other words, the remaining bits constitute a *pointer* to some other word (or words) in memory. Every word in memory has a distinct location, like a house on a street; and its location is called its *address.* Memory may have one "street", or many "streets"—they are called "pages". So a given word is addressed by its page number (if memory is paged) together with its position within the page. Hence the "pointer" part of an instruction is the numerical address of some word(s) in memory. There are no restrictions on the pointer, so an instruction may even "point" at itself, so that when it is executed, it causes a change in itself to be made.

How does the computer know what instruction to execute at any given time? This is kept track of in the CPU. The CPU has a special pointer which points at (i.e., stores the address of) the next word which is to be interpreted as an instruction. The CPU fetches that word from memory, and copies it electronically into a special word belonging to the CPU itself. (Words in the CPU are usually not called "words", but rather, *registers.*) Then the CPU executes that instruction. Now the instruction may call for any of a large number of types of operations to be carried out. Typical ones include:

ADD the word pointed to in the instruction, to a register.
(In this case, the word pointed to is obviously interpreted as a number.)

PRINT the word pointed to in the instruction, as letters.
(In this case, the word is obviously interpreted *not* as a number, but as a string of letters.)

JUMP to the word pointed to in the instruction.
(In this case, the CPU is being told to interpret that particular word as its next instruction.)

Unless the instruction explicitly dictates otherwise, the CPU will pick up the very next word and interpret it as an instruction. In other words, the CPU assumes that it should move down the "street" sequentially, like a mailman, interpreting word after word as an instruction. But this sequential order can be broken by such instructions as the **JUMP** instruction, and others.

Machine Language *vs.* Assembly language

This is a very brief sketch of *machine language.* In this language, the types of operations which exist constitute a finite repertoire which cannot be extended. Thus all programs, no matter how large and complex, must be made out of compounds of these types of instructions. Looking at a program written in machine language is vaguely comparable to looking at a DNA molecule atom by atom. If you glance back to Fig. 41, showing the nucleotide sequence of a DNA molecule—and then if you consider that each nucleotide contains two dozen atoms or so—and if you imagine trying to write the DNA, atom by atom, for a small virus (not to mention a human being!)—then you will get a feeling for what it is like to write a complex program in machine language, and what it is like to try to grasp what is going on in a program if you have access only to its machine language description.

It must be mentioned, however, that computer programming was originally done on an even lower level, if possible, than that of machine language—namely, connecting wires to each other, so that the proper operations were "hard-wired" in. This is so amazingly primitive by modern standards that it is painful even to imagine. Yet undoubtedly the people who first did it experienced as much exhilaration as the pioneers of modern computers ever do . . .

We now wish to move to a higher level of the hierarchy of levels of description of programs. This is the *assembly language* level. There is not a gigantic spread between assembly language and machine language; indeed, the step is rather gentle. In essence, there is a one-to-one correspondence between assembly language instructions and machine language instructions. The idea of assembly language is to "chunk" the individual machine language instructions, so that instead of writing the sequence of bits "010111000" when you want an instruction which adds one number to another, you simply write **ADD**, and then instead of giving the address in binary representation, you can refer to the word in memory by a *name.*

Therefore, a program in assembly language is very much like a machine language program made legible to humans. You might compare the machine language version of a program to a TNT-derivation done in the obscure Gödel-numbered notation, and the assembly language version to the isomorphic TNT-derivation, done in the original TNT-notation, which is much easier to understand. Or, going back to the DNA image, we can liken the difference between machine language and assembly language to the difference between painfully specifying each nucleotide, atom by atom, and specifying a nucleotide by simply giving its *name* (i.e., 'A', 'G', 'C', or 'T'). There is a tremendous saving of labor in this very simple "chunking" operation, although conceptually not much has been changed.

Programs That Translate Programs

Perhaps the central point about assembly language is not its differences from machine language, which are not that enormous, but just the key idea that programs could be written on a different level *at all!* Just think about it: the hardware is built to "understand" machine language programs—sequences of bits—but not letters and decimal numbers. What happens when hardware is fed a program in assembly language? It is as if you tried to get a cell to accept a piece of paper with the nucleotide sequence written out in letters of the alphabet, instead of in chemicals. What can a cell do with a piece of paper? What can a computer do with an assembly language program?

And here is the vital point: someone can write, in machine language, a *translation program.* This program, called an *assembler,* accepts mnemonic instruction names, decimal numbers, and other convenient abbreviations which a programmer can remember easily, and carries out the conversion into the monotonous but critical bit-sequences. After the assembly language program has been *assembled* (i.e., translated), it is *run*—or rather, its machine language equivalent is run. But this is a matter of terminology. Which level program is running? You can never go wrong if you say that the machine language program is running, for hardware is always involved when any program runs—but it is also quite reasonable to think of the running program in terms of assembly language. For instance, you might very well say, "Right now, the CPU is executing a JUMP instruction", instead of saying, "Right now, the CPU is executing a '11101000' instruction". A pianist who plays the notes G-E-B E-G-B is also playing an arpeggio in the chord of E minor. There is no reason to be reluctant about describing things from a higher-level point of view. So one can think of the assembly language program running concurrently with the machine language program. We have two modes of describing what the CPU is doing.

The next level of the hierarchy carries much further the extremely powerful idea of using the computer itself to translate programs from a high level into lower levels. After people had programmed in assembly language for a number of years, in the early 1950's, they realized that there were a number of characteristic structures which kept reappearing in program after program. There seemed to be, just as in chess, certain fundamental patterns which cropped up naturally when human beings tried to formulate *algorithms*—exact descriptions of processes they wanted carried out. In other words, algorithms seemed to have certain higher-level components, in terms of which they could be much more easily and esthetically specified than in the very restricted machine language, or assembly language. Typically, a high-level algorithm component consists not of one or two machine language instructions, but of a whole collection of them, not necesssarily all contiguous in memory. Such a component could be represented in a higher-level language by a single item—a chunk.

Aside from standard chunks—the newly discovered components out of which all algorithms can be built—people realized that almost all programs contain even larger chunks—superchunks, so to speak. These superchunks differ from program to program, depending on the kinds of high-level tasks the program is supposed to carry out. We discussed superchunks in Chapter V, calling them by their usual names: "subroutines" and "procedures". It was clear that a most powerful addition to any programming language would be the ability to *define* new higher-level entities in terms of previously known ones, and then to *call* them by name. This would build the chunking operation right into the language. Instead of there being a determinate repertoire of instructions out of which all programs had to be explicitly assembled, the programmer could construct his own modules, each with its own name, each usable anywhere inside the program, just as if it had been a built-in feature of the language. Of course, there is no getting away from the fact that down below, on a machine language level, everything would still be composed of the same old machine language instructions, but that would not be explicitly visible to the high-level programmer; it would be implicit.

The new languages based on these ideas were called *compiler languages*. One of the earliest and most elegant was called "Algol", for "Algorithmic Language". Unlike the case with assembly language, there is no straightforward one-to-one correspondence between statements in Algol and machine language instructions. To be sure, there is still a type of mapping from Algol into machine language, but it is far more "scrambled" than that between assembly language and machine language. Roughly speaking, an Algol program is to its machine language translation as a word problem in an elementary algebra text is to the equation it translates into. (Actually, getting from a word problem to an equation is far more complex, but it gives some inkling of the types of "unscrambling" that have to be carried out in translating from a high-level language to a lower-level lan-

guage.) In the mid-1950's, successful programs called *compilers* were written whose function was to carry out the translation from compiler languages to machine language.

Also, *interpreters* were invented. Like compilers, interpreters translate from high-level languages into machine language, but instead of translating all the statements first and then executing the machine code, they read one line and execute it immediately. This has the advantage that a user need not have written a complete program to use an interpreter. He may invent his program line by line, and test it out as he goes along. Thus, an interpreter is to a compiler as a simultaneous interpreter is to a translator of a written speech. One of the most important and fascinating of all computer languages is LISP (standing for "List Processing"), which was invented by John McCarthy around the time Algol was invented. Subsequently, LISP has enjoyed great popularity with workers in Artificial Intelligence.

There is one interesting difference between the way interpreters work and compilers work. A compiler takes input (a finished Algol program, for instance) and produces output (a long sequence of machine language instructions). At this point, the compiler has done its duty. The output is then given to the computer to run. By contrast, the interpreter is constantly running while the programmer types in one LISP statement after another, and each one gets executed then and there. But this doesn't mean that each statement gets first translated, then executed, for then an interpreter would be nothing but a line-by-line compiler. Instead, in an interpreter, the operations of reading a new line, "understanding" it, and executing it are intertwined: they occur simultaneously.

Here is the idea, expanded a little more. Each time a new line of LISP is typed in, the interpreter tries to process it. This means that the interpreter jolts into action, and certain (machine language) instructions inside it get executed. Precisely *which* ones get executed depends on the LISP statement itself, of course. There are many **JUMP** instructions inside the interpreter, so that the new line of LISP may cause control to move around in a complex way—forwards, backwards, then forwards again, etc. Thus, each LISP statement gets converted into a "pathway" inside the interpreter, and the act of following that pathway achieves the desired effect.

Sometimes it is helpful to think of the LISP statements as mere pieces of data which are fed sequentially to a constantly running machine language program (the LISP interpreter). When you think of things this way, you get a different image of the relation between a program written in a higher-level language and the machine which is executing it.

Bootstrapping

Of course a compiler, being itself a program, has to be written in some language. The first compilers were written in assembly language, rather than machine language, thus taking full advantage of the already ac-

complished first step up from machine language. A summary of these rather tricky concepts is presented in Figure 58.

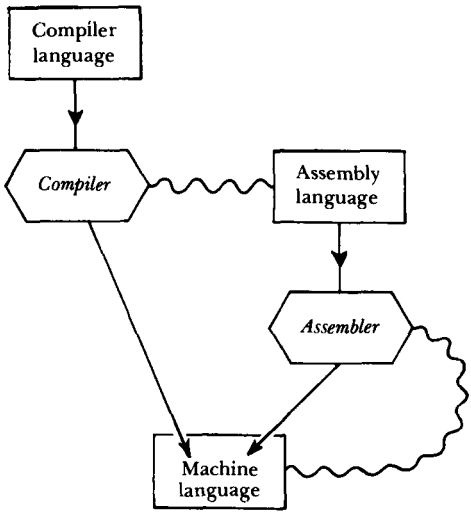


FIGURE 58. Assemblers and compilers are both translators into machine language. This is indicated by the solid lines. Moreover, since they are themselves programs, *they are originally written in a language also. The wavy lines indicate that a compiler can be written in assembly language, and an assembler in machine language.*

Now as sophistication increased, people realized that a partially written compiler could be used to compile extensions of itself. In other words, once a certain minimal core of a compiler had been written, then that minimal compiler could translate bigger compilers into machine language—which in turn could translate yet bigger compilers, until the final, full-blown compiler had been compiled. This process is affectionately known as "bootstrapping"—for obvious reasons (at least if your native language is English it is obvious). It is not so different from the attainment by a child of a critical level of fluency in his native language, from which point on his vocabulary and fluency can grow by leaps and bounds, since he can *use* language to *acquire* new language.

Levels on Which to Describe Running Programs

Compiler languages typically do not reflect the structure of the machines which will run programs written in them. This is one of their chief advantages over the highly specialized assembly and machine languages. Of course, when a compiler language program is translated into machine language, the resulting program is machine-dependent. Therefore one can describe a program which is being executed in a machine-independent way or a machine-dependent way. It is like referring to a paragraph in a book by its subject matter (publisher-independent), or its page number and position on the page (publisher-dependent).

As long as a program is running correctly, it hardly matters how you describe it or think of its functioning. It is when something goes wrong that

it is important to be able to think on different levels. If, for instance, the machine is instructed to divide by zero at some stage, it will come to a halt and let the user know of this problem, by telling where in the program the questionable event occurred. However, the specification is often given on a lower level than that in which the programmer wrote the program. Here are three parallel descriptions of a program grinding to a halt:

Machine Language Level:
"Execution of the program stopped in location 1110010101110111"

Assembly Language Level:
"Execution of the program stopped when the DIV (divide) instruction was hit"

Compiler Language Level:
"Execution of the program stopped during evaluation of the algebraic expression '(A + B)/Z' "

One of the greatest problems for systems programmers (the people who write compilers, interpreters, assemblers, and other programs to be used by many people) is to figure out how to write error-detecting routines in such a way that the messages which they feed to the user whose program has a "bug" provide high-level, rather than low-level, descriptions of the problem. It is an interesting reversal that when something goes wrong in a genetic "program" (e.g., a mutation), the "bug" is manifest only to people on a *high* level—namely on the phenotype level, not the genotype level. Actually, modern biology uses mutations as one of its principal windows onto genetic processes, because of their multilevel traceability.

Microprogramming and Operating Systems

In modern computer systems, there are several other levels of the hierarchy. For instance, some systems—often the so-called "microcomputers"—come with machine language instructions which are even more rudimentary than the instruction to add a number in memory to a number in a register. It is up to the user to decide what kinds of ordinary machine-level instructions he would like to be able to program in; he "microprograms" these instructions in terms of the "micro-instructions" which are available. Then the "higher-level machine language" instructions which he has designed may be burned into the circuitry and become hard-wired, although they need not be. Thus microprogramming allows the user to step a little below the conventional machine language level. One of the consequences is that a computer of one manufacturer can be hard-wired (via microprogramming) so as to have the same machine language instruction set as a computer of the same, or even another, manufacturer. The microprogrammed computer is said to be "emulating" the other computer.

Then there is the level of the *operating system*, which fits between the

machine language program and whatever higher level the user is programming in. The operating system is itself a program which has the functions of shielding the bare machine from access by users (thus protecting the system), and also of insulating the programmer from the many extremely intricate and messy problems of reading the program, calling a translator, running the translated program, directing the output to the proper channels at the proper time, and passing control to the next user. If there are several users "talking" to the same CPU at once, then the operating system is the program that shifts attention from one to the other in some orderly fashion. The complexities of operating systems are formidable indeed, and I shall only hint at them by the following analogy.

Consider the first telephone system. Alexander Graham Bell could phone his assistant in the next room: electronic transmission of a voice! Now that is like a bare computer minus operating system: electronic computation! Consider now a modern telephone system. You have a choice of other telephones to connect to. Not only that, but many different calls can be handled simultaneously. You can add a prefix and dial into different areas. You can call direct, through the operator, collect, by credit card, person-to-person, on a conference call. You can have a call rerouted or traced. You can get a busy signal. You can get a siren-like signal that says that the number you dialed isn't "well-formed", or that you have taken too in long in dialing. You can install a local switchboard so that a group of phones are all locally connected—etc., etc. The list is amazing, when you think of how much flexibility there is, particularly in comparison to the erstwhile miracle of a "bare" telephone. Now sophisticated operating systems carry out similar traffic-handling and level-switching operations with respect to users and their programs. It is virtually certain that there are somewhat parallel things which take place in the brain: handling of many stimuli at the same time; decisions of what should have priority over what and for how long; instantaneous "interrupts" caused by emergencies or other unexpected occurrences; and so on.

Cushioning the User and Protecting the System

The many levels in a complex computer system have the combined effect of "cushioning" the user, preventing him from having to think about the many lower-level goings-on which are most likely totally irrelevant to him anyway. A passenger in an airplane does not usually want to be aware of the levels of fuel in the tanks, or the wind speeds, or how many chicken dinners are to be served, or the status of the rest of the air traffic around the destination—this is all left to employees on different levels of the airlines hierarchy, and the passenger simply gets from one place to another. Here again, it is when something goes *wrong*—such as his baggage not arriving—that the passenger is made aware of the confusing system of levels underneath him.

Are Computers Super-Flexible or Super-Rigid?

One of the major goals of the drive to higher levels has always been to make as natural as possible the task of communicating to the computer what you want it to do. Certainly, the high-level constructs in compiler languages are closer to the concepts which humans naturally think in, than are lower-level constructs such as those in machine language. But in this drive towards ease of communication, one aspect of "naturalness" has been quite neglected. That is the fact that interhuman communication is far less rigidly constrained than human-machine communication. For instance, we often produce meaningless sentence fragments as we search for the best way to express something, we cough in the middle of sentences, we interrupt each other, we use ambiguous descriptions and "improper" syntax, we coin phrases and distort meanings—but our message still gets through, mostly. With programming languages, it has generally been the rule that there is a very strict syntax which has to be obeyed one hundred per cent of the time; there are no ambiguous words or constructions. Interestingly, the printed equivalent of coughing (i.e., a nonessential or irrelevant comment) is allowed, but only provided it is signaled in advance by a key word (e.g., **COMMENT**), and then terminated by another key word (e.g., a semicolon). This small gesture towards flexibility has its own little pitfall, ironically: if a semicolon (or whatever key word is used for terminating a comment) is used inside a comment, the translating program will interpret that semicolon as signaling the end of the comment, and havoc will ensue.

If a procedure named **INSIGHT** has been defined and then called seventeen times in the program, and the eighteenth time it is misspelled as **INSIHGT**, woe to the programmer. The compiler will balk and print a rigidly unsympathetic error message, saying that it has never heard of **INSIHGT**. Often, when such an error is detected by a compiler, the compiler tries to continue, but because of its lack of insihgt, it has not understood what the programmer meant. In fact, it may very well suppose that something entirely different was meant, and proceed under that erroneous assumption. Then a long series of error messages will pepper the rest of the program, because the compiler—not the programmer—got confused. Imagine the chaos that would result if a simultaneous English-Russian interpreter, upon hearing one phrase of French in the English, began trying to interpret all the remaining English as French. Compilers often get lost in such pathetic ways. *C'est la vie.*

Perhaps this sounds condemnatory of computers, but it is not meant to be. In some sense, things had to be that way. When you stop to think what most people use computers for, you realize that it is to carry out very definite and precise tasks, which are too complex for people to do. If the computer is to be reliable, then it is necessary that it should understand, without the slightest chance of ambiguity, what it is supposed to do. It is also necessary that it should do neither more nor less than it is explicitly instructed to do. If there is, in the cushion underneath the programmer, a program whose purpose is to "guess" what the programmer wants or

means, then it is quite conceivable that the programmer could try to communicate his task and be totally misunderstood. So it is important that the high-level program, while comfortable for the human, still should be unambiguous and precise.

Second-Guessing the Programmer

Now it is possible to devise a programming language—and a program which translates it into the lower levels—which allows some sorts of imprecision. One way of putting it would be to say that a translator for such a programming language tries to make sense of things which are done "outside of the rules of the language". But if a language allows certain "transgressions", then transgressions of that type are no longer true transgressions, because they have been included inside the rules! If a programmer is aware that he may make certain types of misspelling, then he may use this feature of the language deliberately, knowing that he is actually operating within the rigid rules of the language, despite appearances. In other words, if the user is aware of all the flexibilities programmed into the translator for his convenience, then he knows the bounds which he cannot overstep, and therefore, to him, the translator still appears rigid and inflexible, although it may allow him much more freedom than early versions of the language, which did not incorporate "automatic compensation for human error".

With "rubbery" languages of that type, there would seem to be two alternatives: (1) the user is aware of the built-in flexibilities of the language and its translator; (2) the user is unaware of them. In the first case, the language is still usable for communicating programs precisely, because the programmer can predict how the computer will interpret the programs he writes in the language. In the second case, the "cushion" has hidden features which may do things that are unpredictable (from the vantage point of a user who doesn't know the inner workings of the translator). This may result in gross misinterpretations of programs, so such a language is unsuitable for purposes where computers are used mainly for their speed and reliability.

Now there is actually a third alternative: (3) the user is aware of the built-in flexibilities of the language and its translator, but there are so many of them and they interact with each other in such a complex way that he cannot tell how his programs will be interpreted. This may well apply to the person who wrote the translating program; he certainly knows its insides as well as anyone could—but he still may not be able to anticipate how it will react to a given type of unusual construction.

One of the major areas of research in Artificial Intelligence today is called *automatic programming*, which is concerned with the development of yet higher-level languages—languages whose translators are sophisticated, in that they can do at least some of the following impressive things: generalize from examples, correct some misprints or grammatical errors,

try to make sense of ambiguous descriptions, try to second-guess the user by having a primitive user model, ask questions when things are unclear, use English itself, etc. The hope is that one can walk the tightrope between reliability and flexibility.

AI Advances Are Language Advances

It is striking how tight the connection is between progress in computer science (particularly Artificial Intelligence) and the development of new languages. A clear trend has emerged in the last decade: the trend to consolidate new types of discoveries in new languages. One key for the understanding and creation of intelligence lies in the constant development and refinement of the languages in terms of which processes for symbol manipulation are describable. Today, there are probably three or four dozen experimental languages which have been developed exclusively for Artificial Intelligence research. It is important to realize that any program which can be written in one of these languages is in principle programmable in lower-level languages, but it would require a supreme effort for a human; and the resulting program would be so long that it would exceed the grasp of humans. It is not that each higher level extends the potential of the computer; the full potential of the computer already exists in its machine language instruction set. It is that the new concepts in a high-level language suggest directions and perspectives by their very nature.

The "space" of all possible programs is so huge that no one can have a sense of what is possible. Each higher-level language is naturally suited for exploring certain regions of "program space"; thus the programmer, by using that language, is channeled into those areas of program space. He is not *forced* by the language into writing programs of any particular type, but the language makes it *easy* for him to do certain kinds of things. Proximity to a concept, and a gentle shove, are often all that is needed for a major discovery—and that is the reason for the drive towards languages of ever higher levels.

Programming in different languages is like composing pieces in different keys, particularly if you work at the keyboard. If you have learned or written pieces in many keys, each key will have its own special emotional aura. Also, certain kinds of figurations "lie in the hand" in one key but are awkward in another. So you are channeled by your choice of key. In some ways, even enharmonic keys, such as C-sharp and D-flat, are quite distinct in feeling. This shows how a notational system can play a significant role in shaping the final product.

A "stratified" picture of AI is shown in Figure 59, with machine components such as transistors on the bottom, and "intelligent programs" on the top. The picture is taken from the book *Artificial Intelligence* by Patrick Henry Winston, and it represents a vision of AI shared by nearly all AI workers. Although I agree with the idea that AI must be stratified in some such way, I do not think that, with so few layers, intelligent programs

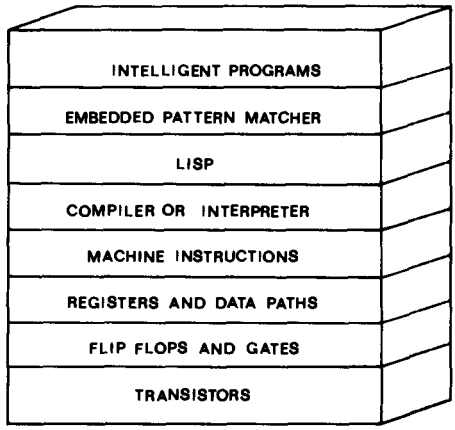


FIGURE 59. To create intelligent programs, one needs to build up a series of levels of hardware and software, so that one is spared the agony of seeing everything only on the lowest level. Descriptions of a single process on different levels will sound very different from each other, only the top one being sufficiently chunked that it is comprehensible to us. [*Adapted from P. H. Winston,* Artificial Intelligence *(Reading, Mass.: Addison-Wesley, 1977).*]

can be reached. Between the machine language level and the level where true intelligence will be reached, I am convinced there will lie perhaps another dozen (or even several dozen!) layers, each new layer building on and extending the flexibilities of the layer below. What they will be like we can hardly dream of now . . .

The Paranoid and the Operating System

The similarity of all levels in a computer system can lead to some strange level-mixing experiences. I once watched a couple of friends—both computer novices—playing with the program "PARRY" on a terminal. PARRY is a rather infamous program which simulates a paranoid in an extremely rudimentary way, by spitting out canned phrases in English chosen from a wide repertoire; its plausibility is due to its ability to tell which of its stock phrases might sound reasonable in response to English sentences typed to it by a human.

At one point, the response time got very sluggish—PARRY was taking very long to reply—and I explained to my friends that this was probably because of the heavy load on the time-sharing system. I told them they could find out how many users were logged on, by typing a special "control" character which would go directly to the operating system, and would be unseen by PARRY. One of my friends pushed the control character. In a flash, some internal data about the operating system's status overwrote some of PARRY's words on the screen. PARRY knew nothing of this: it is a program with "knowledge" only of horse racing and bookies—not operating systems and terminals and special control characters. But to my friends, both PARRY and the operating system were just "the computer"—a mysterious, remote, amorphous entity that responded to them when they typed. And so it made perfect sense when one of them blithely typed, in English, "Why are you overtyping what's on the screen?" The idea that PARRY could know nothing about the operating system it was running

under was not clear to my friends. The idea that "you" know all about "yourself" is so familiar from interaction with people that it was natural to extend it to the computer—after all, it was intelligent enough that it could "talk" to them in English! Their question was not unlike asking a person, "Why are you making so few red blood cells today?" People do not know about that level—the "operating system level"—of their bodies.

The main cause of this level-confusion was that communication with all levels of the computer system was taking place on a single screen, on a single terminal. Although my friends' naïveté might seem rather extreme, even experienced computer people often make similar errors when several levels of a complex system are all present at once on the same screen. They forget "who" they are talking to, and type something which makes no sense at that level, although it would have made perfect sense on another level. It might seem desirable, therefore, to have the system itself sort out the levels—to interpret commands according to what "makes sense". Unfortunately, such interpretation would require the system to have a lot of common sense, as well as perfect knowledge of the programmer's overall intent—both of which would require more artificial intelligence than exists at the present time.

The Border between Software and Hardware

One can also be confused by the flexibility of some levels and the rigidity of others. For instance, on some computers there are marvelous text-editing systems which allow pieces of text to be "poured" from one format into another, practically as liquids can be poured from one vessel into another. A thin page can turn into a wide page, or vice versa. With such power, you might expect that it would be equally trivial to change from one font to another—say from roman to *italics*. Yet there may be only a single font available on the screen, so that such changes are impossible. Or it may be feasible on the screen but not printable by the printer—or the other way around. After dealing with computers for a long time, one gets spoiled, and thinks that everything should be programmable: no printer should be so rigid as to have only one character set, or even a finite repertoire of them—typefaces should be user-specifiable! But once that degree of flexibility has been attained, then one may be annoyed that the printer cannot print in different colors of ink, or that it cannot accept paper of all shapes and sizes, or that it does not fix itself when it breaks . . .

The trouble is that somewhere, all this flexibility has to "bottom out", to use the phrase from Chapter V. There must be a hardware level which underlies it all, and which is inflexible. It may lie deeply hidden, and there may be so much flexibility on levels above it that few users feel the hardware limitations—but it is inevitably there.

What is this proverbial distinction between *software* and *hardware*? It is the distinction between programs and machines—between long complicated sequences of instructions, and the physical machines which carry

them out. I like to think of software as "anything which you could send over the telephone lines", and hardware as "anything else". A piano is hardware, but printed music is software. A telephone set is hardware, but a telephone number is software. The distinction is a useful one, but not always so clear-cut.

We humans also have "software" and "hardware" aspects, and the difference is second nature to us. We are used to the rigidity of our physiology: the fact that we cannot, at will, cure ourselves of diseases, or grow hair of any color—to mention just a couple of simple examples. We can, however, "reprogram" our minds so that we operate in new conceptual frameworks. The amazing flexibility of our minds seems nearly irreconcilable with the notion that our brains must be made out of fixed-rule hardware, which cannot be reprogrammed. We cannot make our neurons fire faster or slower, we cannot rewire our brains, we cannot redesign the interior of a neuron, we cannot make *any* choices about the hardware—and yet, we can control how we think.

But there are clearly aspects of thought which are beyond our control. We cannot make ourselves smarter by an act of will; we cannot learn a new language as fast as we want; we cannot make ourselves think faster than we do; we cannot make ourselves think about several things at once; and so on. This is a kind of primordial self-knowledge which is so obvious that it is hard to see it at all; it is like being conscious that the air is there. We never really bother to think about what might cause these "defects" of our minds: namely, the organization of our brains. To suggest ways of reconciling the software of mind with the hardware of brain is a main goal of this book.

Intermediate Levels and the Weather

We have seen that in computer systems, there are a number of rather sharply defined strata, in terms of any one of which the operation of a running program can be described. Thus there is not merely a single low level and a single high level—there are all degrees of lowness and highness. Is the existence of intermediate levels a general feature of systems which have low and high levels? Consider, for example, the system whose "hardware" is the earth's atmosphere (not very hard, but no matter), and whose "software" is the weather. Keeping track of the motions of all of the molecules simultaneously would be a very low-level way of "understanding" the weather, rather like looking at a huge, complicated program on the machine language level. Obviously it is way beyond human comprehension. But we still have our own peculiarly human ways of looking at, and describing, weather phenomena. Our chunked view of the weather is based on very high-level phenomena, such as: rain, fog, snow, hurricanes, cold fronts, seasons, pressures, trade winds, the jet stream, cumulo-nimbus clouds, thunderstorms, inversion layers, and so on. All of these phenomena involve astronomical numbers of molecules, somehow behaving in concert so that large-scale trends emerge. This is a little like looking at the weather in a compiler language.

Is there something analogous to looking at the weather in an intermediate-level language, such as assembly language? For instance, are there very small local "mini-storms", something like the small whirlwinds which one occasionally sees, whipping up some dust in a swirling column a few feet wide, at most? Is a local gust of wind an intermediate-level chunk which plays a role in creating higher-level weather phenomena? Or is there just no practical way of combining knowledge of such kinds of phenomena to create a more comprehensive explanation of the weather?

Two other questions come to my mind. The first is: "Could it be that the weather phenomena which we perceive on our scale—a tornado, a drought—are just intermediate-level phenomena: parts of vaster, slower phenomena?" If so, then true high-level weather phenomena would be global, and their time scale would be geological. The Ice Age would be a high-level weather event. The second question is: "Are there intermediate-level weather phenomena which have so far escaped human perception, but which, if perceived, could give greater insight into why the weather is as it is?"

From Tornados to Quarks

This last suggestion may sound fanciful, but it is not all that far-fetched. We need only look to the hardest of the hard sciences—physics—to find peculiar examples of systems which are explained in terms of interacting "parts" which are themselves invisible. In physics, as in any other discipline, a *system* is a group of interacting *parts*. In most systems that we know, the parts retain their identities during the interaction, so that we still see the parts inside the system. For example, when a team of football players assembles, the individual players retain their separateness—they do not melt into some composite entity, in which their individuality is lost. Still—and this is important—some processes are going on in their brains which are evoked by the team-context, and which would not go on otherwise, so that in a minor way, the players change identity when they become part of the larger system, the team. This kind of system is called a *nearly decomposable system* (the term comes from H. A. Simon's article "The Architecture of Complexity"; see the Bibliography). Such a system consists of weakly interacting modules, each of which maintains its own private identity throughout the interaction but by becoming slightly different from how it is when outside of the system, contributes to the cohesive behavior of the whole system. The systems studied in physics are usually of this type. For instance, an atom is seen as made of a nucleus whose positive charge captures a number of electrons in "orbits", or bound states. The bound electrons are very much like free electrons, despite their being internal to a composite object.

Some systems studied in physics offer a contrast to the relatively straightforward atom. Such systems involve extremely strong interactions, as a result of which the parts are swallowed up into the larger system, and lose some or all of their individuality. An example of this is the nucleus of an atom, which is usually described as being "a collection of protons and

neutrons". But the forces which pull the component particles together are so strong that the component particles do not survive in anything like their "free" form (the form they have when outside a nucleus). And in fact a nucleus acts in many ways as a single particle, rather than as a collection of interacting particles. When a nucleus is split, protons and neutrons are often released, but also other particles, such as pi-mesons and gamma rays, are commonly produced. Are all those different particles physically present inside a nucleus before it is split, or are they just "sparks" which fly off when the nucleus is split? It is perhaps not meaningful to try to give an answer to such a question. On the level of particle physics, the difference between storing the potential to make "sparks" and storing actual subparticles is not so clear.

A nucleus is thus one system whose "parts", even though they are not visible while on the inside, can be pulled out and made visible. However, there are more pathological cases, such as the proton and neutron seen as systems themselves. Each of them has been hypothesized to be constituted from a trio of "quarks"—hypothetical particles which can be combined in twos or threes to make many known fundamental particles. However, the interaction between quarks is so strong that not only can they not be seen inside the proton and neutron, but they cannot even be pulled out at all! Thus, although quarks help to give a theoretical understanding of certain properties of protons and neutrons, their own existence may perhaps never be independently established. Here we have the antithesis of a "nearly decomposable system"—it is a system which, if anything, is "nearly indecomposable". Yet what is curious is that a quark-based theory of protons and neutrons (and other particles) has considerable explanatory power, in that many experimental results concerning the particles which quarks supposedly compose can be accounted for quite well, quantitatively, by using the "quark model".

Superconductivity: A "Paradox" of Renormalization

In Chapter V we discussed how renormalized particles emerge from their bare cores, by recursively compounded interactions with virtual particles. A renormalized particle can be seen either as this complex mathematical construct, or as the single lump which it is, physically. One of the strangest and most dramatic consequences of this way of describing particles is the explanation it provides for the famous phenomenon of *superconductivity:* resistance-free flow of electrons in certain solids, at extremely low temperatures.

It turns out that electrons in solids are renormalized by their interactions with strange quanta of vibration called *phonons* (themselves renormalized as well!). These renormalized electrons are called *polarons.* Calculation shows that at very low temperatures, two oppositely spinning polarons will begin to attract each other, and can actually become bound together in a certain way. Under the proper conditions, all the current-carrying polar-

ons will get paired up, forming *Cooper pairs.* Ironically, this pairing comes about precisely because electrons—the bare cores of the paired polarons—repel each other electrically. In contrast to the electrons, each Cooper pair feels neither attracted to nor repelled by any other Cooper pair, and consequently it can can slip freely through a metal as if the metal were a vacuum. If you convert the mathematical description of such a metal from one whose primitive units are polarons into one whose primitive units are Cooper pairs, you get a considerably simplified set of equations. This mathematical simplicity is the physicist's way of knowing that "chunking" into Cooper pairs is the natural way to look at superconductivity.

Here we have several levels of particle: the Cooper pair itself; the two oppositely spinning polarons which compose it; the electrons and phonons which make up the polarons; and then, within the electrons, the virtual photons and positrons, etc. etc. We can look at each level and perceive phenomena there, which are explained by an understanding of the levels below.

"Sealing-off"

Similarly, and fortunately, one does not have to know all about quarks to understand many things about the particles which they may compose. Thus, a nuclear physicist can proceed with theories of nuclei that are based on protons and neutrons, and ignore quark theories and their rivals. The nuclear physicist has a *chunked* picture of protons and neutrons—a description derived from lower-level theories but which does not require understanding the lower-level theories. Likewise, an atomic physicist has a chunked picture of an atomic nucleus derived from nuclear theory. Then a chemist has a chunked picture of the electrons and their orbits, and builds theories of small molecules, theories which can be taken over in a chunked way by the molecular biologist, who has an intuition for how small molecules hang together, but whose technical expertise is in the field of extremely large molecules and how they interact. Then the cell biologist has a chunked picture of the units which the molecular biologist pores over, and tries to use them to account for the ways that cells interact. The point is clear. Each level is, in some sense, "sealed off" from the levels below it. This is another of Simon's vivid terms, recalling the way in which a submarine is built in compartments, so that if one part is damaged, and water begins pouring in, the trouble can be prevented from spreading, by closing the doors, thereby sealing off the damaged compartment from neighboring compartments.

Although there is always some "leakage" between the hierarchical levels of science, so that a chemist cannot afford to ignore lower-level physics totally, or a biologist to ignore chemistry totally, there is almost no leakage from one level to a distant level. That is why people can have intuitive understandings of other people without necessarily understanding the quark model, the structure of nuclei, the nature of electron orbits,

the chemical bond, the structure of proteins, the organelles in a cell, the methods of intercellular communication, the physiology of the various organs of the human body, or the complex interactions among organs. All that a person needs is a chunked model of how the highest level acts; and as we all know, such models are very realistic and successful.

The Trade-off between Chunking and Determinism

There is, however, perhaps one significant negative feature of a chunked model: it usually does not have exact predictive power. That is, we save ourselves from the impossible task of seeing people as collections of quarks (or whatever is at the lowest level) by using chunked models; but of course such models only give us probabilistic estimates of how other people feel, will react to what we say or do, and so on. In short, in using chunked high-level models, we sacrifice determinism for simplicity. Despite not being sure how people will react to a joke, we tell it with the expectation that they will do something such as laugh, or not laugh—rather than, say, climb the nearest flagpole. (Zen masters might well do the latter!) A chunked model defines a "space" within which behavior is expected to fall, and specifies probabilities of its falling in different parts of that space.

"Computers Can Only Do What You Tell Them to Do"

Now these ideas can be applied as well to computer programs as to composite physical systems. There is an old saw which says, "Computers can only do what you tell them to do." This is right in one sense, but it misses the point: you don't know in advance the consequences of what you tell a computer to do; therefore its behavior can be as baffling and surprising and unpredictable to you as that of a person. You generally know in advance the *space* in which the output will fall, but you don't know details of where it will fall. For instance, you might write a program to calculate the first million digits of π. Your program will spew forth digits of π much faster than you can—but there is no paradox in the fact that the computer is outracing its programmer. You know in advance the space in which the output will lie—namely the space of digits between 0 and 9—which is to say, you have a chunked model of the program's behavior; but if you'd known the rest, you wouldn't have written the program.

There is another sense in which this old saw is rusty. This involves the fact that as you program in ever higher-level languages, you know less and less precisely what you've told the computer to do! Layers and layers of translation may separate the "front end" of a complex program from the actual machine language instructions. At the level you think and program, your statements may resemble declaratives and suggestions more than they resemble imperatives or commands. And all the internal rumbling provoked by the input of a high-level statement is invisible to you, generally, just as when you eat a sandwich, you are spared conscious awareness of the digestive processes that it triggers.

In any case, this notion that "computers can only do what they are told to do," first propounded by Lady Lovelace in her famous memoir, is so prevalent and so connected with the notion that "computers cannot think" that we shall return to it in later Chapters when our level of sophistication is greater.

Two Types of System

There is an important division between two types of system built up from many parts. There are those systems in which the behavior of some parts tends to *cancel out* the behavior of other parts, with the result that it does not matter too much what happens on the low level, because most anything will yield similar high-level behavior. An example of this kind of system is a container of gas, where all the molecules bump and bang against each other in very complex microscopic ways; but the total outcome, from a macroscopic point of view, is a very calm, stable system with a certain temperature, pressure, and volume. Then there are systems where the effect of a single low-level event may get *magnified* into an enormous high-level consequence. Such a system is a pinball machine, where the exact angle with which a ball strikes each post is crucial in determining the rest of its descending pathway.

A computer is an elaborate combination of these two types of system. It contains subunits such as wires, which behave in a highly predictable fashion: they conduct electricity according to Ohm's law, a very precise, chunked law which resembles the laws governing gases in containers, since it depends on statistical effects in which billions of random effects cancel each other out, yielding a predictable overall behavior. A computer also contains macroscopic subunits, such as a printer, whose behavior is completely determined by delicate patterns of currents. What the printer prints is not by any means created by a myriad canceling microscopic effects. In fact, in the case of most computer programs, the value of every single bit in the program plays a critical role in the output that gets printed. If any bit were changed, the output would also change drastically.

Systems which are made up of "reliable" subsystems only—that is, subsystems whose behavior can be reliably predicted from chunked descriptions—play inestimably important roles in our daily lives, because they are pillars of stability. We can rely on walls not to fall down, on sidewalks to go where they went yesterday, on the sun to shine, on clocks to tell the time correctly, and so on. Chunked models of such systems are virtually entirely deterministic. Of course, the other kind of system which plays a very large role in our lives is a system that has variable behavior which depends on some internal microscopic parameters—often a very large number of them, moreover—which we cannot directly observe. Our chunked model of such a system is necessarily in terms of the "space" of operation, and involves probabilistic estimates of landing in different regions of that space.

A container of gas, which, as I already pointed out, is a reliable system

because of many canceling effects, obeys precise, deterministic laws of physics. Such laws are *chunked laws*, in that they deal with the gas as a whole, and ignore its constituents. Furthermore, the microscopic and macroscopic descriptions of a gas use entirely different terms. The former requires the specification of the position and velocity of every single component molecule; the latter requires only the specification of three new quantities: temperature, pressure, and volume, the first two of which do not even have microscopic counterparts. The simple mathematical relationship which relates these three parameters—$pV = cT$, where c is a constant—is a law which depends on, yet is independent of, the lower-level phenomena. Less paradoxically, this law can be derived from the laws governing the molecular level; in that sense it depends on the lower level. On the other hand, it is a law which allows you to ignore the lower level completely, if you wish; in that sense it is independent of the lower level.

It is important to realize that the high-level law cannot be stated in the vocabulary of the low-level description. "Pressure" and "temperature" are new terms which experience with the low level alone cannot convey. We humans perceive temperature and pressure directly; that is how we are built, so that it is not amazing that we should have found this law. But creatures which knew gases only as theoretical mathematical constructs would have to have an ability to synthesize new concepts, if they were to discover this law.

Epiphenomena

In drawing this Chapter to a close, I would like to relate a story about a complex system. I was talking one day with two systems programmers for the computer I was using. They mentioned that the operating system seemed to be able to handle up to about thirty-five users with great comfort, but at about thirty-five users or so, the response time all of a sudden shot up, getting so slow that you might as well log off and go home and wait until later. Jokingly I said, "Well, that's simple to fix—just find the place in the operating system where the number '35' is stored, and change it to '60'!" Everyone laughed. The point is, of course, that there is no such place. Where, then, does the critical number—35 users—come from? The answer is: *It is a visible consequence of the overall system organization—an "epiphenomenon".*

Similarly, you might ask about a sprinter, "Where is the '9.3' stored, that makes him be able to run 100 yards in 9.3 seconds?" Obviously, it is not stored anywhere. His time is a result of how he is built, what his reaction time is, a million factors all interacting when he runs. The time is quite reproducible, but it is not stored in his body anywhere. It is spread around among all the cells of his body and only manifests itself in the act of the sprint itself.

Epiphenomena abound. In the game of "Go", there is the feature that "two eyes live". It is not built into the rules, but it is a consequence of the

rules. In the human brain, there is gullibility. How gullible are you? Is your gullibility located in some "gullibility center" in your brain? Could a neurosurgeon reach in and perform some delicate operation to lower your gullibility, otherwise leaving you alone? If you believe this, you are pretty gullible, and should perhaps consider such an operation.

Mind *vs.* Brain

In coming Chapters, where we discuss the brain, we shall examine whether the brain's top level—the mind—can be understood without understanding the lower levels on which it both depends and does not depend. Are there laws of thinking which are "sealed off" from the lower laws that govern the microscopic activity in the cells of the brain? Can mind be "skimmed" off of brain and transplanted into other systems? Or is it impossible to unravel thinking processes into neat and modular subsystems? Is the brain more like an atom, a renormalized electron, a nucleus, a neutron, or a quark? Is consciousness an epiphenomenon? To understand the mind, must one go all the way down to the level of nerve cells?