CHAPTER 3

Fundamental Concepts

(from William Curtis's *How to Improve Your Math Grades,* on occampress.com)[1]

1. This line has been added to the first page of the chapter because Google does not reveal this information when it makes the chapter accessible following a user's search.

Importance of Fundamental Concepts

The means of overcoming the difficulties we have considered so far is by the use of Environments. Roughly speaking, an Environment is a new way of organizing your course notes, and adding to them, so that you can always find the information you want very quickly. Environments are explained in the next few chapters, but, on the basis of years of experience in constructing Environments, I have come to the conclusion that it is essential to have a clear idea of what the fundamental concepts are that we use in this task. If you are sorely pressed for time, you can skip this chapter and go on to the next, then come back to this one later on. But you should read this chapter sooner or later.

I sometimes feel that the concepts in this chapter should be called "the fundamental concepts of Western civilization", although that would be an exaggeration, since there are certainly other fundamental ones as well. But they are so important in all technical subjects — in all technical *thinking* — that if you do nothing more than understand them and start to apply them, you will have gotten your money's worth out of this book.

The first of these fundamental concepts is the *What* vs. the *How*, or semantics vs. syntax.

Fundamental Concept 1: the What versus the How, or Semantics versus Syntax

Examples

We will begin with a few examples.

Example 1

The following story has been told many times. I do not know its source — it seems to have first appeared in the early 1970s— or if this version is the same as the original one, but that is not important as far as the point of the story is concerned.

In a high-school physics exam, the question was asked, "Suppose you had only a barometer and were asked to measure the height of a very tall building. How would you do it?"

The teacher received various answers from his students, including the one he wanted, which was that you should take the barometer to the top of the building, note the air pressure it indicates, then use a table to convert that air pressure into altitude

(height). However, one answer he didn't expect was the following: "I would tie the barometer to the end of a very long string, go to the top of the building, lower the barometer until it touched the ground, then measure the length of the string."

The teacher marked the answer wrong. The student protested. The teacher agreed to give the student another chance. This time, the student answered: "I would go to the top of the building, drop the barometer off, and time how long it took to reach the ground. Then I would use the formula

$$s = \frac{gt^2}{2}$$

where:

g is the gravitational constant,
t is the number of seconds the barometer took to reach the ground, and
s is the height of the building."

Again the teacher, not getting the answer he wanted, marked the answer wrong. Again the student protested, and now his parents joined in. Eventually the teacher again relented and agreed to give the student one last chance. This time the student wrote the following on his answer sheet: "I would go to the superintendent of the building and say, 'Here, Mr. Superintendent, I will give you this nice new barometer if you will tell me how tall this building is.'"

How long the controversy raged, or what its final outcome was, I do not know, but the lesson of the story is clear, namely, that there is seldom only one way to do something. Or, in other words, for a given What (finding the height of a very tall building), there are usually many Hows.

The What: measure the height of a very tall building.

The Hows: the three ways described. You can probably think of at least one or two additional ones.

Here are a few more examples.

Example 2

The What: Go from your house to work or school.

The Hows: Go by car, bus, train, bicycle, on foot, or use some combination of these. Of course, some ways are better than others, depending on what is most import-

ant to you at the time (e.g., speed of getting there, convenience, cost, not polluting the environment).

Example 3

The What: Obtain a bachelor's degree in computer science.

The Hows: All the different colleges and universities you could attend. Some, of course, are better than others (e.g., will enable you to get a higher paying job, or are cheaper).

Example 4

The What: Solve the following quadratic equation for x:

$x^2 + x - 2 = 0$.

The Hows: Among these are:

1. Use the quadratic formula we learned in high school math courses. It works for any quadratic equation:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

where a is the coefficient of x^2 (hence $a = 1$ in our example), b is the coefficient of x (hence $b = 1$ in our example), and c is the constant standing alone (hence $c = -2$ in our example).

We find that $x = 1$, $x = -2$, are the solutions.

2. Factor the left-hand side of the quadratic equation by trial and error and find that

$(x - 1)(x + 2) = 0$,

hence

$x = 1$, $x = -2$ are the solutions.

Semantics versus Syntax

Other terms for the *What* versus the *How* are *semantics* versus *syntax*. In normal everyday circumstances, *what* you want to say (what you mean, i.e., the semantics) can usually be said in many ways. A grammar book for a natural language gives you the rules for stringing words together, i.e., the syntactic rules, but it only incidentally discusses the meanings of the strings of words.

In computer programming, the distinction between *What* and *How* is clearer. The *What* is the function, e.g., addition or subtraction or multiplication or division or sorting a set of numbers or accessing data from a data base or displaying information on a computer screen in some format; the *How*s are the various programs that can compute (implement) the function. Some programs, of course, are better than others for the goals at hand; some produce an answer faster than others, some programs are easier to write and test than others, some require less memory space, etc.

In mathematics, the distinction between *What* and *How* is similarly clear. For example, each integer (the *What*) can be expressed in an infinite number of ways (*How*s). Take, e.g., the number 2:

$$2 = 3 - 1$$

$$= 4 - 2$$

$$= 5 - 3$$

$$= 6 - 4$$

.
.
.

$$= -18 + 5 - 14 + 1 + 29 - 1$$

...

If you ask, "But what, then, *is* the number 2, or any integer for that matter?" you are asking a question that was only answered satisfactorily in the early years of the twentieth century. In brief, the number 2 is defined as the set of all pairs, the number 3 as the set of all triples, the number 4 as the set of all quadruples, etc. If you are not familiar with this definition, or are bothered by its apparent circularity, see, e.g., Russell, Bertrand, *Principles of Mathematics*, W. W. Norton & Co., Inc., N.Y.,1903, pp. 114 ff.

Another example of *What* vs. *How* is the following. Suppose, for your next homework assignment, you had access, say, via phone, to a professor who would do no teaching or explaining, but would answer any request for specific information. Thus, e.g., when you were attempting to solve a problem, you could ask him or her, "Is ... a theorem?", and he or she would give you the correct yes or no answer. Similarly, if you asked him or her, "What is the formula for...?", he or she would either give you the formula, or else tell you that no general formula was known. You could also ask questions such as, "Please give me a list of all the theorems and lemmas that contain the term ... or a synonym of it."

I think you will agree that, even though the professor is not explaining how to solve the problem, the *What*s you can get from him or her make it much easier for you to figure out the *How* of a solution to a problem. The Environments you will learn to construct in the next few chapters are simply limited implementations of such a Question-Answering professor's function.

Knowing how to recognize the difference between the *What* and the *How* is one of the most important skills you can have, and one that will serve you in fields outside of mathematics. In industry, a common, and very expensive, failing of managers and the engineers who work for them, is the belief that there is only one way to develop a new product, namely, by designing it from scratch. But in fact the best design solution is often to do *as little* original design as possible, and to use *as many* existing parts and assemblies as possible— something that most new engineers do not like to realize. But that realization can only come to one who is used to asking himself, *What* is the goal here? and then, *How* can we go about accomplishing it? and then, Of all the ways of accomplishing it, which are the best for our purposes?

Representation

The issue of representation is closely related to *What* vs. *How*, and it will make your life in mathematics easier if you learn to recognize when you are confronting what is fundamentally a question of representation. We begin with a familiar example.

Consider the mathematical entity known as a "vector". Intuitively, we may think of a vector as an arrow of a certain length pointing in a certain direction relative to a fixed set of axes. There are many ways of representing a vector, and beginning courses spend time showing you several, e.g., as an ordered *n*-tuple of numbers, e.g., <2, 5>, <6, – 4, 21>, or as a vector-length and an angle, e.g., <2, 30°>.

Here are some other examples of the representation concept:

• Any program that computes a given function can be considered a representation of that function, although in this case we normally speak of the program as an *implementation* of the function.

• Any infinite series that converges can be considered a representation of the number it converges to.

• A labelled drawing can be considered an (informal) representation of a mathematical idea.

• All the textbooks on a given subject in mathematics can be considered representations of (some of) the mathematical concepts that constitute that subject. A good question to think about — and one that we will address later — is, "Why are textbooks on the same subject different, apart from the differences in their coverage of the subject?" If your answer is, "Because their explanations are different", keep reading this book!

We will conclude this section with a brief summary of some of the *What*s vs. *How*s we have mentioned:

Table 1: What vs. How in Various Domains

Domain	What	Hows
Everyday life	Task	Ways of accomplishing task
Natural language	Meaning (semantics)	Grammatical ways (syntactically correct ways) of expressing Meaning
Computers	Function	Programs to compute Function
Mathematics	Theorem	Proofs of Theorem

Fundamental Concept 2: Structure, or Breaking Complex Things into Simpler Things

The advantage of breaking complex things into simpler things is so taken for granted in Western culture that we usually consider it obvious. We break speech into certain constituent sounds, then represent those sounds with strings of letters. We (and nature) break matter into molecules, then molecules into atoms, then atoms into subatomic particles. We break a large business corporation into divisions, then each division into various departments, e.g., Research and Development, Marketing, Manufacturing. In the U.S., we break the government into the Executive, Legislative,

and Judicial branches. All perfectly obvious. But the recognition of the importance of the concept — the recognition of its nearly universal applicability — was, as far as I know, limited to the West until Western culture began to spread throughout the world in the nineteenth century. Perhaps, as Marshall McLuhan, the sixties philosopher of communications media argues, the reason this was so clear to the West was that, from the time of the ancient Greeks, literate Western man has had the benefits of an alphabetic writing system staring him in the face. I don't know. But in any case, this is an example of an idea whose importance is in no way diminished by the fact that its value is obvious.

Breaking a complex thing down into smaller things — or structuring a thing — is a way of simplifying intellectual labor. In particular, a *How* can be broken down into a set of simpler *What*s, and the *How* for each of those *What*s can in turn be broken down into a set of even simpler *What*s until, eventually, the *What*s are things we know how to do. To take a simple example, consider the rule in elementary calculus for finding the derivative of a sum, i.e., the rule that enables us to evaluate

$$\frac{\mathrm{d}}{\mathrm{d}x}(u_1 + u_2 + \ldots + u_n)$$

where each u_i , $1 \le i \le n$, is a function solely of x.

You may know the rule, which, stated informally, is, "the derivative of a sum is the sum of the derivatives", i.e.,

$$\frac{\mathrm{d}}{\mathrm{d}x}(u_1 + u_2 + \ldots + u_n) = \frac{\mathrm{d}}{\mathrm{d}x}(u_1) + \frac{\mathrm{d}}{\mathrm{d}x}(u_2) + \ldots + \frac{\mathrm{d}}{\mathrm{d}x}(u_n)$$

Here, we have broken down the initial *What* (the initial sum) into simpler *What*s.

I would like now to give you a much larger example of the power of breaking things down. The example is important because it illustrates some of the ideas you will be using to develop Environments. It is taken from Peter Schorer's, *How to Create Zero-Search-Time Computer Documentation* (available online at zsthelp.com).

One field in which complexity (i.e., intellectual labor) is always in danger of overwhelming those who work in the field is computer science, in particular, computer programming. Thus, soon after the art of programming came into being in the forties,

programmers recognized the need to break programs up into more easily manageable pieces that became known as *subroutines*. A subroutine performs a specific task, e.g., properly handling the carries in addition, or printing the result of a calculation. Strangely enough, the importance of structure, or, rather, of certain kinds of structure over other kinds, did not become clear to the programming community until the early 1970s. The story, in brief, is that in the March 1968 issue of *Communications of the ACM* (the Association of Computing Machinery), there appeared a letter to the editor titled, "Go To Statement Considered Harmful." A go-to statement is a statement in a computer program that commands the computer to execute not the next sequential statement in the program, but some other statement elsewhere in the program. In effect, it enables the programmer to introduce all sorts of special cases into the process by which the program performs its computation; it enables him or her to make the program "jump all over the place" during the course of a computation.

According to the author of the letter, computer scientist Edsger Dijkstra, this leads to programs that are difficult to understand, hence difficult to check for correctness, and difficult to debug. Hence the go-to statement should be avoided as much as possible.

This letter is generally considered the beginning of the programming methodology now known as *structured programming* (also known as *top-down programming*). In 1972, Dijkstra, along with O.-J. Dahl and C. A. R. Hoare, published a book, *Structured Programming*, which set forth the structured programming methodology in more detail. (This methodology was based on the use of block-structured, or Algol-like (nowadays, Pascal-like, or C-like) languages.) In 1976, Dijkstra developed the idea still further, introducing a method of writing correct programs (i.e. programs that compute the function we want them to compute) by, in effect, writing them in a way that permits the programmer to easily prove the correctness of each successive approximation to the final, complete program.

As with many new ideas, the technique of structured programming, as well as its value, was obvious after it had been pointed out. In fact, structured programming is nothing but the application to programming of the old technique of outlining that writers of term papers learn to use (sooner or later). Structured programming is simply a method of breaking down a large programming task into a manageable set of smaller programming tasks, and then breaking each of these down into a manageable set of still smaller programming tasks, etc., and doing so in a way that enables the programmer to prove the correctness of the program as it develops. (The breaking down process is sometimes called "chunking" in programming circles.) In other words, structured programming is a prime example of breaking a given *What* down into a *How* consisting of smaller *What*s.

The idea of a structured program lies at the very root of the Environment method of improving your math grades described in this book. An Environment is like a structured program with you as the computer that executes it. This is an important point, and I will explain why.

The traditional view of a computer, its software, and the user of both, is shown in Fig. 2-1a. Here, the computer is a machine (a "black box") *over there* that is capable of solving certain problems. The user of the machine is a separate entity that brings problems to the machine for it to solve — specifically, gives inputs to the software in the form of commands and data that may be in a file, or accessed by the software from some specified remote source, or else manually input by the user. The user starts the software and hardware running (the hardware being a central processing unit (cpu) and memory), and together they then perform various computations, or, more correctly, information processings, and produce the result as output. The computing entity, in this view, is the computer and the software, as shown by the dotted line in the figure.

The new view, and one that leads to the design of software that is much easier to use, is that shown in Fig. 2-1b. Here, the computing entity is hardware, software, *and user*. The *user* is the "central processing unit" that runs the hardware and software. The user's "program" is the user Environment — the keyboard plus the terminal screen plus the manuals. (I will write this type of program in quotation marks because it is not a program in the formal sense of the word, i.e., it is not necessarily capable of being executed by a machine.) The input is given to this entire entity — by another person who gives it to the user, or by the user bringing it him- or herself. Similarly, the output is an output from this entire entity.

All of which is summed up by the motto, "Not human alone, or machine alone, but human and machine as a unit" — a motto for the design of the computers and software of the future.

Viewed in this way, it is clear that the "program" (Environment) that makes computer *and* user solve a given problem is only partially the software that runs the computer. The Environment must also tell the user how to operate the computer so that the computer's software can then complete the work of solving the problem.

The "program" (i.e., Environment) is what computer scientists call *non-deterministic*, in the sense that there are usually a variety of ways by which the user can achieve the desired result: he or she may have a variety of different programs to use, some of which may be interactive, meaning that the user may guide the computation as it proceeds (this type of operation is also known as *on-line*); some may not be interactive, meaning that once the computation is started, it runs to completion without user intervention; this type of operation is also known as *off-line* or *background*.

Several things become clear from this new point of view. First of all, the obsession of most computer hardware designers and programmers with speed of computation — i.e., the speed at which the hardware cpu can execute program instructions — misses the point. Equally important, and probably more so, is the average (or typical) speed at which a given type of problem can be solved or type of "job" can be processed, e.g., a job such as typing and printing a report, or obtaining a certain type of information from a data base. The speed to increase is the speed at which the average user *plus* the computer can solve a given problem. This speed must be averaged over *all* users, including first-time users. In the computer industry, the rate at which jobs can be processed through a system is sometimes called the *throughput*. The speed measurement must include all training, if training is in fact necessary in order for first-time users to use the system, and/or all time spent reading the manuals, and/or all time getting help from the manufacturer's customer support service. It is by no means always the case that increasing the computer's speed will increase the throughput, especially when it takes hours, even days, to figure out how to make the computer solve the problem in the first place.

An analogy to Environments outside the field of computers is that of an astronaut in a space suit exploring the moon. The astronaut alone does not carry out such exploration, nor does the space suit alone; both together do.

In the case of mathematics (and all technical subjects) "throughput" is the rate at which you can solve problems.

In mathematics, the "problem solver" is not you the student, nor is it the textbook or lecture notes or computer or calculator you happen to use, it is you *plus* these, considered as a unit. The Environments you will learn how to build and use in this book, are (partial) "programs" that enable you to solve certain types of problems. You *plus* the Environment together, are the problem solver. You are the (intelligent) "computer" that the Environment is a "program" for.

I should tell you that the idea of regarding the user of a computer Environment as a kind of "central processing unit" (cpu) that executes the non-deterministic "program" that is the Environment, in order to solve the problem represented by the input — that this idea is considered a giant step backward by some computer scientists, especially by those with a vested interest in some of the woolier branches of the discipline, e.g., cognitive science. "Man is not a machine!" these researchers proclaim. But the idea of an Environment in no way is aimed at making the computer user do dull, repetitive machine-like work. It is aimed rather at rendering "look-upable" as much as possible of the information required for the use of a computer system — or for solving problems in a mathematical subject.

In the case of an Environment for a mathematics subject, at present, the user (meaning you, the student), typically only has a pocket-calulator in the way of hardware, and thus has to do more of the problem-solving work for him- or herself.

I should also tell you that most teachers of mathematics do not like the idea of a "program" for students to use in solving problems. They feel that such an idea suggests to the student that he or she need not "learn" the subject. But, paradoxical as it may seem, "learning" in this sense has been one of the greatest detriments to improving the problem-solving ability of students. We will discuss this further in "Criticisms of the Environment Concept" on page 127.

Once you start creating Environments, you will find that you become more and more "structure conscious". In particular, you will find yourself more and more thinking in terms of partial solutions, i.e., approximations to a complete solution. You will come to realize that problem solving is never an all or nothing matter. Another way of saying this is that you will think of the task of solving a problem as a matter of progressively narrowing your ignorance of the (or a) solution. At first you begin with a "big ignorance". Then you break this big ignorance into smaller ignorances such that, if you could do away with each of these smaller ignorances, then the big ignorance would also be done away with.

Such a description of problem solving may bother you if you are a particularly ambitious student. You may feel that, if you were *really* OK, then you would be able to achieve perfection. A partial solution, for you, is no solution at all, since you know that those who will really matter in later life are those who always get the whole answer. ("It's not winning or losing that counts, it's winning!") I can only reply to this by telling you that, in my experience, pursuit of perfection has produced far less success than pursuit of the *least bad solution to the problem that time, circumstances, and my own abilities permit*. Pursuit of perfection is, in general, an enormous waste of energy because it means that you are prepared to spend as much time on raising your grade from an A to an A+ as you are on raising it from a C– to an A. Pursuit of perfection means that, to you, everything is equally important. It also means that you are sure you know what perfect is!

Every student of mathematics or of any technical subject, should keep in mind the Moon Lander that, in 1969, put the first human beings on the moon. When the Lander was first being designed, the idea of what it should look like came straight out of science-fiction ideas of what a space-travel rocket should look like, namely, pointed at the front, smoothly widening to a fixed diameter, then cylindrical — the whole thing smooth and streamlined. Then, one of the designers realized that the reason rockets are designed that way is that they have to travel at high speeds through the earth's atmosphere. But there is no atmosphere on the moon! Therefore the Moon Lander did

not have to be streamlined. The result was what we have have seen in photographs and on television — a craft that looks like a crudely built sheet-metal hut on stilts. But it worked. It was, among other things, a fine example of the virtue of putting the What before the How.

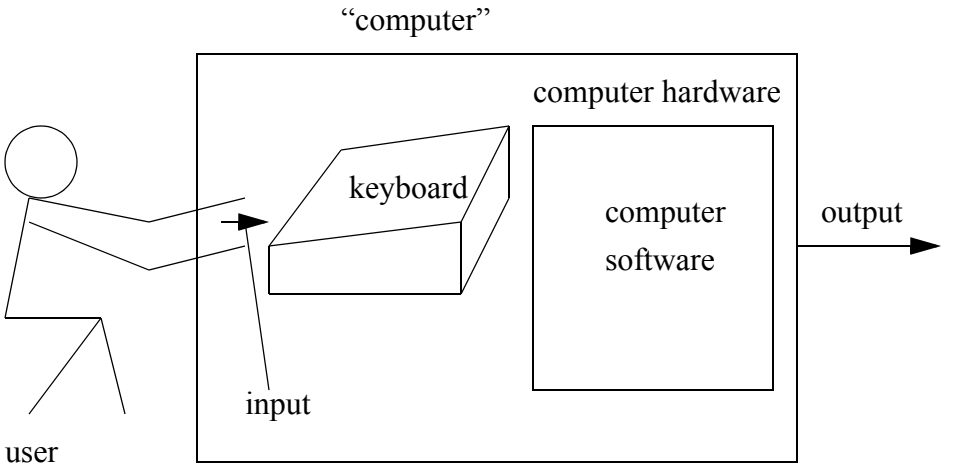
Or consider the DC-XA reusable rocket, which began its initial testing on the White Sands Missile Range, N.M. in spring, 1996:

"From a few hundred feet away, the $50 million DC-XA looks clean and crisp. Up close it looks like the well-used low-budget vehicle that it is. The seams are crooked. Access panels are dented. The bright white insulation sprayed on its bottom — commercial stuff called 'Power Scrub' from a New Mexico company — is lumpy and irregular.

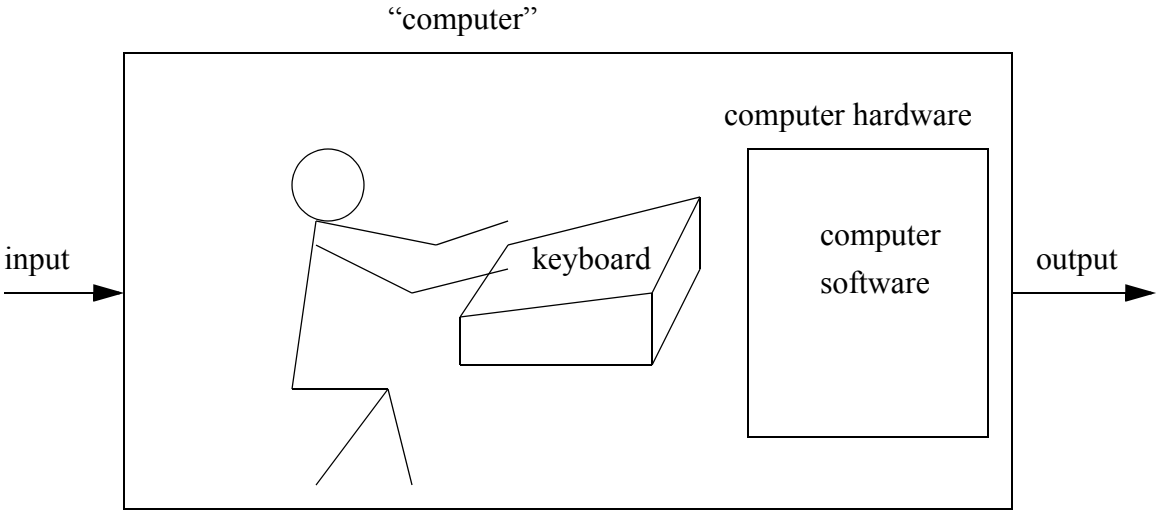
"'That's how we do things,' said technology development engineer Bruce Leonard. "It doesn't have to be perfect. It just has to be good enough." — Petit, Charles, "Unusual Reusable Rocket Getting Plenty of Reuse", in *San Francisico Chronicle*, June 11, 1996, p. A4.

Pursuit of perfection is usually a waste of time and energy. I prefer the motto of the manager of a Research and Development Dept. I once worked in, a man whose position forced him to be concerned with achieving results: "Good enough is perfect."

Figure 2-1. Old and New Views of User and Computer



a) Old View of User and Computer



b) New View of User and Computer

Fundamental Concept 3: Algorithms and Heuristics

An algorithm is a mechanical procedure for producing an answer to a problem in a mathematically-based subject. The phrase *mechanical procedure* means a procedure that can be carried out by a machine. (This definition can be made more precise using the concept of an idealized, simplified computer called a "Turing machine," which is named after the British mathematician who first used it in certain proofs in formal logic in the thirties.) Actually, an algorithm is not merely a mechanical procedure, but one that is guaranteed to produce an answer no matter what the input. As it turns out, there are many mechanical procedures that will produce answers for *some* inputs, but not for all; in these latter cases the procedure may, e.g., simply repeat certain steps over and over, forever. Some examples of algorithms are: the familiar rules for adding, subtracting, multiplying, and dividing that students are taught in grammar school; variations on these rules that are implemented in pocket calculators and computers; the procedures, implemented as programs, that operate automatic tellers in banks; just about all procedures, implemented as programs, for processing information in business, e.g., inventory control, payroll check writing, employee record keeping, etc.

A heuristic, on the other hand, is a procedure, and not necessarily one that can be performed by a machine, which may or may not yield an answer, but which experience suggests will do so in most cases. *Webster's New Collegiate Dictionary* defines the term *heuristic* as "of or relating to exploratory problem-solving techniques that utilize self-educating techniques (as the evaluation of feedback) to improve performance." In this book, I will sometimes use the term *procedure* instead of *heuristic*.

As you create Environments, you will become skillful at developing — at *recognizing* — good heuristics. At the very least, a good heuristic is one that decreases the amount of trial and error you have to go through in most cases. A good heuristic typically says, "First try to find the answer, by trial and error, using only this small set of possibilities; then, if that doesn't work, try to find the answer, by trial and error, on this other small set of possibilities; then, if that doesn't work..." An Environment is, among other things, an attempt to reduce the amount of low-level intellectual labor involved in such trial and error searches — low-level labor such as looking up the meanings of terms, finding all the theorems that concern a given concept, recollecting techniques for proving theorems, etc.

Fundamental Concept(s) 4: Program Structures

I said earlier in this chapter that an Environment *can be* regarded as a kind of non-deterministic program which you, the student, "execute" in order to solve problems. You might reply that the task of solving all but the simplest math problems is far too complicated to be regarded as something that can be performed by a person following the steps in a "program". You might argue that programming concepts apply to machines, not to people. The best counterargument I can give is an essay that is quoted in Schorer's *How to Create Zero-Search-Time Computer Documentation* (pp. 95 - 98) (available online at zsthelp.com).

"Mother's Day Meditations from the Computer Room"

by Gertrude Martin's son David (martindm@acm.org)

"A friend recently asked me what training it takes to work with computers. I gave a brief answer mentioning some college courses, some on-the-job training, and a long time in the school of hard knocks. But upon reflection, I realize that most of my training in fundamental computer concepts came from my mother.

"When I was a baby, mother taught me about input buffering: 'Don't try to stuff all your food in your mouth at once. Leave it on your plate until you're ready to eat it, and then take it in one mouthful at a time.'

"She also taught me about processing the entire input buffer before going on to the next step: 'Eat everything on your plate. Then you can have dessert.'

"(It will occur to some readers that mother also taught me about output buffering, but I'd like to keep these meditations G-rated.)

"When I was about four, mother introduced the concept of sequentially executed instructions: 'We're going to set the table.' (That's identification of the procedure.) 'First put the table cloth on the table. Check it to make sure it's straight. Then put a plate at each place. Then put a cup at each place. Then...'

"Later, mother introduced the concept of a procedure call: 'We're going to have dinner. Please set the table.'

"Still later, when I was about 14, mother would set up tasks for me and use 'job control language' in a note on the refrigerator door: 'We're going to have dinner at 6:00. You make it when you get home from school. The menu is pinned up on the bulletin board, the meat is in the refrigerator, and I've put the rest of the food out on the counter. Set an extra place — Uncle Jack is coming tonight.'

"Mother demonstrated what it means to multi-process: She could deal with the interruptions of four children (those were the real-time, foreground tasks) while doing the housework (as a background task).

"Mother used the concept of hierarchical storage for her cooking tools. The cooking forks and spoons were hung on hooks right by the stove. The potato slicer and the egg beater, which weren't used for every meal, were kept in a drawer. And the big roaster, which she only used once a year to cook the Thanksgiving turkey, was kept in the storage closet in the basement.

"Once we had about fourteen people for Thanksgiving dinner, and our kitchen seemed too small for the job. That's when mother introduced the concept of backing store. She cleared off the ping-pong table in the rec room next to the kitchen and laid out all her ingredients on one side of the net. My sister and I fetched things from the 'input' side of the ping-pong table as mother called for them, carried partially finished dishes to and from the 'backing store' on the other side of the net, and delivered finished food to the 'output' dining table.

"This system worked well, until my sister and I collided in the doorway between the two rooms and we nearly lost the creamed onions. Mother solved this problem of 'channel contention' by establishing a protocol: 'First say "May I come through?" and then wait until you get the answer "Yes, it's clear."'

"It was also in the kitchen that mother taught me about looping and testing: 'Cook the fudge, while stirring it, and test it every couple of minutes to see if it's done. You test it by dropping a bit of it in the cold water. When it forms a soft ball, it's done.'

"For years I badgered my mother with questions about whether Santa Claus is a real person or not. Her answer was always 'Well, you asked for the presents and they came, didn't they?' I finally understood the full meaning of her reply when I heard the definition of a virtual device: 'A software or hardware entity which responds to commands in a manner indistinguishable from the real device.' Mother was telling me that Santa Claus is a virtual person (simulated by loving parents) who responds to requests from children in a manner indistinguishable from the real saint.

"Mother also taught the IF...THEN...ELSE structure: 'If it's snowing, then put your boots on before you go to school; otherwise just wear your shoes.'

"Mother explained the difference between batch and transaction processing: 'We'll wash the white clothes when we get enough of them to make a load, but we'll wash these socks out right now by hand because you'll need them this afternoon.'

"Mother taught me about linked lists. Once, for a birthday party, she laid out a treasure hunt of ten hidden clues, with each clue telling where to find the next one and the last one leading to the treasure. She then gave us the first clue.

"Mother understood about parity errors. When she counted socks after doing the laundry, she expected to find an even number and groaned when only one sock of a pair emerged from the washing machine. Later she applied the principles of redundancy engineering to this problem by buying our socks three identical pairs at a time.

This greatly increased the odds of being able to come up with at least one matching pair.

"Mother had all of us children write our Christmas thank you notes to Grandmother, one after another, on a single large sheet of paper which was then mailed in a single envelope with a single stamp. This was obviously an instance of blocking records in order to save money by reducing the number of physical I/O operations.

"Mother used flags to help her manage the housework. Whenever she turned on the stove, she put a pot holder on top of her purse to remind herself to turn it off again before leaving the house.

"Mother knew about the devices which raise an interrupt signal to be serviced when they have completed any operation. She had a whistling teakettle.

"Mother understood about LIFO ordering. In my lunch bag she put the dessert on the bottom, the sandwich in the middle, and the napkin on top so that things would come out in the right order at lunchtime.

"There is an old story that God knew He couldn't be physically present everywhere at once, to show His love for His people, and so He created mothers. That is the difference between centralized and distributed processing. As any kid who's ever misbehaved at a neighbor's house finds out, all the mothers in the neighborhood talk to each other. That's a local area network of distributed processors that can't be beat."

If programming concepts can be applied to such ordinary, everyday activities as are described in the quoted essay, then maybe they also apply to the solving of mathematical problems.

Fundamental Concept 5: The Concept of Function

The concept of *function* has been called the single most important discovery (or invention) of modern mathematics, i.e., of mathematics since, say, 1600. The definition that students nowadays learn in their first year of college — a function is a set of ordered pairs, $<x, y>$, such that each x has only one y associated with it — this definition seems so simple that it is hard to believe it was not obvious all along. But it was not. "This clear daylight is now a matter of course, but it replaces an obscurity of midnight," said J.E. Littlewood, one of the 20th century's great mathematicians (Littlewood, J. E., *Littlewood's Miscellany*, Cambridge University Press, N.Y., 1990, p. 79.)

The idea of the function didn't really exist before Descartes' book on analytical geometry *La Géométrie* in 1637, which showed how a geometric curve could represent the dependence of one variable (y) on another (x), i.e., showed how a curve could represent y as a function of x. The term "function" was first used by Leibnitz (co-dis-

coverer, with Newton, of the differential and integral calculus) in 1694. The notation $f(x)$, as in $y = f(x)$, was first used by Euler in 1734.

The function concept lay at the basis of some of the most of the important theoretical developments in programming theory during the sixties and seventies.

During the late eighties, the concept — as expressed by the idea of tasks —was the first breath of fresh air to blow through the stodgy craft of technical writing and through the field that is now subsuming it, namely, computer-human interface design. The set of *tasks* (functions) *that the equipment enables the user to perform* is now seen by a growing number of workers in the field as the correct point of departure for designing computers that are easy to use.

There are engineers who will disagree with me, but I think one of the sure signs of a mature thinker in any technical subject is the ability to "think functionally", in other words, to think first about the *What*, and then, only secondarily, to think about the *How.* Yet even at this late date, electronics courses and textbooks still mix the *How* and the *What*. The *What* is simply various functions on direct and alternating currents and voltages (and why not consider direct current as alternating current with a frequency of zero?). The hardware that performs these functions (inplements the *How*) includes resistors, capacitors, inductors, filters, attenuators, frequency converters, transformers. The subject of electronics really boils down to the study of mappings (functions) from the first set of functions to organizations of the second (i.e., to circuits).

When we first confront the function concept in mathematics, a function is something that "produces" a number (the dependent variable) out of one or more other numbers (the independent variables); it "converts" one or more numbers into another number. This view of function is reinforced when we study other technical subjects, e.g., engineering. However, in mathematics the function concept serves another, deeper, purpose, namely, that of providing us with a way of determining if one thing is "like", or "equivalent" to another thing in a given subject. Thus, in modern algebra, two sets are "equivalent" if there exists the type of function called an "isomorphism" between them; in topology, two sets are "equivalent" if there exists the type of function called a "homeomorphism" between them; in some theories of computation, two functions (programs) are "equivalent" if they can both be reduced to the same normal form. In your problem-solving efforts, it is sometimes very helpful to be able to show that something is like, or equivalent to, something else, and to do this, all you need to do is find an appropriate function!

Fundamental Concept 6. Pictures to Illustrate Technical Concepts

Just about every author who writes a book on doing mathematics emphasizes the importance of pictures as an aid in solving problems and in understanding mathematical ideas. The majority of mathematicians themselves do a great deal of pictorial thinking:

"If I can't picture it, I can't understand it." — Einstein, Albert, quoted by Wheeler in "Profile: Physicist John A. Wheeler," *Scientific American*, June, 1991, p. 36.

"Some mathematicians, perhaps 10 per cent, think in formulae. Their intuition deals in formulae. But the rest think in pictures; their intuition is geometrical. Pictures carry so much more information than words." — Stewart, Ian, *Concepts of Modern Mathematics*, Penguin Books, 1981, p. 5.

One of the 20th century's leading mathematicians said:

"When I think about mathematical ideas, I see the abstract notions in symbolic pictures. They are visual assemblages, for example, a schematized picture of actual sets of points in a plane. In reading a statement like 'an infinity of spheres or an infinity of sets', I imagine a picture with such almost real objects, getting smaller, vanishing on some horizon." — Ulam, S. M., *Adventures of a Mathematician*, Charles Scribner's Sons

And yet, as one of the 20th century's great mathematicians, has said:

"A heavy warning used to be given that pictures are not rigorous; this has never had its bluff called and has permanently frightened its victims into playing for safety. Some pictures, of course, are not rigorous, but I should say most are (and I use them whenever possible myself). An obviously legitimate case is to use a graph to define an awkward function (e.g. behaving differently in successive stretches): recently I had to plough through a definition quite comparable with the 'bad' one above [i.e., in a preceding example not quoted here], where a graph would have told the story in a matter of seconds. *This* sort of pictoriality does not differ in status from a convention like 'SW corner', now fully acclimatized. But pictorial *arguments*, while not so purely conventional, can be quite legitimate." —Littlewood, J. E., *Littlewood's Miscellany*, ed. Béla Bollobás, Cambridge University Press, N.Y., 1990, p. 54.

The source of the prejudice against the use of pictures is apparently that, since rigorous arguments are expressed in the language of symbolic logic, which is a language of letters, numbers, and other printed symbols, and since it was, of course, possible to be led astray by the wrong picture, *therefore* every proof should be carried out solely in this language. But, as you know if you have ever attempted a long, difficult proof, it is also possible to be led astray, in fact to get lost, in a long argument carried out *without* pictures. (Ironically, the argument against pictures is usually made by teachers

who, when they want to explain a proof or other concept to a student, automatically start drawing diagrams and other pictures!)

As Littlewood affirms, pictures are perfectly legitimate to use even when doing formal proofs.

Another reason for the scarcity of pictures in textbooks and in mathematical papers is that, until very recently, with the growing availability of computer-generated graphics, pictures have been expensive *in printed books and papers* in comparison with words. The picture had to be drawn "professionally" (straight lines, round circles), then pasted down on the camera-ready copy prior to printing. Perhaps this expense has been a significant motivation over the years for mathematicians' scorn for the use of pictures.

Fundamental Concept 7: Alphabetical Order

You may not be inclined to consider alphabetical order a particularly important idea, but it is. For one thing, it is an order that every literate person knows, and it works systematically for strings of letters and numbers of any length.

Do not take alphabetical order for granted! In many languages, its equivalent is difficult, if not impossible, to achieve. In ideogrammatic languages such as Chinese, for example, the nearest thing to alphabetical order is a listing of written characters in terms of the number of strokes needed to make them. Thus all one-stroke characters come first, then all two-stroke characters, then all three-stroke characters, etc. But there is no systematic way to further break up each set of characters having a given number of strokes.

As you develop Environments, you will also develop a new appreciation of this natural, universal (in the Western world) ordering. For some interesting speculations about the extraordinary influence that alphabetic systems of writing may have had on Western man — in particular, on the *thinking* of Western man — you might enjoy some of Marshall McLuhan's books, e.g., *The Mechanical Bride*, *The Gutenberg Galaxy*, *Understanding Media*.

And now, with these fundamental ideas as a basis, we can get down to the business of constructing Environments.