(TOUGH OPTIONAL PART)

COMPUTER LANGUAGES

The original computer languages were simplified ways of writing down the undermost small instructions of the computer, one by one. This is still very respectable. But when such details do not matter in their particulars, we use programming systems that take care of such things for us, letting us think clearly about the instructions that really need our attention. These systems are often called "higher" languages, or just computer languages.

Surprise! There are thousands of different computer languages.

These are at least two dozen important computer languages; the experienced programmer generally knows between three and seven.

THE HIGHER LANGUAGES

A computer is, as we have seen, a device for following a plan. This plan can be expressed in any number of ways, provided that the computer is properly set up to recognize and carry out the steps of the plan. Computer languages are simply these different ways of expressing plans. And there is no single standardized way.

The different computer languages arise from the profusion of things computers can do. Computers can do so many things — pictures and music and printing and sorting, not to mention numerical applications — and more you think about it, the more different possible things you may want the computer to do.

There are many kinds of things people want done with computers, and many styles for doing them. Indeed, little astonishes the newcomer as much as the complete blankness of the computer, the fact that it really can be made to do anything whatever that its electronics will allow.

But different people have different things in mind. Since the very beginnings, many have used the computer for rapid numerical calculation. Others use the computer principally for business accounting and for storing records of business transactions. Yet others see the computer as an extremely deft motion-picture toy.

All these people are right, no one is wrong. But with these different emphases, and the natural variation of human mentality, many different styles of programming, and local rules of operation for programmers to follow, have come into being. By and by, using the computer for a given range of problems, and in a certain style, gives rise to a new programming language. A computer language does not jump out of the air. It is designed by someone to be a useful way of telling the computer what he or she wants.

Each of the higher computer languages allows you, as a rule, to program some particular range of problems, and in a particular style. In part this is because each language handles a lot of details for you automatically. Today's larger programs call in dozens or thousands of littler programs which have themselves been perfected — little programs for putting things in alphabetical order, typing a character on a terminal, moving a picture on a screen, and

thousands of other functions. These are called *subprograms* and are of various types. While you do not want to have to create each of these subprograms, you want to be able to use them. So you need a shorthand method of telling the computer to carry out these little programs, and of tying them together. And such a shorthand method is a computer language.

Beginners are startled to learn what a lot of different computing languages there are and what little agreement about their merit among experts. Indeed, laymen commonly ask "how do you say in computer language?" and this has no general answer at all — because there are so many.

Just as the blind men misconstrue the elephant, and just as different computer users see the computer differently, different computer users likewise prefer different languages, because the different languages are tied to people's different ways of seeing and areas of concern.

People get very uptight about computing languages; the subject is as touchy as religion, if not more so. If you insult a man's favorite computer language, you cease to be his friend.

Indeed, there is no more emotional issue in the computer field than that of computer languages. While physical violence rarely occurs, the levels of emotional commitment and rage to be seen when computer people discuss computer languages is truly awesome. Many hobbyists who have only learned BASIC tend to go through this stage. Since all they have seen are programs in BASIC, all they can imagine is programs in BASIC, and thus they naturally think computers can have no uses *except* those which are easily programmed in BASIC. And indeed they get indignant, just like regular computer people, to hear anyone say they might be missing something.

The most important subject for the computer beginner is not electronics or mathematics; it is a subject that did not in any way exist thirty years ago. It is the subject of computer languages.

THE MAIN COMPUTER LANGUAGES

While this is not the place to get into computer languages deeply, let's at least do a rundown of some main areas. Because there are thousands of computer languages, there are also many different ways of categorizing them. This is a simple book, and the categorization we will make here is a simple categorization. (It might startle some professionals.)

TRADITIONAL LANGUAGES

In lumping together the following as "traditional" languages, I am taking a few liberties, but anybody who minds is probably too mad to have gotten this far in the book anyway. Traditional languages require the programmer to figure out ahead of time the exact division of memory to be used for each piece of information that needs to be stored or operated upon. One way or another, the programmer sets places aside for each kind or piece of information that will be needed. (This is one of the main pitfalls of the traditional languages, as it reduces their flexibility.)

FORTRAN

Because the first use of the computers were for arithmetical and formula computations, it was natural that a computer language should be developed which simplified the programming of algebraic formulas. This language was called FORTRAN, supposedly standing for "formula translation." Because it was the first, it became standard. Once it was a milestone; now it is a millstone. People learn it first because it is standard. It was originally designed for mathematical applications; but it is in most cases far inferior for these purposes to APL (described later). But still they go on teaching it in the universities.

COBOL

Spurred particularly by the efforts of Grace Hopper at the department of defense, a language was devised for business application, called COBOL (Common Business-oreinted Language). It has certain strengths, but is very inflexible compared to the lambda languages (described later). COBOL programmers are the coolies of the computer field.

ALGOL

In Europe, mathematicians and scientists who became disturbed at the inflexibility of FORTRAN created a language capable of expressing (and thus programming) much more elaborate and subtle types of procedures. The resulting language, ALGOL, is widely used in other countries, and is standard even in this country as a way of writing down computing procedures so that other programmers can use and understand them. This is because it has no extraneous features, as does FORTRAN.

PL/I

The language PL/I (Programming Language I) was developed as an IBM product. Roughly speaking, it is a combination of FORTRAN, COBOL and ALGOL all together, preserving the complications of each and the distinct philosophy of none. Many companies with IBM computers use it, however.

BASIC

A group of determined young men at Dartmouth College, in the early 60's, created a computer system for everybody there to use, acting on the determination to make computers easy. For this they created a new programming language called BASIC, which was the simplest of all languages to learn at the outset. Since that time, BASIC has become the standard language of hobby and amateur computing, and indeed has caught on throughout the world for many other purposes.

"Basic" is not a description, it's a name. Essentially BASIC is a simplified FORTRAN. The BASIC language, then, is not (as you might think) language somehow intrinsic to computers, but a language which was created to make programming quick and easy.

The fact that BASIC is easy to use does not mean it is efficient, and there are a lot of things that simply cannot be done in BASIC. Truly complex programs can be created in BASIC only with the greatest difficulty. However, the new computers being set up for home use all come with Basic, and so its use is growing dramatically even while it limitatpons are felt ever more painfully by those concerned with creating really versatile and complex programs.

By common consent the amateur world is deeply committed to BASIC; but there is no exact standard of what BASIC is, and so there is plenty of room for improvement. One possible hope is that the best elements of LOGO (see below) could be slyly introduced to BASIC, until BASIC comes more and more to have some of the power of LOGO. (One sort of superBASIC, called GRASS, may become available soon for amateur machines.)

THE LAMBDA LANGUAGES

The second category of computer languages will be, in the opinion of the author, the important ones for tomorrow. They offer a power, and in some cases a simplicity, that has not been widely seen as yet. The Lambda languages are called that because they are based, somewhere deep down, on something called the Lambda calculus. But you don't have to know about that.

This mysterious thing, the Lambda Calculus, is simply a systematic way of tying things together; of taking the results of one operation and making then the starting point of another operation. The Lambda languages, accordingly, are extremely versatile, as the results of any operation can be used as the beginning of any new operation. Thus, they have few of the restrictions that are so common in the other languages. Space need not be exactly prearranged, as in the traditional languages.

The Lambda languages were first used in obscure research laboratories, especially those where many delightful odd people work on what is referred to as artificial intelligence (to be discussed later). The original Lambda language is called LISP, and it is so intricate and obscure to most computer people that its practitioners have come to be seen as strange eccentrics — a priesthood within the priesthood. Yet there was a reason for this strange computer language, and all of its frightening parentheses: anything which can be done in any other computer language can be done in LISP, while things can be done in LISP that cannot be done in any other compter language.

People versed in FORTRAN and COBOL were alarmed by LISP because it contained hundreds of parentheses. The parenthesis is the most common character in LISP. This annoys and offends those who don't understand it, because they naturally think anything can be programmed in FORTRAN and COBOL, which is not true.

But LISP ordinarily only runs on big machines (although a group at MIT is endeavoring to build a LISP machine small enough to be a personal computer.)

There are, however, other languages which have all the power of LISP and yet have certain other advantages. An important one of these is LOGO. Created by Papert, Feurzeig, and others, LOGO is as simple to use as BASIC, but far more powerful. It may well become available for hobbyist computer machines in the near future.

A group at MIT, doing research in LOGO as a tool for teaching programming to children, asserts that in two weeks of instruction, children who were taught LOGO could program circles around children of the same age being taught BASIC for comparison.

But LOGO has so far been a washout for political reasons.

Picture the situation if you will. Some extremely bright and visibly eccectric people, who have very little respect for computer programming as it is ordinarily done, have been saying that computer programming should be taught to very young children in a way that most computer programmers don't understand. They have asserted that this scheme will make the children better programmers than the professionals; and they have sought funds to carry on this teaching in schools where nobody knows what a computer is at all.

Such is the computer field.

Another Lambda Language which may become important is TRAC language, invented by Calvin N. Mooers, the same man who brought you the phrase "information retrieval." (Mooers may sue me if I neglect to mention that TRAC is the trade mark and service mark of Rockford Research, Inc., 140 and one half Mt. Auburn St., Cambridge, MA 02138. He does make things difficult for those who try to use it without his permission.)

TRAC Language will run on a much smaller computer; one authorized version of TRAC language runs in only 8K spaces of the main hobby computer. TRAC Language is like LISP in that is uses many parentheses. Computer people who have been turned off to LISP — and that seems to be a lot of people — see the parentheses in TRAC and say, "Forget it". People who only know BASIC often have the same reaction.

But TRAC has certain special qualifications which ideally suit it for the very small computers that are now becoming so very widespread. It does not need large amounts of memory, and it has important features for highly interactive systems. The ability to control user input, so that if a user types the letter "F" he instantly sees, say, a picture of a fish instead of the letter F — is an extremely important feature for user-level systems of the future.

The last Lambda language we will mention here is probably the most exciting. It is called SMALLTALK and was devised by Alan Kay and his associates at Xerox Palo Alto Research Center. It's written up with neat pictures in the September '77 issue of the *Scientific American*, 231-244.

This language was created around Kay's notion of a personal computer, which he calls a "Dynabook." (Apparently the term Dynabook simply means a computer that you can program with the SMALLTALK language.) But Kay and his associates have proceeded on the correct assumption that it would be possible within a few years to build a computer the size of a book that will run on batteries, have an elaborate graphics screen, and sell for $400.

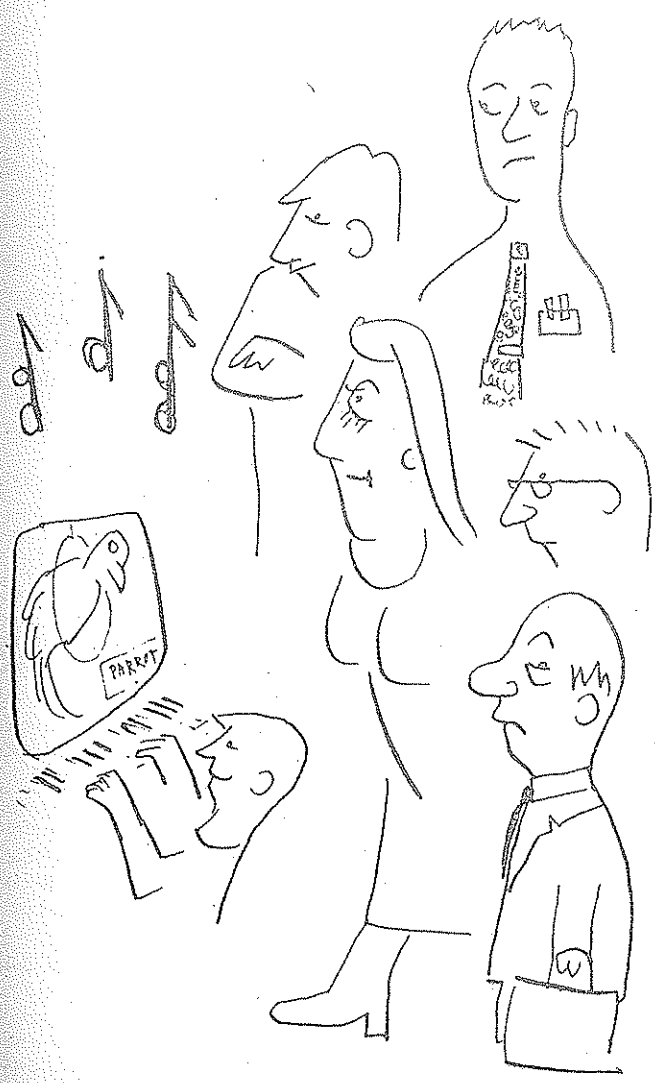
This prediction, which seemed outrageous to some people only a few years now seems firmly possible for the year 1980. Whether the management of Xerox, deeply entrenched in a paper-oriented way of thinking, will understand this development and bring it to market, remains to be seen. SMALLTALK, anyway, is a Lambda language with numerous exciting features. The parentheses are few, not the tangle of LISP. Instead, some commands of the languages consist of smiling faces and pointing hands, amongst the other symbols and phrases.

Secondly, the language is set up for the use of a finely detailed computer screen, of some half a million dots, on which the programmer may typewrite in numerous typefaces. SMALLTALK may produce dazzling animations on the screen, interacting with the user. (In another amazing form of interaction, Kay hooks SMALLTALK up to an organ keyboard coming out of loudspeakers through the computer. At the same time, the SMALLTALK program shows the notes on the screen, transcribed from his pressings of the keys.)

SMALLTALK programs are sectioned into a number of parts, called "processes," which are independent entities with a special kind of autonomy. Processes cannot interfere with each other, and thus a program may be debugged, or corrected, by sections.

But numerous copies of a process may exist. SMALLTALK programs, amazingly, are much more "like real life" than most computer programs. For instance, if you write a program to simulate traffic, you have one copy of the "car" process for each car on your highway.

IF you've done ordinary programming, you know how odd that seems to most programmers. Yet it has an intuitive simplicity. Thus SMALLTALK may turn out to be both the most powerful computing language and the ideal language for beginners. (Let's hope Xerox management gets moving on it.)



OTHER LANGUAGES, ESPECIALLY APL

There are many other languages; some have very specific ranges of purpose, others are "general purpose" but reveal a certain slant and certain special aptitudes. Foremost among these other languages is APL, or "A Programming Language," devised by Kenneth Iverson. Iverson is a fiery and upright figure, with the dignity and self-certitude of a Raymond Massey, or a religious leader.

Iverson claims that his language was always intended as a way of writing things down, especially for mathematicians and scientists, and feigns surprise that it turned out to be "a good way to drive a computer," For Iverson's notation is a powerful and elegant system of expressing mathematical meaning. Having detected, as a young mathematician, that the notations of science and mathematics are really quite chaotic and irregular, he began writing them out in a form which adhered to certain basic rules. Working all this out, he gradually put together a notational system of complete generality.

No attempt will be made to give examples here. But Iverson's language has become one of the most influential forces in the world of scientific computing. APL is a work of art, not unlike a beautiful set of surgical tools, or a set of matched gems.

Iverson's language permits the expression of mathematical concepts from across the whole of science and statistics, thousands of different ideas and functions each resolved to a crisp and concise expression in this new, common form.

The language requires learning new symbols, but a few hours of time spent with an interactive terminal and a good tutor make one able to do astonishing things.

It is interesting to note that APL has come into use almost entirely on a word-of-mouth basis. An ever-growing fraternity of scientists (and, more recently, business users) have discovered its power for a vast assemblage of purposes.

The original APL program was created within IBM, not as a planned product, but as a private project at the initiative of Iverson and his friends. But the language then caught on within IBM, becoming addictive to its users, and became a part of the IBM product line by popular demand

from the outside. It is now affecting the rest of IBM's product line, as both scientific and business users work with it more and more.

APL is now available for personal computers, especially the 8080. (Prices vary from $10 to $650 for different versions.) One version sells for as little as ten dollars; but that from Microsoft, a very respectable programming firm, it is expected to sell for about $650.

For many purposes, APL is slow and inefficient — especially for interactive graphics and music. But then again, David Steinbrook, a doughty young composer, is using it as a music machine anyway, and maybe he's onto something.

IBM sells a small computer that runs APL. This is one of IBM's best products. However, because of its cost ($5000 to $15,000), we will not consider it here as being within the range of personal computers.

OTHER NON-STANDARD LANGUAGES

There are fifty or a hundred languages that ought to be mentioned. But you can see there is no room for that here. The different languages embody different ways of thinking, different styles, different purposes. Many are variations of ALGOL. (If you want to immerse yourself in the great range of them, Jean Sammet's monumental book on programming languages is surprisingly readable.)

Suffice it to say that if you get serious about computer programming, you can make computer languages your never-ending study. Of if you go to do research at the Gazerkis Institute o f Tough Science, if there is such a place, you will probably become a fan of their language and see no other.

THE PROGRAM YOU SEE MOST OF

Many computers, big and small, come with a program that serves as a general butler of the computer system. Sitting at the system, you ask it to bring forth whatever programs you want to use, or put away data in the closet (i.e., on disk or tape).

This program butler is the operating system, or monitor. They are good things to have. They are offered for many dinky computers.

Sometimes a language processor, such as the BASIC processor, serves also as a monitor, and will store data and edit files with you. Such monitors come with most amateur versions of BASIC.

UNIVERSAL PROBLEMS OF SOFTWARE

"Software" means computer programs. Regardless of your area of interest or the language we use, some questions are inescapable.

DEBUGGING

It is natural to make mistakes while you are programming. Some people get better and better at programming, and make fewer and fewer mistakes. However, the mistakes anybody makes can be awfully big ones.

Mistakes in programming, also called bugs, are not easy to find. Surprisingly, it is impossible to tell by looking at a computer program whether it will work or not. The only way to test a program, except in a small number of mathematical cases, is to try the program and see if it works. Indeed, a program may work correctly at one time and yet have hidden bugs that may make it fail later on.

The problem gets worse as programs get bigger. Ordinarily a medium-sized program does not work the first time. Or the second. Or the tenth. But the human creating this program, struggling to find his omissions and mistakes, perfects small pieces of it at a time. And with the perfection of each piece, gets a sense of drawing closer to the overall goal.

The complications of computer programming were not

obvious at the start. Henry Tropp, who has done a research project on the history of computing, interviewed the man who discovered debugging, an English scientist. He wrote a program for a computer of the nineteen-fifties and discovered that the program did not run correctly. He found one of the errors, changed it, and discovered that the program still did not run correctly. With sinking heart it occurred to him that he would spend the rest of his working life "attempting to correct my own mistakes."

The programmer subsists on piecemeal reward, sometimes a little reward for a lot of effort, sometimes a great reward for a little effort, sometimes seemingly no reward at all. Yet this intermittency of reward, and the rare grand feeling you get when it works, seem to be enough to keep great numbers of people hard at work in programming activity. (Behavioral psychologists are quick to tell us that intermittent reward is the kind that promotes learning most effectively. But what may be more important is the good feeling when the program works.

CAN A THING BE DONE?

We get lots of ideas for things to do with computers; but not every idea is doable.

A very serious problem for the beginner is not knowing what constitutes an undoable problem, or one which is just too big. The beginner, successful with a small project, rushes right on to attempt the impossible, rushing in where experts fear to tread. (But it is just through the fearlessness of the newcomer — the kids who know no fear or modesty — that many important innovations occur.)

STRUCTURED PROGRAMMING

A new set of rules is having a great impact. "Structured programming" is a set of rules for writing programs that are easier to debug, cheaper to produce, easier to improve or fix up. Basically, structured programming means dividing the programs into blocks of certain kinds, which behave and interrelate in certain ways. The rules are just a hair too complex for this volume.

Structured programming has become sort of a religion in recent years, spread by its founder, Edsger Dijkstra of the

Netherlands; by Harlan Mills within IBM; and many others, notably Henry Ledgard, author of *Programming Proverbs*, and Brain Kernighan, author of *The Elements of Programming Style*.

Basic, and some traditional computer languages, are difficult to use according to the rules of structured programming. This is beginning to look like a strong argument for the Lambda languages, and certain others; they make it possible to get your programs running faster, and change them more readily.

THE COMPLICATIONS GO ON AND ON

Mankind is just learning what the consequences and complications of such plans of operation — detailed computer programs — really are. In the twenty years since programming began, it has been studied extensively. A great deal has been learned within the field about how programs work, and even more has been discovered that is confusing and unknowable. As the amount of known territory has increased, the amount of unknown territory has increased even faster. Programming is still an art, not a science.

Each small step forward has revealed the immensity of the unknown void beyond, just as astronomy in the 20th Century has shrunk mankind faster and faster in an unthinkably large universe.

SOFTWARE QUAGMIRES

It is all too easy to keep trying to fix programs that were really very bad in the first place, and throw good money and effort after bad. What is worse, other people have to use the quagmire that is thus created. (IBM, indeed, is notorious for their cumbersome and sprawling software — but if the customer is locked in, IBM profits from the inefficiency of that software).

Yet, like the Vietnam War, software can become justified simply on the grounds that too much has been invested in it already.

It is best to take the advice offered in *Programming Proverbs* by Henry Ledgard. "Don't be afraid to start over."

PROGRAMS AND PROGRAM PACKAGES

Since the 1950s, computer programs have been valuable objects of sale. Programs have been sold for wide varieties of purposes — usually for business, but also for science and government.

The price of individual program packages has always been, of course, what the market would bear. It is not uncommon for a language processor to cost tens of thousands of dollars to a user organization. Application programs — for specific business uses on large computers — can also cost tens of thousands of dollars. Programs may be rented instead, in which case the monthly payments can be very, very high.

This has been the world of software. The little computers, though, should have a drastic effect on the price and style of software. Right now nobody quite knows *what* effect. What is going to happen with software in the amateur market is a mystery, but we can expect the price to go down for businesses. The price of good programs for personal users may go up into the hundreds. (Thousands???)

Depending on what hardware becomes popular, programs may be sold in little wafers, or sticks like chewing gum, or cubes, all plugging into the computer somehow.

And some will be sold as they already are, on cassettes and paper tape and disks. All these are merely forms of storage for the programs, the series of commands that run the computer. But because programming is hard work, the programs may be sold as objects of value.

The principal software for the personal market will consist of canned interactive systems for an every-widening spectrum of purposes, and in a growing range of styles. The programs for your home computer will not merely be sold singly. They will also come in suites, that is, integrated collections of programs that fit together. (We may even look forward to panoramic software, linked programs for a broad spectrum of personal uses.)

We discuss below the legal matter of software protection. But whatever outcome there is to this legal issue, there will surely evolve a stable fashion by which developers of good programs can receive financial reward for them.

You can do anything with your computer that you have a program for. If you buy a canned or prepared computer system for some purpose, you do not have to learn to program. You are like the game-player and secretary mentioned earlier. Most personal computer applications are going to use software somebody else has developed.

Now, either the program exists, or it doesn't. But just because there exist programs for a given purpose does not mean they are any good.

There is usually considerable leeway in how a program can be designed. Programs that supposedly do the same thing can be as different as hats, or dogs. Many writeups on home computing in the popular presses might give the impressions that the computer will do whatever you want, in the style you expect, with someone else's program. This is almost never true. You will have to adapt to another's idea of what aspects are important, and how they are best explicated in the program. Even if a program like the one you wanted exists already, it probably is not in the style you would like. And if it does not exist, you are going to have to create it. One's personal fantasies, often so clear, tend not to be what the other guy programmed. (Great disappointments occur.) Each person's preferred style of use may be different from another's.

Unless you are the one who programs it, it will not be focused as you would have it, nor as flexible in ways you might want.

If you are going to use a pre-existing program, you have to adapt to *it*. Otherwise you must program it yourself, or adapt the pre-existing program. Thus you must learn how to program. The same goes for many of these new applications you are going to have to program yourself, or have somebody do. "If you want a thing done right, do it yourself." The way you design it is crucial. Can it be made easy to use? Making things easy for people is hard. But it can be done. You have to try hard enough, and be able to visualize. (See the later section, "Virtuality.")

Programs are being sold on paper tape and cassette. When loaded, their contents slide into the otherwise empty spaces of the machine. When you're done, you obliterate the old program and use the computer's memory for something else.

However, programs will also be sold by some manufacturers as little plug in thingies. "Thingies" is a vague term, but these plug-in programs can come in any size and shape. Some are now sold, not for computers but for calculators, in little wafers the size of sugar cubes.

These are ROMs — Read-Only Memories. These little memories, filled with their programs, behave just like the regular changeable memories of the computer when they are temporarily loaded with a program. But the ROMs are permanent.

There is no real logic distinction between one type of program and the other. But the Roms are more convenient — and people are a perhaps less likely to copy the programs that are on them.

But this is not clear. Let us consider, at this point, steps that can be taken to enforce the ownership and salability of programs.

PROGRAM PROTECTION

Most amateur computers can presently use each other's programs. By law, the owner may charge anyone who wants a copy of a program he has developed — but in fact, one hobbyist may easily give a copy to another on the sly. This is the copyright problem. There has been a great deal of program copying by hobbyists in the last couple of years. Nobody knows how much, and of course nobody — except a few troublemakers — is going around bragging that he has done this.

It is easy to make a perfect copy of much of the software for little computers.

Herein lies the temptation.

Business users pay readily for software, since it is an obvious business expense.

counterfeiting a dollar bill. In the next few years, however, it will become clear how much most people will depend on programs that are developed by others, and how very much better some of them are than others. This will affect people's thinking on the issue.

INTELLECTUAL PROPERTY

Just as background, let us review the main ways that United States law allows you to own something you come up with in your mind. A lot of people seem to think you can patent or copyright anything. This is far from the truth.

(Note that these laymen's descriptions should not be taken as a legal guide. Consult a lawyer for the exact information — and the latest. Things are changing fast.)

The law provides several methods by which people are granted certain rights to things they make up:

PATENTS

The most famous of these is the patent. The patent is expensive to get, may not be binding, and lasts for only 17 years. It protects your invention only in the narrowest sense: with reference to certain specific features which nobody can copy without your permission.

The patent was established by Congress with the stated intent of encouraging the communication of technical knowledge. For this reason it must describe fully what is being covered. In return for this description, the government gives the inventor exclusive rights to the invention — in the narrow sense covered by the wording on the actual patent — for the 17 years.

To patent something, you must search to see what is already patented, or known, that is like it. If you think yours is original you submit a patent application. Your attorney argues with patent examiners for months or years, then maybe you get it and maybe you don't.

The expense of getting a patent is generally several thousand dollars — in part based upon the attorney's judgment of your willingness to pay, in part on the complexity of your patent application. But such sums are usually out of reach for people trying to start a business on a shoestring, as most of the people interested in this matter are.

Furthermore, there is some considerable doubt as to whether patents can be obtained for computer programs. The Supreme Court has ruled lately that programs by themselves are not patentable, but that clears up less than some people think.

Until such cases have been further tried in the courts, the true status of the law will not really have been decided. That is the way American law works.

PATENTING SOFTWARE AS HARDWARE

A number of patents have been issued on fictitious machines. These machines are described with care in the patent documents but the claims are written to be actually satisfied by ordinary computers holding a certain program.

Thus, since the description in the claims exactly applies to a computer holding this program, the document could be said to have "patented the program" by patenting all uses of it. No one knows how many of these things there are or whether they are valid.

COPYRIGHT

Another very important form of intellectual property is copyright. This was originally instituted for the purpose of protecting an author's right to publish his own literary works from those publishers who might otherwise print it without paying him.

For this reason, the copyright is granted to an entire body of writing, — say a book or a play — almost automatically, and no attempt need be made by the artist at the outset to decide what, if anything, is unique about the work. That is left for the courts to decide if and when a copyright holder sues someone else as infringing on his copyright.

Copyrights are cheap — the material is supposed to be filed with the library of Congress, but in some cases is thought to be protected even without such filing. This will be rendered more precise by the new copyright law soon to go into effect, which holds that an individual need not even file his work for copyright. It is automatic. Soon there need not even be a copyright notice printed in it, a ritual observance formerly considered to be at the very core of copyright. The old copyright law held for 26 years and was renewable for another 26; the new copyright will hold good till 50 years after the author's death.

For computer programs the copyright question is this: since computer programs may generally be written on paper, and consist of symbols, are they writings? Does the 26-year protection of copyright, now granted almost automatically to authors of novels, apply in equal force and sense to the copyrighting of computer programming?

Some think so and some think not.

To those who believe in copyright, the same protection applies by a very clear extension of the existing laws and court decisions. Those who are not in favor of it point out that it was never intended to cover such things and therefore should not.

We take no position on these matters. They're tricky.

3. TRADE SECRET

There is also the possibility of simply keeping software's content secret, which can be done in various ways. This "trade secret" is the final, and perhaps the strongest, way of protecting intellectual property rights. Many programmers claim to be able to figure out how any program works. They probably can't.

HOGGING THE MACHINE FOR TEMPORARY PROFIT

Standardization benefits you far more than you realize, because it is actually an absence of impediment. Imagine if half the different states of the U.S.A. drove on the left. Or had different official spelling. Or used different sized typewriting paper, or used different television systems or had different rules about what advertising could be used on the air.

And so on. Forever.

Stress the "forever," because there is no limit to possible impediments! It's just that when things are standard, there aren't any.

The reason standardization is good is that it makes things easier, and so you can do more with less effort. (Those who argue that things should be made difficult for some moralistic reason are not welcome in this book. May their corridors never end and their forks be too heavy to lift.)

Furthermore, any standard is far better than no standard. When it comes to important things, like which side to drive on, or how to make home computers, a standard is crucial. One has come into being for home computers. It is called the S-100 standard. Designed as the interconnection system of the Altair, it is the arrangement by which all the compatible accessories manage to be compatible.

The manufacturers of the prebuilt computers, however, seem to want to monopolize their machines so that only programs they write will run.

But these people should study the example of the Altair — and of the original Philips cassette. Both are more or less in the public domain, but by setting the standard and allowing other makers in, both manufacturers assured a much greater market than they could have had if they had kept a monopoly.

The manufacturers who have broken away from the S-100 standard have set things up for themselves in the short run. Software for the PET, or the Compucolor, will be sold in permanent memory chips which are plugged into the computer. Since the programs will not work on any other computers, these manufacturers have assured a captive market for their software which other sellers are not likely to invade.

But this is self-defeating in the long run: it leads to increased separatism, diversity, and incompatibility among computers. Users who realize this may search for machines closer to a standard configuration.

Trying to keep other people's programs off the hardware you manufacture is self-defeating.

Trying to sell a machine that only takes your programs is like playing roulette at Monte Carlo; you may or may not make it big. But letting everybody sell software, and taking a cut, is like *being the house* at Monte Carlo.

(END OF TOUGH OPTIONAL PART)