Popular Electronics®

WORLD'S LARGEST SELLING ELECTRONICS MAGAZINE DECEMBER 1981/$1

THE ELECTRONIC WORLD

A User's Guide to Computer Languages

Detect Car-Battery Drain Before It's Too Late

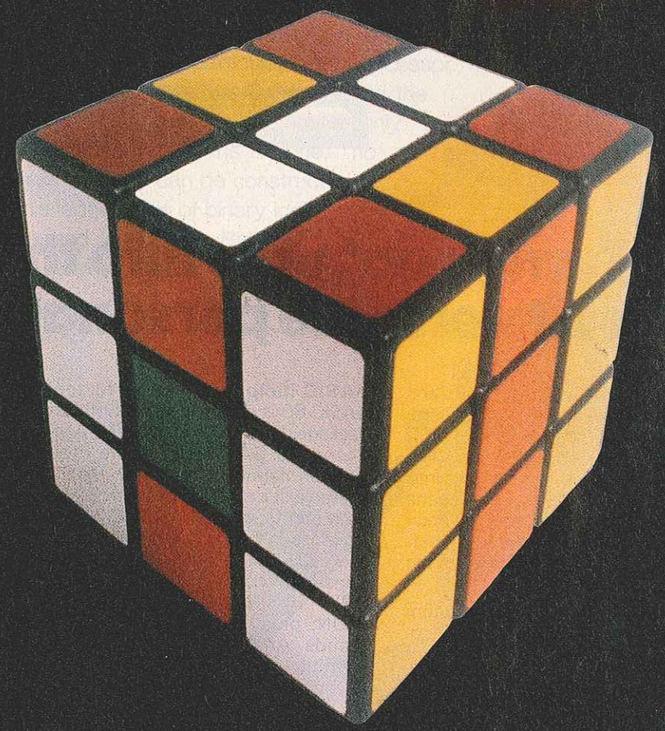
Comparing New High-Tech Audio Cassettes

0 12
14024 14278

LOOK! ...sts IBM's New Personal Computer

COMPUTER LANGUAGE CONFUSION... SORTING IT OUT

BASIC
APL
Forth
COBOL
Pascal
FLEX
Ada
PL/1
C
ALGOL
LISP
PILOT
LOGO
FORTRAN



BY STANLEY S. VEIT

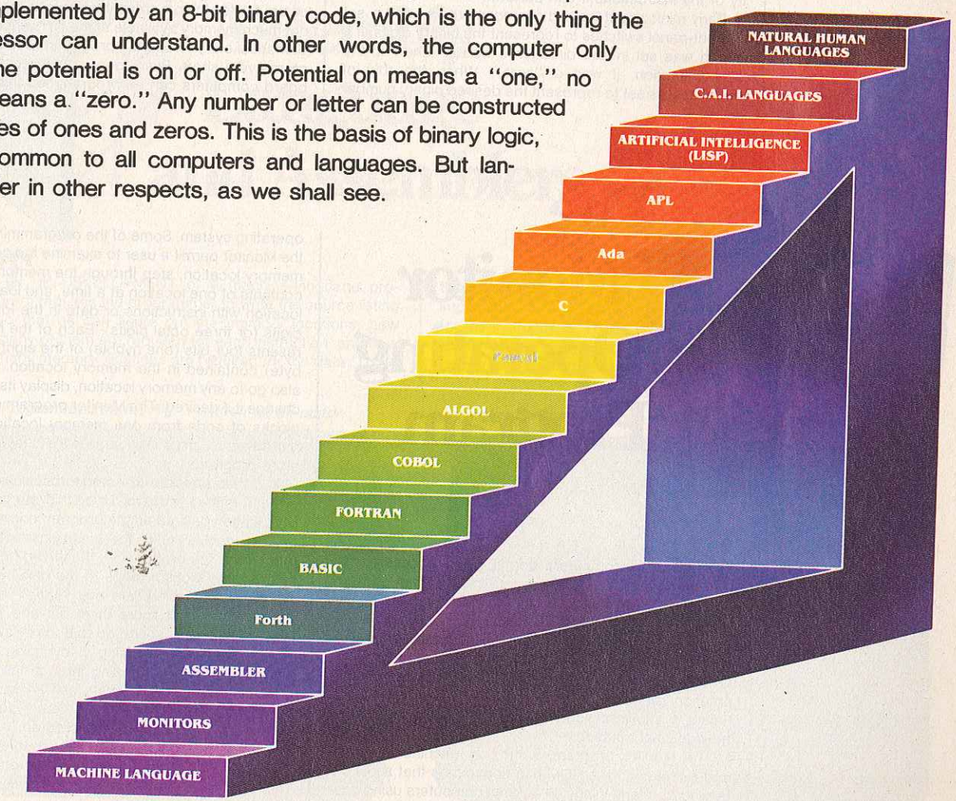
THE ELECTRONIC WORLD

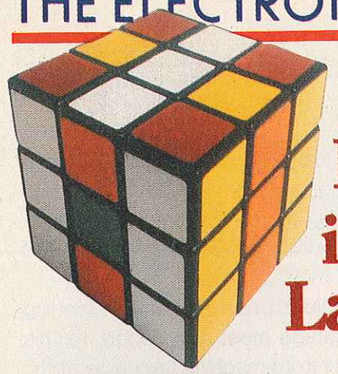
IT WOULD BE nice if people and computers shared a common language. However, since human languages are so complex, it has been necessary to invent computer languages. Most people who buy personal or small-business computers have little choice of computer languages. Someone has already made the decision for them so that, when they turn on the machine, an announcement such as "RADIO SHACK LEVEL II" or "APPLESOFT" is seen on the monitor.

In this article, we will discuss computer languages as thoroughly as possible to provide an understanding of what is available and to enable the computer user to choose the language most appropriate for his needs. The diagram below shows the various levels of computer languages from machine language at the bottom to some types at the top that actually resemble human language. We will examine each step and weigh its cost in computer resources and flexibility.

At the heart of your small computer is the microprocessor chip. It *is* the computer. Everything else, except the memory, supports the chip. It may be an 8080, Z80, 8085, 6502, 6800, etc. What these chips have in common is more important than their differences. They are all 8-bit chips with an 8-bit bi-directional data bus and 16-bit address bus. They all use instructions of one, two, or three bytes that perform similar operations. All of the microprocessors consist of an arithmetical logic unit, control circuits, and various registers. The 8-bit bus means that the word size is 8 bits (one byte) and the data bus is used to send 8 bits of data to or from external memory or input/output devices. The 16-bit address bus means that the microprocessor can directly address 65,536 (2^{16}) unique memory locations.

All of the computer instructions are unique to a particular microprocessor and are implemented by an 8-bit binary code, which is the only thing the microprocessor can understand. In other words, the computer only knows if the potential is on or off. Potential on means a "one," no potential means a "zero." Any number or letter can be constructed from a series of ones and zeros. This is the basis of binary logic, and it is common to all computers and languages. But languages differ in other respects, as we shall see.





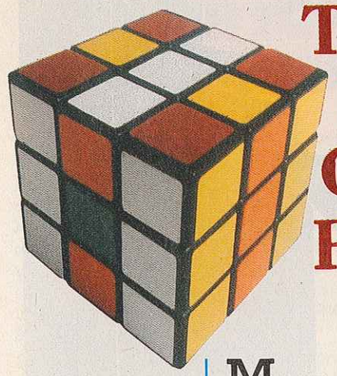
Direct Programming in Machine Language

THE METHOD of directly programming the microcomputer in binary form (or its equivalent) is called machine language. Everything else must be translated into machine language before it can be used by the microprocessor.

Programming in machine language consists of supplying the microprocessor with machine instructions, memory locations, and data in certain forms and sequences. The microprocessor cannot distinguish between instructions and data except through the form of the program. The instructions are unique to each microprocessor and are built into the chip, with the "power" of the microprocessor defined by the number and complexity of the instructions it can perform.

Early minicomputers and microcomputers used a set of front-panel switches to represent the binary digits. If a switch was set in one direction it was a "one." In the other direction, it was a "zero." When the row of switches was set to represent the desired binary number for the starting memory location, another switch was operated to cause the computer to go to the specified memory location.

The data switches were then set to represent the first instruction. To enter the instruction into the computer, a switch marked ENTER had to be operated. After each ENTER, the system automatically stepped to the next memory location. The data switches were set to the next binary number and so forth, until the program was entered one binary number at a time. There was usually a set of LEDs associated with the switches to indicate that the program had been entered correctly. When the correct binary number was stored in each successive memory location, another switch was pressed and the computer ran the program. It was a good thing there was not much memory available at the time since the process of manual entry forced the programmer to keep the programs very short. Even today, however, simple single-board computers can be programmed this way.



The Monitor Operating Program

MODERN computers permit direct access to the microprocessor through an operating program called a Monitor. The program is usually stored in Read Only Memory (ROM) and starts to run as soon as the computer power is turned on. Since it is more difficult to work with strings of binary numbers, it has become common practice to shorten the machine codes by using octal notation (base 8) or hexadecimal notation (base 16). Figure 1 shows a short program in binary, octal, and hexadecimal notations.

The Monitor program, written in machine language and burned into the ROM has commands that allow the user to do many things. In minimal computers using cassette-tape data storage, the monitor may act as the operating system. Some of the programming functions in the Monitor permit a user to examine the contents of any memory location, step through the memory showing the contents of one location at a time, and load the memory location with instructions or data in the form of two hex digits (or three octal digits). Each of the hex digits represents four bits (one nybble) of the eight-bit code (one byte) contained in the memory location. The user can also go to any memory location, display its contents, and change it if desired. The Monitor program can also move blocks of code from one memory location to another. Moreover, it can usually operate the cassette machine to store programs.

It is thus possible to enter a machine-language program from the keyboard. To do this, starting at the memory location desired as the program origin, a user enters the hex equivalent of the binary instructions and data comprising the program. As each two hex numbers are entered, the Monitor steps to the next memory location in sequence. If there are any "jump" instructions, the programmer must figure them out and tell the Monitor what memory location to go to and return. After entering the program and checking it, by going to the starting location and again stepping through the program while observing the indicator lights or CRT screen, it can be run by returning to the starting memory location and giving the GO command. If the program runs correctly, it may be saved on a cassette tape so that the next time it may be loaded from tape instead of from the keyboard. This procedure is an improvement over the front panel switches, but not a big one.

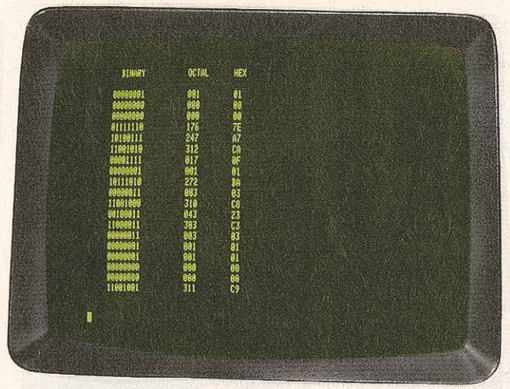
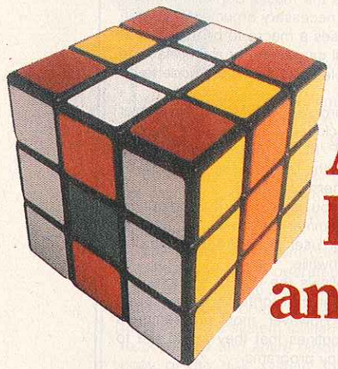


Fig. 1. A short program in three numbering systems: binary, octal and hexadecimal.

Some computers do not come with Monitor programs so it may be necessary to purchase them either as a tape or in the form of a plug-in ROM.

In cases where only BASIC is provided (the TRS-80 MOD I, for example), you might want to enter a machine-language program to do something that you can't do in BASIC. The PEEK statement is provided to examine the contents of any memory location and the POKE statement is provided to change its contents, should you wish to. To enter the machine language program you have to POKE each memory location with the correct instruction, or data byte. When the program is loaded, it can be run by using the USR statement in BASIC; or your version of BASIC may permit a CALL to a machine language program which may be in a separate library of machine language routines.

It should be noted that in Microsoft BASIC (including Radio Shack and Applesoft), the contents of memory for PEEK and POKE statements are expressed in decimal rather than in hex or octal. It is therefore necessary to convert the code in the machine-language program into decimal format before using PEEK and POKE. It is also necessary to be familiar with the memory map of the computer you are using so that POKES are not made into reserved memory areas.



Assembly Language and Assemblers

THE DEVELOPMENT of a machine-language program requires the programmer to write the source listing so that he can figure out the memory locations, how many of them are required for each instruction, and how many memory locations are required for the data. To do this using binary numbers makes the program almost impossible to read, so symbols were invented to stand for the instructions. These symbols are called mnemonics (memory aids), and each microprocessor has a unique set of them. Usually several manufacturers will

Fig. 2. Program source for Fig. 1 written on an editor.

```
;THIS SUBROUTINE SEARCHES A STRING (BASE ADDRESS IN REG PAIR HL)
;FOR THE FIRST OCCURRENCE OF A CHARACTER (IN REG D)
;THE LOCATION OF THE CHARACTER IS RETURNED IN REG PAIR BC
;THE END OF THE STRING IS MARKED WITH A ZERO BYTE (ASCII NULL)
;IF THE CHARACTER IS NOT FOUND A ZERO IS RETURNED IN REG PAIR BC

      ORG 100H
SEARCH:
      LXI  B,0          ;INITIALIZE COUNT TO ZERO
SEARCH1:
      MOV  A,M          ;PICK UP NEXT CHARACTER
      ANA  A            ;FAST TEST FOR ZERO
      JZ   STREND       ;END OF STRING
      CMP  D            ;IS THIS IT?
      INX  B            ;ADJUST SO THAT ZERO OFFSET IS LOCATION 1
      RZ                ;INX DOESN'T CHANGE FLAGS
      INX  H            ;POINT TO NEXT CHARACTER
      JMP SEARCH1       ;LOOK AGAIN
STREND:
      LXI  B,0          ;NOT FOUND SO RETURN ZERO
      RET
```

Fig. 3. Same program as in Fig. 1 in assembly language.

```
                ;THIS SUBROUTINE SEARCHES A STRING (BASE ADDRESS IN REG PAIR HL
                ;FOR THE FIRST OCCURRENCE OF A CHARACTER (IN REG D)
                ;THE LOCATION OF THE CHARACTER IS RETURNED IN REG PAIR BC
                ;THE END OF THE STRING IS MARKED WITH A ZERO BYTE (ASCII NULL)
                ;IF THE CHARACTER IS NOT FOUND A ZERO IS RETURNED IN REG PAIR BC
0100            ORG 100H
                SEARCH:
0100 010000     LXI B,0     ;INITIALIZE COUNT TO ZERO
                SEARCH1:
0103 7E         MOV A,M     ;PICK UP NEXT CHARACTER
0104 A7         ANA A       ;FAST TEST FOR ZERO
0105 CA0F01     JZ STREND   ;END OF STRING
0108 BA         CMP D       ;IS THIS IT?
0109 03         INX B       ;ADJUST FOR ZERO OFFSET AT LOCATION 1
010A C8         RZ          ;INX DOES NOT CHANGE FLAGS
010B 23         INX H       ;POINT TO NEXT CHARACTER
010C C30301     JMP SEARCH1;LOOK AGAIN
                STREND:
010F 010000     LXI B,0     ;NOT FOUND SO RETURN ZERO
0112 C9         RET
```

make the same chip, but all of them use the mnemonics of the originator. These instruction sets are published by the semiconductor manufacturers and are made available to users.

Four identical programs are shown in Figs. 2 through 5. They represent the program source as written on an editor, as well as the assembled, BASIC, and Pascal versions.

The process of using mnemonics and a particular syntax to write a program is called Assembly Language. The form of an Assembly Language line for one instruction is shown below.

```
LABEL: OPERAND   ARGUMENTS   ; COMMENTS
SEARCH:
       LXI       B,0         ;INITIALIZE
                             COUNT TO
                             ZERO
```

In the example, taken from Fig. 1, the Operand is from the Intel 8080 instruction set, where LXI means Load Immediate. The arguments B,0 indicate that the register pair BC are to be loaded with zero. Sometimes, when the value of an argument is not known, a symbolic expression is used to represent it. The label field is optional and represents the location of the memory address of the instruction (which may be unknown), and which will change from application to application. In addition, each Assembler will have a set of pseudo-ops which do not produce machine code. The pseudo-op replaces the instruction mnemonic in the operand field. For example, the pseudo-op ORG (origin) tells the assembler that the program starts at a given location, e.g., ORG 100H (where H stands for hexadecimal).

The pseudo-op END ends the program, while EQU equates the symbolic name in the program to the argument given (*i.e.:* EQU KBD 1 equates the name KBD to the value 1). At assembly time, the name KBD is placed in a Symbol Table and the value 1 is associated with it.

The program that translates the assembly language *source* code into the machine language *object* code is called the Assembler. Sometimes a simple Assembler is provided as part of a Monitor program, but most often the Assembler is a stand-alone program sold for your computer, or provided with the Operating System. CP/M and UCSD Pascal Operating Systems include Assemblers, and they also take care of the input/output connection and disk storage management required by the assembled programs.

A typical Assembler is a two-pass program. The first pass figures the length of the instruction and updates the location pointer in the microprocessor. It will also construct a Symbol Table, putting each symbolic name in alphabetic order. By the end of the first pass, all the symbols should have been given a value; if not, it is called an unresolved reference, and an error message will be printed.

On the second pass, the Assembler again reads the source statements and translates them into object code in machine language, filling in the memory reference addresses with values from the operand field of the source, or with symbolic values from the Symbol Table. If there is an error, an error message is printed. If not, the object code version is printed (See Fig. 3) and the result is saved on the disk.

The object code presents the machine code in a form that can be read by a Loader program and run on the computer.

Some Assembler programs have the capability of creating and using a collection of routines called macros, and are therefore called Macro Assemblers. These are defined as one or more valid statements that may be called up by using a single symbol within the assembly-language program.

The macro has to have been previously defined by the user, within the body of the program. The macro call is the statement that names the macro as the statement operator and gives it the necessary arguments. Such a call within a program causes a macro to be included at the point of the call. It will cause one or more machine instructions to be assembled and the binary code to be generated.

Not every Assembler program is a Macro Assembler and they are not usually included within an operating system. For example, in CP/M the Assembler is not a Macro Assembler (MAC is the Macro Assembler sold by Digital Research, the authors of CP/M).

If you want to use one, you have to buy it as an option. However, once a programmer becomes an accomplished assembly-language user, the extra expense of a Macro Assembler is worthwhile.

A Macro Assembler allows the use of a subroutine (macro) many times in a Linker program by simply calling it by an assigned name. However, most programmers have collections of subroutines that they would like to use over again within many programs.

This program permits the user to have a library of macros stored on a disk. The Linker ties them into a program at assembly time. Of course, the Linker you use must be compatible with the Assembler program in your machine.

One thing that keeps programmers from using assembly language to a greater extent is that finding and correcting errors—debugging—is like trying to map a can of worms. Don't worry. There are Debugging programs that make life easy. Using these, you can trace the execution of a program, and see the instructions executed. A program can be traced one step, or more, at a time, and can be modified to correct errors. Breakpoints can be set to stop on an error, and the contents of registers and memory can be displayed. The CP/M debugger is called DDT (Dynamic Debugging Technique), and it's a good one, but it is not the only one in use. There are others with equal capacity and features. Some are sold separately and others are part of operating-system packages. Check the package before buying.

Fig. 4. The sample program is written here in BASIC.

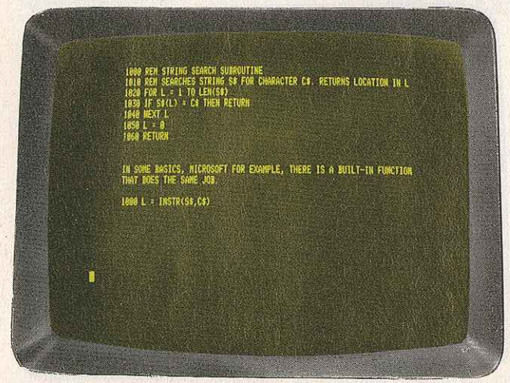
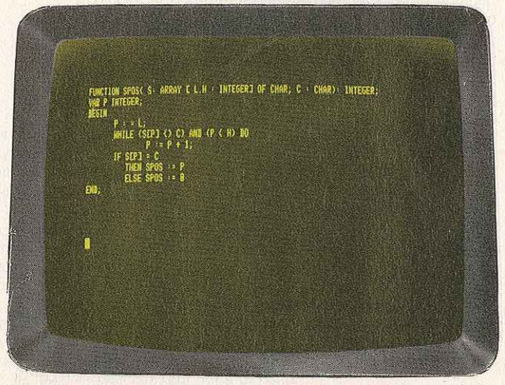
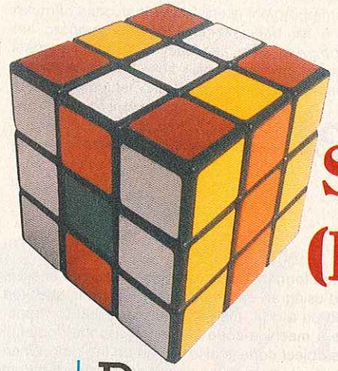


Fig. 5. Same program as before is shorter in Pascal.





Disk-Operating Systems (DOS)

DISK-OPERATING systems are a topic that must be included in any discussion of computer languages because they are really the control language that makes everything else work.

Before the use of floppy disks for microcomputers, the Monitor program could handle the simple I/O and language support, but with the use of disks everything changed.

The operating system controls the allocation of the system resources. It operates the disk system, keeping track of the storage and retrieval of programs and data. It also creates, opens and closes files. The DOS contains all of the system utilities used to format diskettes, to copy files and entire disks, and to make a back-up copy of the disk system. In addition it provides the input/output connection for all of the languages running on the computer.

A DOS is indeed a complete language, full of commands that must be used in correct syntax to direct the operation of the system. Many of the commands require a complete set of extensions and modifiers. Since the DOS is not one program, but a software system, learning it requires study and experience before a user or programmer can become skilled in its use. CP/M, for example, comes with six manuals to explain its operation. The same can be said of UNIX and almost any complete disk-operating system.

When you buy a computer with floppy disks, it must include a disk-operating system. To get the most out of the operating system you must make a commitment to it that will cost a lot of money, and take a lot of time. The choice of an operating system, more than anything else, may determine the success or failure of your computing program.

The language and the application programs that you choose all depend upon the operating system that runs on the computer. Since the disk-operating system also controls the input/output methods, it indirectly determines the type of peripherals you can use.

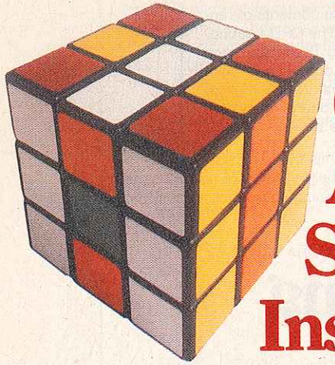
Often we read about some great new language we would like to use, or an application program that is just what we need. Upon reading a little more, we find that it runs on a different operating system, uses a different disk format, or needs more memory than the system allows. Sorry, but you are just not going to run that software. With some computers, it is possible to change your operating system. Of course, you may lose your investment in languages and application software, but it could still be worth it. If you have an 8080, Z80, 8085, or 8088 system running in an S-100 bus, you have several choices. You can use CP/M, or one of its offspring such as CDOS, SDOS, IMDOS, OS, MPP/M, and TP/M. The same chip family can also use OASIS, MV FAMOS, UCSD-P System, CIS COBOL, and many others.

The TRS-80 Mod I, II, and III have TRSDOS in several versions, to say nothing of NEWDOS, VDOS and LDOS. And they can also use CP/M. Zenith/Heath has its own system called HDOS, and you can also choose CP/M or UCSD P-system. If you use a 6800 or 6809 on the SS-50 bus there is FLEX, UNIFLEX, OS-9, UCSD P-system, and a few others. The 6502-based APPLE II offers a choice of APPLE DOS and the UCSD P-system, but you can also plug in a Z80 and run CP/M. APPLE III uses a new system called SOS. The OSI computers have several versions of OS-65 and the UCSD P-system. Challenger III models, which have several microprocessors, can also run CP/M on the Z80. Commodore PET/CBM machines can use only the version of PET DOS they were made to use, and nothing else.

If you begin to get the idea that CP/M is almost a de facto standard for microcomputers, you are correct. That is why IBM and Xerox made sure their new micros could use it. CP/M offers the widest choice of languages and application software; and it will remain that way for some time because software authors write for the market where there are the most customers. The UCSD Pascal system (now called the UCSD P-system) is available as an alternate operating system on most computer systems—underscoring Pascal's rising popularity.

With the advent of 16-bit chips and multi-user systems, new DOS's are being introduced. We do not know what will be the "CP/M" of the future, but there are already indications that today's CP/M software will be able to run under the new improvements.

The UNIX operating system from Bell Labs has many advantages—among them, excellent language and application support. (Cromemco's CROMIX is based on it.) It is predicted that UNIX will become the major operating system for minicomputers and microcomputers. However it runs only on 16-bit machines with memory in excess of 64K, and is intended primarily for multi-user systems. There are some "UNIX lookalikes" that may run on 8-bit machines, but they have not become widely used. In the future, UNIX-based systems may replace CP/M, although MP/M II and CPM-86 are also contenders for future dominance, and they would not make all the present software obsolete.



BASIC (Beginners All-Purpose Symbolic Instruction Code)

BASIC is the most popular of all computer languages. It is also the most versatile. Not really one language, BASIC is a tribe of languages having a common root. They range from Tiny BASIC, which fits into a single-chip computer, to a multi-dimensional disk BASIC for a large main-frame computer.

BASIC was invented in 1963 at Dartmouth College by Professors Kemeny and Kurtz to enable noncomputer science students to use the school computer, which was one of the first interactive time-sharing systems.

BASIC was based on an earlier language, FORTRAN (discussed next), and it can do many of the same things. It has been so effective that it was extended by Digital Equipment Corp. (among others) and soon became a standard language for minicomputers.

The first microcomputers had only enough memory for small machine-language programs. With the introduction of the Altair microcomputer and the S-100 bus by MITS, more memory was added and Altair BASIC from Microsoft became the micro standard. Today, most of the microcomputers on the market use some form of Microsoft BASIC. It has been configured for 8080, Z80, 6502, and 6800, as well as for some of the 16-bit chips of tomorrow. Versions exist in ROM, and there are some on cassettes, floppy disks, and hard disks. Other notable versions are DEC BASIC-Plus, Commercial BASIC (Basic Four), E-BASIC and CBASIC-2 (both intermediate code types). There are also North Star BASIC, Benton Harbor BASIC, Alpha BASIC, TSC BASIC, TI BASIC, and many more.

There are both interpretive and compiled versions of BASIC and some that use a combination of both methods. An interpretive language translates the program source from the high-level language syntax into machine-readable code, on a line-by-line basis. When the RUN command is given, the interpreter translates the first line into machine code and the computer executes it. Then the second line is translated and executed, then the rest of the program, one line at a time. If an error is found, the process stops and an error message is printed. If the BREAK key (or "control C" is pressed) the program stops and the line number where the halt occurred is reported.

The interpretive process must be repeated each time the program is run. However, if changes or corrections are needed, only a few words or lines need be modified. The interpreter is quite large and it must always be present in the memory.

A compiled language has several parts associated with the writing and running of a program. First, there is the original document consisting of statements written in the language format. This is called the "source" and it is composed using an editor program. After it is checked and stored on a disk, the source is compiled (or translated) into a machine-code version called the "object" code. The object code is also stored on the disk. When the program is to be run, the object code is loaded from the disk into the computer memory. Then the RUN command is given. This is much faster than the interpreted method because only the object code has to be loaded into the computer. However, if any change is needed, it is made on the source code and the program must be recompiled. The old object code is destroyed.

There is a third method of translation called Intermediate that is a combination of compiler and interpreter. Both E-BASIC and CBASIC2 (among the most popular BASIC dialects) use this method. In intermediate code languages, the source is written and complied as if it were a compiled language. However, the compiler produces an intermediate-code version rather than a machine-language version. Both the source and the intermediate code are saved on the disk. When the program is run, another program is also loaded at the same time as the intermediate code. This is called a Run-time Package and it is an interpreter that translates the intermediate code into machine code on a line-by-line basis and executes it. You may ask "why go to all that trouble?" The answer is that it makes it very easy to transport the language from one computer to another.

When we examine the reasons for the universal popularity of BASIC we find that it is mainly because BASIC is so friendly. Other computer languages are complicated. They use unfamiliar words, symbols and syntax, but BASIC speaks English. It is a very simple English, using only a few hundred words instead of the thousands of words in human language, but you can understand it from the start.

BASIC does have some defects caused by its inherent lack of structure. It is often said that, in BASIC, programmers have too much freedom to jump around. If a complex BASIC program is not well documented with comments, after a while it is even hard for the author to understand what has been done. To overcome this, Structured BASIC was developed. However, purists claim that the best cure for the defects of BASIC is not to use it.

As in FORTRAN, the common mathematical rules are generally followed, except that multiplication uses "*" as a sign instead of "x." Trigonometry, arrays, matrices and other advanced operations can be done in many versions. With the extensions added over the years, BASIC has become an almost universal language.

The letters of the alphabet are used in equations. If you run out of A through Z, some versions let you use two letters, or a letter and a number.

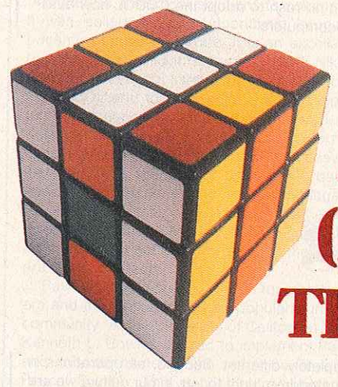
Although BASIC is simple, it must be "spoken" with precision. It will not tolerate sloppiness. There are a few ground rules that must be followed.

A BASIC program (Fig. 4) consists of statements on numbered lines which are executed one at a time. The program can be made to jump around successive statements, or to other sections of the program, and then return to execute the next line in the program. Control of the program operation is executed via a few easily learned commands, such as PRINT, RUN, GOTO, GOSUB, RETURN, READ, and INPUT. A beginner who has never operated a computer can be writing programs after one or two hours of instruction.

One of the most useful features is BASIC's ability to access machine-language routines through call instructions and PEEK and POKE commands. Some versions have the ability to chain BASIC programs together into a complete software system.

The original operating mode of BASIC was as an interactive interpreter. In the interactive mode of operation, the user types a line and then presses the RETURN key or its equivalent. This returns control to the computer which acts upon what the user has typed. The BASIC then returns control to the user, who types the next line.

Many of the more complex versions of BASIC have been written as compilers. Previously, many commercial publishers of business software were afraid to publish application programs in interpretive BASIC because the source had to be supplied to run the program. The introduction of compiled BASIC and intermediate versions removed this condition and has been an important factor in the growth of microcomputer software development for the entire field.



FORTRAN (FORmula TRANslator)

THIS was one of the first high-level languages to achieve standardization and wide acceptance. It was mainly designed for scientific and mathematical use with large computers, and is always a compiled language. Many business calculations are performed in the FORTRAN language.

FORTRAN is a statement-oriented language, using alphanumeric, mathematical symbols, and logical expressions. Only lines with labels that are referenced elsewhere in the program are numbered in ascending order. FORTRAN executes statements in order and is much more rigid in format than BASIC. The sequence of the program statements is: 1. Specification, 2. Statement Function Definition, 3. Executable Statements. In addition, it is possible to call machine-language subroutines, or FORTRAN subroutines from a previously compiled library.

Early microcomputers did not have the memory capacity to run FORTRAN programs and, therefore, made do with BASIC. With the development of larger memories and the ability to run compiled languages, it became possible to run FORTRAN on smaller computers. There are now many FORTRAN compilers used with microcomputers. But because of the wide availability of compiled BASIC and its ability to do anything that FORTRAN can do, weaknesses aside, the latter has not replaced BASIC in popularity.

COBOL (COmmon Business Oriented Language)

DESIGNED to be used in a business environment, within a short time after COBOL's introduction it seemed that every computer installation had evolved its own version. To straighten out the confusion and under the leadership of the U.S. Navy, a new standard language of business was created, known as ANS (American National Standard) COBOL. Today, many versions of COBOL have extensions beyond the standard, but this is clearly indicated in the instruction manuals.

COBOL is a statement-oriented, compiled language, very rigid in format, designed to match the flow of data in normal business transactions. This data is collected, punched into cards, and processed under COBOL programs in a batch mode on mainframe computers.

Interactive COBOL was only recently developed, matching the development of key-to-disk data input (replacing punch cards). This type of COBOL can run on minicomputers.

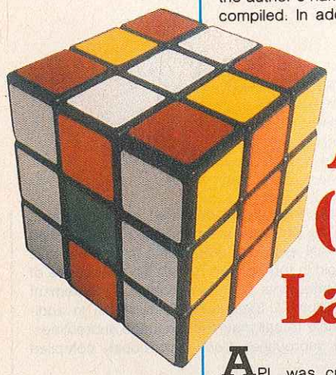
COBOL has only been available on micros for a short time and it has been used for many business applications from large computer sources. Little original programming has yet been done with microcomputer COBOL, so it is early to say if it will prove to be as popular on micros as it is on larger machines. Where software packages written in COBOL have been sold, often only the object code has been supplied, while the language has been transparent to the user.

COBOL programs are separated into divisions. The first is the Identification Division which always includes the author's name, and the date when the program was compiled. In addition, this division lists the Installation, Date Written, Security level and Remarks, if desired.

The second is the Environmental Division which specifies the hardware needed by the program, and how it relates to the files. The Configuration Section and Special Names are also part of this division. The Input/Output Section is needed if files are used. For each file there is a SELECT entry and an ASSIGN clause.

The Data Division includes the File Section and the Working Storage section. Finally, the Procedure Division completes the program. All of this must be done within rigid specifications. This creates a self-documenting program that only requires an additional explanation of why certain procedures were selected for a problem.

The trouble with using COBOL for interactive microcomputer programs is that all these requirements were originally developed for a punch-card operation in a batch mode, where the programming and operating functions are separated. In an interactive operating mode, these requirements take up space and use a lot of the computer memory. This is one of the reasons there has been no rush to adopt the COBOL now available for microcomputers.



APL (A Programming Language)

APL was created at IBM by Kenneth Iverson. It remained an internal language until it was released in the 1960s as an interactive, time-sharing language running on large mainframe computers. APL was originally a scientific language noted for its ability to create and manipulate multidimensional matrices.

A number of the IBM people who had been involved in the development of APL left the company and started time-sharing services devoted to the language. These services extended APL, added file structures, and made it into a business-oriented language. APL is especially valued by insurance companies and airlines for their complex routing and scheduling problems.

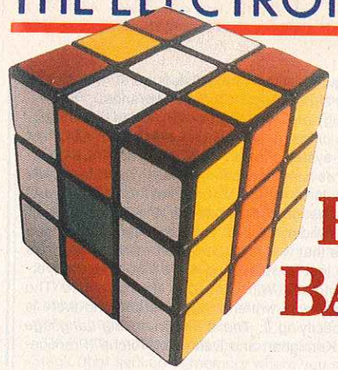
APL has been implemented on mainframes and large minis, but only on two microcomputers. Recently, however, two versions were released to run under CP/M. There are several problems with APL as a popular language. First, it has a character set which is different from any other. Some of the characters require two keystrokes with a backspace in the middle and this is not compatible with most terminals. It does not use the ASCII code (the industry standard) but instead uses Z-code, which is completely different. Second, all operations in APL are evaluated from right to left. In our culture we are used to evaluating things from left to right and this can be confusing. Third, APL uses very complex operators that permit programmers to express complicated ideas on a single line. This has lead to a very compact code that is hard to read unless it has frequent comments.

APL is always an interpretive language, using a lot of memory space and running slower than compiled languages. It gives each user a large block of memory called the Workspace as soon as he signs-on the system. In addition, it does not save individual programs (called functions), but instead saves the entire Workspace. This also uses up memory space.

APL has been restricted to large computers until recently. Thus, it does not have the wide application of COBOL, FORTRAN, or BASIC. In addition, with the development of Pascal, C, and Ada, which can also use multidimensional arrays, one of the main advantages of APL has been usurped. It should be mentioned that the new IBM small computers no longer support APL and that cassette software in APL for the Model 5100 is also no longer supported.

STRUCTURED LANGUAGES

Structured language is based on a hierarchy of operations starting with the general and proceeding to the specific. There has been a movement in computer science toward languages that fully utilize data structures and require a user to declare all the specifications for a program change. Pascal, PL/1, C, and Ada are languages of this type; and they are replacing the older languages for both business and scientific computing. Structured languages, especially Pascal, are becoming the major instructional languages in computer science departments. Accordingly, they are expected to encroach on FORTRAN, COBOL, and possibly even BASIC in time as the most important languages. Pascal is already supported on all major minicomputers, microcomputers, and a great many mainframes. PL/1, which was originally a language for very large computers, has been adapted for smaller machines. And the U.S. Defense Dept. will make Ada mandatory for all its offices and contractors by 1985.



Pascal— BASIC's Successor?

ABOUT the same time that FORTRAN was being developed in the United States, another language was designed to implement solutions to complex algorithms. It was called ALGOL (Algorithmic Language) and it became popular in Europe. It is an excellent language, but somewhat difficult to learn. In 1971, Niklaus Wirth of Zurich, Switzerland, invented Pascal as a tool for teaching ALGOL and to demonstrate the principals of structured language. In 1975, the standard Pascal was defined in *Pascal User Manual and Report* by Kathleen Jensen and Niklaus Wirth. Pascal is an easy language to learn and it is suitable for defining the data structures needed for problem solutions. The language was named for Blaise Pascal, the French mathematician who invented one of the first mechanical computing devices. So the name is not an acronym and, therefore, all letters are not capitalized.

Pascal compilers were written for mainframe computers and the language gained popularity in the computer community. At the University of California, San Diego, Dr. Kenneth L. Bowles started to implement Pascal on mini and microcomputers. The result has been the UCSD Pascal System. This is not only an implementation of Pascal, but an entire operating system that includes several Editors, a File Handling System, an Assembler, a Compiler, and a Debugger. UCSD Pascal now runs on Apple, North Star, Texas Instrument, Radio Shack TRS-80 Mod II, OSI, DEC, Western Digital Microengine, and many other personal computers. Other versions of Pascal are used on minicomputers, large mainframes, and microcomputers. Almost every manufacturer of computers supports some kind of Pascal in addition to whatever other language he uses.

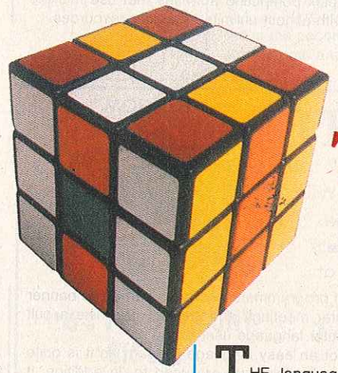
Like BASIC, Pascal "speaks" English, and uses the conventional mathematical symbols. It can do trigonometry and advanced mathematical operations, and deal with character data and strings. Pascal operates on standard data types such as integer, real, and Boolean, but gives the programmer the freedom to define new data types. The programmer can also define new functions and procedures.

Pascal is a compiled language, but it does not usually compile into machine code. Instead, it compiles into an intermediate pseudo-code called p-code. The p-code is then saved on the disk file system. At run-time, the p-code file is interpreted into the machine code of the computer.

As with all other intermediate-code languages, this method makes the language portable. To move Pascal to a new computer, all you have to do is write a new interpreter from p-code to the machine language, a far simpler task than adapting a complete language. In one computer, the Western Digital Microengine, p-code is the machine language of the microprocessor, allowing it to run without the interpretive step.

There are other versions of Pascal that do compile into machine code, called "native-code compilers." They run very fast, but are not transportable from machine to machine. Pascal-Z, produced by Ithaca Intersystems Inc. for the Z-80, is of this type.

Pascal is fast becoming the most popular language for application software, a fact that is not always apparent because only the object code and a run-time package are delivered with the application system. (This provides a measure of protection for the software publisher). Figure 5 shows the same problem used in preceding references, done in Pascal.



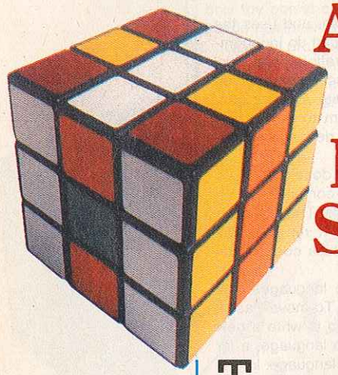
The C Language

THE language called C is a computer language designed at Bell Laboratories to operate upon the powerful OS called UNIX. It is a structured language with some resemblance to Pascal. However, Pascal uses both functions and procedures, while the C language achieves modularity only through the use of functions. It builds the entire program structure through the use of functions even to the point of having no Print or Read statements. It does input and output through use of functions. It does have "if-then-else," "while loops," global and local variables and data types, pointers and arrays. Like Pascal, it can replace single statements with compound statements to promote program flow. C is a compiled language in which programs are composed using an Editor and then are compiled into machine code versions to be run on the computer. It has no I/O structure

of its own; instead, it uses the I/O of whatever operating system it is implemented on.

A C program is a set of functions. The ability of the programmer to create his own functions according to his needs make C an unusually flexible language. There are no line numbers in C. The program starts with the name of the function, then a square bracket to start function definition. This consists of compound statements between two square brackets. Statements are nested to any depth required and are treated just like simple statements. There are libraries of standard functions and those functions previously defined by the user. All of these can be called for use in the program. There can be both Global and Local variables and there can be arguments for the functions. There are also Expressions which are used to calculate and store data. C can call machine-language routines when needed as well as any of the personal or standard function libraries.

While C originated on the UNIX OS, it has been transported to run on other operating systems such as CP/M and Unix-like systems. Many application software packages for micros have now been written in the C language and sold only in object code form. As the UNIX OS and its look alikes become more popular, the C language will become more widely used. With Pascal, the original idea of a language that was transportable between machines seems to be lost as more and more incompatable versions come into use. With C this has not happened. The entire language was written by one person and there is one book specifying it: *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritche (Prentice-Hall, 1978).



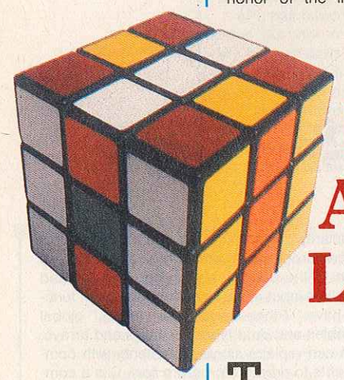
Ada—The Rising Star

THE COBOL language became a standard business language because it was required by government agencies in the 1960's. Now, the U.S. Department of Defense has decided that a new language is needed to coordinate the application needs of the Army, Navy, and Air Force. The process of development for this language was started in 1975 when suggestions were solicited from the three military services, industry, and academia. At the same time, an intensive study of existing languages was made to determine if any of them met the requirements for a universal language, and if not, to develop specifications for one. It was recommended that either Pascal, ALGOL 68, or PL/1 be used as the starting point for the new design. Pascal was the one put forward, and the new language was developed from it. Called the "Green" language, it was renamed Ada in honor of the first computer programmer, Lady Ada Augusta Byron, Countess Lovelace, daughter of the poet Lord Byron. Since then, Ada has been undergoing extensive tests and compilers have been written for several mainframe computers.

Dr. Kenneth Bowles, the leader of the project that developed UCSD Pascal, has left the University of California and is working on implementing Ada on some of the advanced microcomputers using Western Digital microprocessors and the Motorola 68000 chip. Other versions will follow. This kind of support, and the fact that Ada will be mandatory for the Defense Dept. and all its contractors by 1985, portends a very important role for the new language.

Ada looks like Pascal. It has a declarative part and a statement part. It is a strongly typed language because all identifiers must be declared and their attributes specified. The two most important control structures are the conditional statements (which select alternative actions) and loop statements (which specify repetition of an action). Ada uses many types of functions and subprograms called Procedures. In addition, Ada also has two kinds of modules called Packages and Tasks.

Packages are used to define logically related collections of resources for use in computation. Tasks are separate jobs that are done at the same time in either a time-sharing environment, or a distributed processing system. The collective term for this is 'Multitasking' and ADA has been designed to set up and run such jobs as part of a program. Similar things can be done by Pascal, C, or PL/1, but Ada is the first language created to manage the complex computing activities that use multiple processors with almost unlimited memory resources.



Forth—An Independent Language

THIS is another language created by one man, Charles H. More, and it was first used to control the telescope at Kitts Peak Observatory. It has since developed a following of programmers who have carried its banner at all computer meetings and shows, more like a cult than a computer language user group.

Forth is not an easy language to learn as it is quite dissimilar from anything we are used to. In addition, it does its calculations in Reverse Polish Notation (RPN). It is sometimes called the unfinished language because the programmer has almost unlimited freedom to create new Words. *Everything* in Forth is a Word (which is another term for a function). It is not much good as a number cruncher, but it can link to subroutines in other languages for the heavy math. It greatly reduces the cost and work of subroutines. The programmer keeps on defining new words by using old ones, and before you notice, the job is done. You do not have to do much original work to write a new program because when the system comes up, so does all the work you have ever done before, just as if it had always been part of the language! Now you see why programmers take the trouble to learn this strange language.

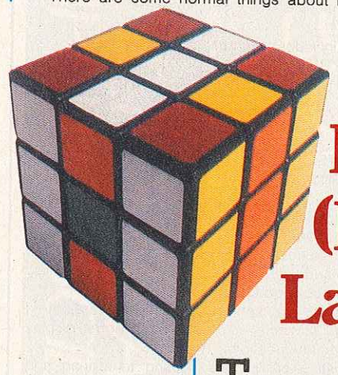
When you look at a Forth program, it is confusing because everything seems to run together, but after a while it begins to make sense.

In Forth, most operations communicate by using the Stack (that section of memory where you store numbers in last-in-first-out order). While all languages use the stack, its operation is usually incorporated into the language itself. Not so in Forth, where the programmer controls the stack directly.

There are some normal things about Forth. It is a structured language with no GOTOS or labels for statements. It is an interpretive language that is later compiled into machine-readable code, and therefore needs very little space in the computer memory. A full Forth can fit into a 16K machine and have room for 8K of programs. In addition, it is low cost. The Forth Interest Group has made versions available for almost every microprocessor. Even the commercial versions are cheap and offer a lot of features for the money.

One other characteristic of Forth is that it is a Threaded Language. This means that programs are constructed from a few subroutines which are connected together by a series of subroutine calls to perform a larger task. This entity is then called by a larger routine and connected to others to form a still larger entity. Threadedness is not restricted to Forth, but while other languages can use it, all Forth versions do use it.

When you go to a computer show and you see the cult members who have found the One True Language and wear the funny buttons, don't laugh. Perhaps they have found it and the rest of us are just lazy.

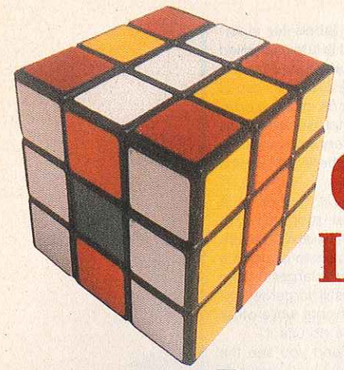


LISP (LISt Processing Language)

THE language of artificial intelligence research, LISP was based upon John McCarthy's work on nonnumeric computer languages published in 1960. The language LISP was implemented at MIT and is described in the LISP 1.5 Programmers Manual. It has since been configured on many mainframe computers, minicomputers and microcomputers,

LISP is a nonmathematical language composed of words, like any language. In LISP there are two kinds of words: atoms and lists. Atoms are the basic entities of LISP. Basically, any combination of the characters of the alphabet, A,B,C. . . . X,Y,Z with any of the ten digits 0,1. . . 9 is an atom, as long as it starts with a letter. A list is the second type of word in LISP, and it is built up from atoms and other lists. A list consists of a left parenthesis followed by any number of atoms and lists, terminated with a right parenthesis. The language has functions, variables, and arithmetic operators, but it looks strange to BASIC programmers because all the arithmetic operations are in Reverse Polish Notation (RPN). A LISP sentence looks like a list, but it carries meaning and it is actually an elementary program. All LISP functions have a single value and a program consisting of functions applied to arguments. The LISP language has many built-in functions, and the programmer can create functions at will.

The printout of a LISP program looks unusual, but if the LISP includes a "Prettyprint program," which formats a program by indenting subsections, the listing will look much more conventional.



PILOT (Programmed Inquiry, Learning Or Teaching)

PILOT was the first computer language dedicated to computer-aided instruction, and was developed at the University of California, San Francisco. It has been implemented on many computers ranging from very large mainframes to the simplest micros. This interactive language enables a person without prior computer experience to develop and test dialogue programs for teaching, since its structure and syntax are easy to explain to a student.

Using PILOT, the teacher can present the student with a reading passage, give him time to study it and then ask him multiple-choice questions based upon the passage. The program can include computer responses keyed to the answer the student has given. It can also scan his response and give him advice or comment based upon that response. It can introduce a mathematical problem and offer the solution on a step-by-step basis or give the student an opportunity to discover as many of the steps as he can, with hints from the computer.

PILOT instructions are divided into six categories:

1. Core Instructions. These basic functions are single-letter instructions and are standard for all of the versions of PILOT. Thus the programs are portable from machine to machine. The instructions are:

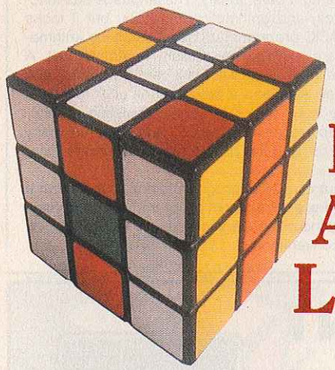
T: TYPE (includes Y: and N:)
A: ACCEPT
M: MATCH
J: JUMP
U: USE
E: END
C:COMPUTE
R: REMARK

There are also multiword instructions called "keywords" that have been added to PILOT for special applications and are not included in all versions.

2. Cursor and video instructions to determine where the text will appear on the screen.

3. Instructions that set various kinds of parameters related to the computer such as output ports, display speed, or memory locations.

4. File system instructions relating to storing and retrieving programs and data.



LOGO– A Learning Language

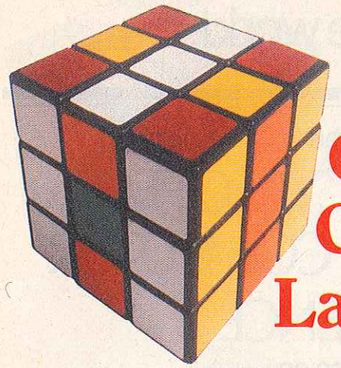
THE LOGO language represents a completely different path to learning than does PILOT. Its inventor, Seymour Papert, believes that CAI techniques, like PILOT, only transfer the old methods of teaching to the computer without using the unique capabilities of this new tool to combine text, form, color, and sound into a new learning system. For the last ten years at MIT, he and his colleagues have been working to perfect the techniques used in LOGO. The result is a language in which a five-year-old can quickly learn to write a program. Yet it is sophisticated enough for higher instruction.

In the child's version, LOGO uses basic modes called sprites and turtles. The sprites are forms that the child creates that move around the screen at any speed the child selects. The turtle is a figure that the child can interact with, moving it over the screen, coloring it, and making it draw or erase lines.

The teacher can also program more complex functions (programs) that children can interact with through simple keyboard responses. Children learn color, direction, letters, words, and sounds through this medium and usually find it fun. It also teaches them planning, and the use of the computer which will be one of their major educational tools throughout their school years.

At this time, LOGO is available for the T.I. 990/4 computer, and there is a version for the more popular Apple II that MIT has not yet released.

(Continued on page 56)



Other Computer Languages

PL/1. This language was one fo the first structured languages and was designed by IBM to run its mainframe computers. The language suffered from its great scope and complexity. Later subsets of PL/1, such as PL/M, were designed to run on minicomputers, and they were often used as cross-compilers to develop microcomputer software on larger machines. Until very recently, no micro had enough memory and capacity to run PL/1. However, with the development of the 16-bit and 32-bit microprocessors, this language has a future for use on the large micros. Digital Research, the developer of CP/M, has a version called PL/1-80 to run under advanced versions of CP/M and MP/M.

CAI. (Computer Aided Instruction) Languages. This term designates a family of languages used with computers as a teaching tool. In the next decade, CAI will become more important as we learn to use the computer to enhance our educational system. Some of the languages designed to aid in education have already been discussed.

Report Generation Languages. RPG II is one of the most widely used languages for mainframes and large minicomputers. It is used to create report formats for the output of all kinds of application software. In the future, with multiuser and multitasking computers being designed around 16-bit and 32-bit microprocessors, RPG languages may be used by all computers.

Data Description Languages. These languages are used to create, input, select, sort, and format information stored in a general application data base. They are generally not called "languages" by the software publishers, who only refer to the complete system by name. However, the CODASYL (Conference on Data Description Languages) which was formed to set standards for data base systems uses this term to refer to the entire Data Base Management System (DBMS).

Some of the commercial DBMS systems using very complex data description languages are TOTAL, RAMIS, ADBS, and IMS. All of these run on large mainframe computers. With the development of large floppy-disk and hard-disk storage systems, data-base systems became possible using mini and microcomputers. Some of the larger DBMS were scaled down to run on minis, but most of the micro systems were written for microcomputers specifically. They all use operators and functions that are complex enough to be a complete language.

Program Generation Languages. This is a new family of software systems that constitute a set of languages. Their object is to automatically write programs in another language. They are a kind of "paint-by-numbers" software. They present the user with a set of fill-in-the-blank screens to enable the user to specify just what he wants to do. The answers to these questions constitute a psuedo program from which the system "writes" a program in BASIC, or whatever language the system is designed to use. In reality, the "system" is an English-to-BASIC translator language. PEARL and "The Last One" are typical of this type of system, but are by no means the only ones or the last ones.

Conclusion. Once you have read about computer languages and begun to understand their differences, you may find that you still can't decide which one is the best for you. There are just too many choices. For example, you may need to control a robot that has a single-board computer and only 4K of memory. You could use machine language or assembly language, or you could write in Forth and compile to machine code. However, you could also use Control BASIC or Tiny Pascal. The choice is yours, and there is no single answer.

If you are interested in business applications, home controls, or scientific research, the options are still wider. Perhaps this is why Charles More invented Forth to control his telescope; machine and assembly language took too long to use while other languages were too rigid and did not allow him adequate flexibility. Today, you do not have to invent a language to tailor a program to your needs.

For most people, the choice has been made for them. Usually, a computer comes with a language, most often a simple version of BASIC in ROM. Once you learn this BASIC, you will probably find that you can do all kinds of wonderful things with it. You will likely want to do more by adding memory and a floppy disk or two. This, in turn, opens up the world of disk software for word processing, business applications, etc. This is also when you find out that you have to buy the exact package to run with your BASIC and your DOS because there were many different types.

Do you want to try different languages on your computer? Well, it's simple. All you have to do is buy a version of a language that runs under your DOS and does not require more memory than you have. Then read the manual that comes with the package, put the diskette in your drive, and you are running PASCAL, FORTRAN, or COBOL. You also will need a good textbook—one designed for microcomputer versions of the language. The manual you get with the language package teaches you how to run the language and what special things are in that version. It does not teach you the language. Study your text, use what you learn on your computer, and it will bring back the fun you had when you first bought the machine as well as giving you opportunities to use more efficient languages for particular purposes.

For those who haven't yet bought a computer, but are thinking of it, everything we have said applies to you also. If you want to have a choice of languages later, be sure you choose a machine that has a variety of languages available.

Most often, language packages are available from the computer manufacturer. However, they are also sold by software companies that specialize in one or more languages. Another good source is the computer clubs since they may serve as a distribution channel for languages developed by universities and the government. Language user groups also distribute languages at low cost. The FIG (Forth Interest Group), for example, sells its software and books—and at reasonable prices. ◊