Why Logo?

Brian Harvey, Logo Computer Systems Inc., Boston, USA

Logo is a language for learning. That sentence, one of the slogans of the Logo movement, contains a subtle pun. The obvious meaning is that Logo is a language for learning programming; it is designed to make computer programming as easy as possible to understand. But Logo is also a language for learning in general. To put it somewhat grandly, Logo is a language for learning how to think. Its history is rooted strongly in computer-science research, especially in artificial intelligence. But it is also rooted in Jean Piaget's research into how children develop thinking skills.

In a certain sense, all programming languages are the same. That is, if you can solve a problem in one language, you can solve it in another – somehow. What makes languages different is that some types of problems are easier to solve in one language than in another. Language designers decide what kinds of problems their language should do best. They then make design choices in terms of those goals.

LOGO AS A PROGRAMMING LANGUAGE

Let's postpone for a while the broader educational issues. First, we'll consider Logo simply as a programming language. How is it similar to other languages; how is it different? Syntactic details aside, there are several substantial points of language design through which Logo can be compared to other languages.

Logo is procedural

A programming project in Logo is not written as one huge program. Instead the problem is divided into small pieces, and a separate *procedure* is written for each piece. In this respect, Logo is like most modern languages. Pascal, APL, LISP, C, and even FORTRAN permit the division of a program into independent procedures. Among the popular general-purpose languages only BASIC lacks this capability. (The sample Logo programs in this article are written in Apple Logo, a dialect written by Logo Computer Systems Inc. Other versions of Logo will be slightly different in details.)

Consider the Logo program in listing (1a). Even if you don't know anything about Logo, it's probably obvious what this pair of procedures does. Compare it to the BASIC version in listing (1b).

```
(1a) TO QUIZ
     QA [WHAT'S THE BEST MOVIE EVER?] [CASABLANCA]
     QA [HOW MUCH IS 2 + 2?] [5]
     QA [WHO WROTE "COMPULSORY MISEDUCATION"?] [PAUL
          GOODMAN]
     END

     TO QA :QUESTION :ANSWER
     TYPE :QUESTION
     TEST EQUALP :ANSWER READLIST
     IFTRUE[PRINT [YOU'RE RIGHT!]]
     IFFALSE [PRINT SENTENCE [NO, DUMMY, IT'S] :ANSWER]
     END

(1b) 10 Q$="WHAT'S THE BEST MOVIE EVER?"
     20 A$="CASABLANCA"
     30 GOSUB 1000
     40 Q$="HOW MUCH IS 2 + 2?"
     50 A$="5"
     60 GOSUB 1000
     70 Q$="WHO WROTE 'COMPULSORY MISEDUCATION'?"
     80 A$="PAUL GOODMAN"
     90 GOSUB 1000
     100 GOTO 9999
     1000 PRINT Q$;
     1010 INPUT R$
     1020 IF R$=A$ THEN GOTO 1100
     1030 PRINT "NO, DUMMY, IT'S";A$
     1040 RETURN
     1100 PRINT "YOU'RE RIGHT!"
     1110 RETURN
     9999 END
```

Listing 1 – Comparison of Logo and BASIC. Each program asks the same set of three questions and compares the user's response to the author's answer. In the BASIC version (listing 1b), the 'questioning' subroutine (lines 1000–1110) is not an independent program. In the Logo version (listing 1a), the procedure QA could stand alone, and might conceivably be used by other programs.

The GOSUB construct in BASIC is weaker than a true procedure capability in several ways. For one thing, the BASIC subroutine is not an independent program; if line 100 were omitted, the program would 'fall into' the subroutine. More important, there is no concept in BASIC of inputs to procedures, like QUESTION and ANSWER in the Logo program. Instead, extra statements must be used to assign values to the variables Q$ and A$, explicitly.

This explicit assignment is not simply an inconvenience. It means that the main part of the program has to 'know' about the inner workings of the subroutine. In the Logo version, the procedure named QUIZ knows only that the procedure QA has two inputs, a question and an answer. If QA were modified to use different names for the variables, QUIZ would still work. Similarly, although this particular example doesn't show it, Logo procedures can have an *output* that is communicated to the calling procedure. (The DEF statement in BASIC provides a limited version of procedures with outputs; the limitations are that the inputs and outputs must be numbers, and the definition must be a single line without conditional branching).

Logo is interactive

Like BASIC, but unlike Pascal, Logo lets you type in a command to be carried out *right away*. It's also quick and easy to change one line of a program. Other interactive languages are LISP and APL; other noninteractive languages are C and FORTRAN.

Whether or not a language is interactive has an effect on its efficiency. In brief, program development is generally faster with an interactive language, but already-written programs generally run faster in a language that is not interactive. The difference has to do with the mechanism by which the computer 'understands' your program.

Every computer is built to understand one particular language. This machine language is different for each type of computer. Since machine-language instructions are represented as numbers, they're not easy for people to read. For example, the number 23147265 might mean 'add the number in memory location number 147 to the number in memory location 265'. Programs written in a high-level language. including Logo and the other languages mentioned here, must be translated into machine language before the computer can carry them out. This translation is done by another computer program that comes in one of two flavors: compiler or interpreter.

A Pascal compiler, for example, takes a program written in Pascal and translates (compiles) it into the machine language of whatever computer you're using. The translated program is permanently saved as machine language (probably as a file on your floppy disk). Thereafter, the machine-language program can be executed directly. The compiling process takes a long time. But once it's finished, running the compiled program is very fast because it need never be compiled again.

A Logo interpreter, on the other hand, does not create a permanent machine-language version of your program. Instead, each Logo statement is translated and executed every time the statement is supposed to be executed. The interpreter does not produce a machine-language representation of your program but simply carries out the machine-language steps itself. If a Logo statement is to be executed six times, it's translated six times. (Actually, some interpreters, including Apple Logo, save a partial translation of each procedure so that the second execution is somewhat faster than the first; this process is too complicated to explain in this article).

Interpreted languages can be interactive. Suppose you want to find the value of 2 + 2 in Pascal. First, you must use the text-editor part of your Pascal system to write a disk file containing a Pascal program. Then, you run the Pascal compiler. which will translate the program into machine language. Finally, you run the compiled program and your computer types out 4. In an interpreted language like Logo, you can simply type PRINT 2 + 2 to see the same result.

The situation in which interaction is most important is program development. If you are writing a complicated program, it probably won't work right the first time you try it. You'll have to try it, see what goes wrong, change the program, and try again. In order to see what went wrong, you'd like to be able to use interactive debugging. (You stop the program where the error happens and type in commands to examine the values of variables at that moment.) This debugging cycle may be repeated many times before the program finally works completely. Even though a compiler might make the program run faster, an interpreter is likely to make the entire debugging process faster because it's so much easier to find and fix your mistakes. It's only after the program works, and you want to use it every day without modification, that the compiled version is really faster.

The flexibility and ease of use of an interactive language is particularly valuable in an educational setting. For a student of programming, there often is no production phase – the program is of interest only as long as it doesn't work. When it does work, the student goes on to the next problem. In that sort of environment, the speed advantage of the compiler never materializes. In a business environment, on the other hand, the actual production use of a program is likely to be more important, which makes a compiler more desirable.

Some languages use mixed schemes. BASIC (normally an interpreted language) has compilers that allow the user to give up interaction for efficiency. Some LISP compilers can coexist with interpreters, so that some procedures can be compiled while others are being debugged interactively. Some versions of Pascal are compiled into an intermediate language called p-code, which is then interpreted. FORTH uses a similar system of partial compilation, but the compiler is part of the run-time environment, so single statements can be compiled and run interactively.

Logo is recursive

In a procedural language, one procedure can use another procedure as a *subprocedure* to do part of its work. A language is *recursive* if a procedure can be a subprocedure of itself.

All modern procedural languages allow recursion. Among widely used languages, only FORTRAN allows procedures but no recursion. (BASIC, as was mentioned earlier, has neither.) It may seem as though recursion isn't too important. Why should it be any different from any other use of subprocedures? It's hard to explain in a simple way why recursion is important. The idea behind recursion, though, has profound mathematical importance. By allowing a complicated problem to be described in terms of simpler versions of itself, recursion allows very large problems to be stated in a very compact form.

A well-known example of a problem best solved using recursion is the Tower of Hanoi puzzle. This puzzle has a number of different-size disks piled initially on one of three pegs, with the smallest at the top. The problem is to move the disks onto a different peg, moving one disk at a time and never moving a disk onto a smaller disk (see Fig. 1).

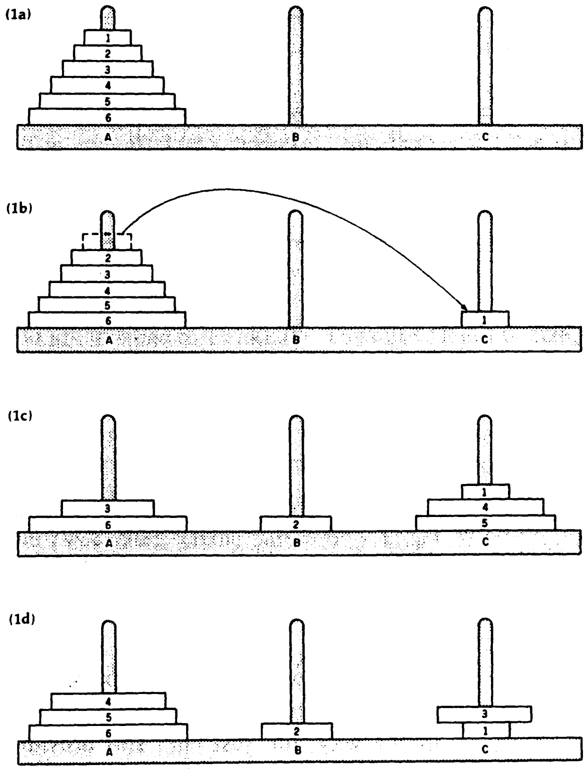

Fig. 1 – Typical moves in the Tower of Hanoi puzzle. Fig. 1a shows the initial position, Fig. 1b the first move, and Fig. 1c the position after several moves. Figure 1d shows an illegal situation with a larger disk on a smaller one.

To solve this problem, first notice that it's very easy with only two disks (see Fig. 2a). It's easy to see that we have to get disk 2 onto peg B somehow. To do that, we have to get disk 1 out of the way. Therefore, move disk 1 to peg C, disk 2 to peg B, and disk 1 to peg B.

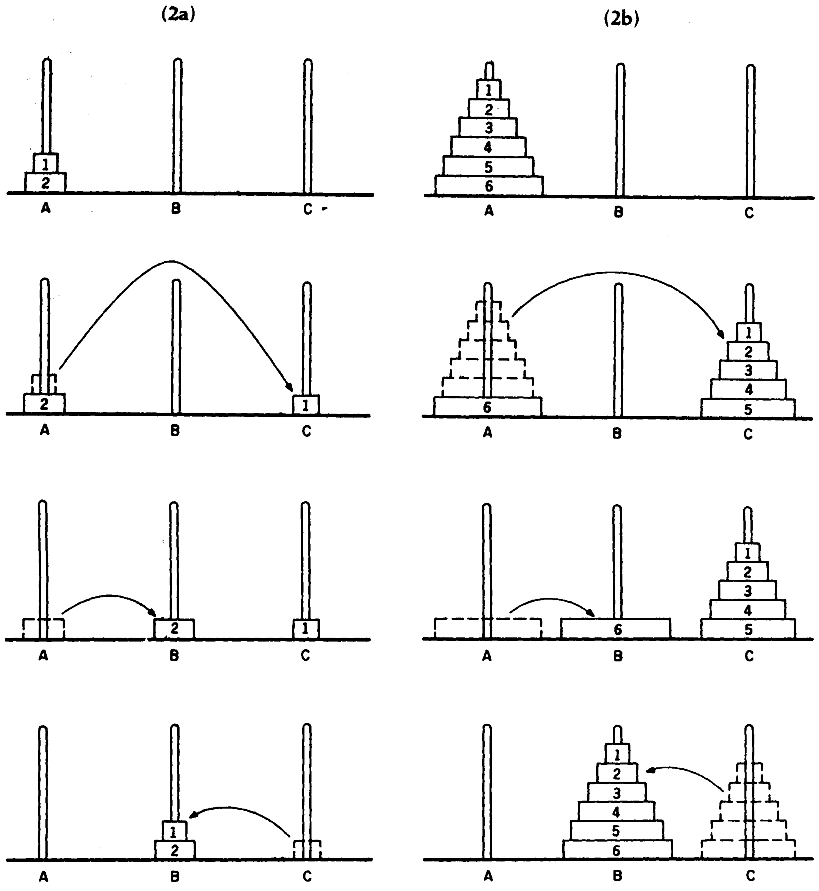

Fig. 2 – How the puzzle breaks down into simpler subproblems with similar solutions. Fig. 2a shows the simplest solution to a puzzle involving only two disks. Fig. 2b shows the situation when the same procedure is used on more disks.

Now suppose there are six disks (see Fig. 2b). Again, we have to begin by getting disk 6, the largest one, from peg A to peg B. But now there are five disks in the way, not just one. This provides us with a subproblem: move five disks from peg A to peg C. But this is exactly the Tower of Hanoi puzzle itself with five disks instead of six! The subproblem is a simpler version of the main problem. This calls for the recursive solution shown in listing 2.

```
TO HANOI :NUMBER :FROM :TO :OTHER
IF :NUMBER = 0 [STOP]
HANOI :NUMBER – 1 :FROM :OTHER :TO
PRINT (SENTENCE [MOVE DISK] :NUMBER [FROM PEG] :FROM
    [TO PEG] :TO)
HANOI :NUMBER – 1 :OTHER :TO :FROM
END
```

Listing 2 – General solution to the Tower of Hanoi puzzle in Logo. The program requires four inputs. The variable NUMBER tells the program how many disks to the puzzle; the other three inputs are the names of the pegs. The IF statement detects the trivial subproblem of moving zero disks, for which there is nothing to do.

The solution is found by dividing the problem into a series of simpler subproblems, all of which can be solved by repeating a simple series of moves. First, move all but the bottom disk to the third peg; then, move the bottom disk to the destination peg; and finally, move all but the bottom disk to the destination peg (see Fig. 2).

In working through this program, bear in mind that each use of the HANOI procedure has its own, private variables; the value of NUMBER, for example, remains constant throughout any particular use of the procedure, even though there is another use of HANOI with a different value for NUMBER in the middle.

In addition to Logo, many other languages allow recursion (these include Pascal, C, LISP, and APL). The style of Logo, however, encourages the use of recursion more than some other languages. C and Pascal allow recursion but encourage iteration. (Iteration means telling the computer to execute something repeatedly. The FOR . . . NEXT construct in BASIC is an example.) Logo is the other way around: iteration is possible, but recursion is preferred. For many purposes, neither approach is clearly right. Iteration is somewhat simpler for the situations in which it works at all; in some cases like the Tower of Hanoi puzzle, however, nothing but recursion will do.

Until recently, iteration was much more efficient than recursion, both in speed and in the use of memory. A major advance in recent implementations of Logo, including the versions available for the Apple II and the Texas Instruments TI-99/4A microcomputers, is that *tail recursion* is recognized by the interpreter and treated as if it were written as iteration. Tail recursion is the situation in which the recursive use of a procedure is the last thing done in the procedure. In general, it is only tail-recursive programs that could just as easily be done iteratively. The HANOI procedure, for example, is not tail recursive because two recursive procedure calls are in it, only one of which is at the end.

Logo has list processing

Every major programming language has some way to group several pieces of information (numbers, for example) into one large unit. In FORTRAN and BASIC, this mechanism is the array. In Pascal and C, arrays are also used, along with a more complicated grouping called a record in Pascal or a structure in C. In Logo, the main grouping mechanism is called the *list*.

Lists and arrays have two major differences. First, arrays have a fixed size, while lists can become bigger or smaller as a program executes. (There is no equivalent in Logo to BASIC's DIM statement, which is used to specify how big an array will be.) The second difference is that arrays must be uniform. That is, you can have an array of 12 numbers, or an array of strings each 23 characters long, but you can't have an array of some of both. (A Pascal record or C structure can have some of both, but only in one predeclared pattern.) Each element of a Logo list can be any Logo object: a number, a word, or even another list. Thus, the following are examples of lists:

```
[VANILLA CHOCOLATE MOCHA]
[VANILLA [MINT CHOCOLATE CHIP] [FUDGE SWIRL] ]
[BANANA 3.14159 [RED BLUE YELLOW] 2.71828]
[FLAVORS [VANILLA CHOCOLATE] SIZES [LARGE SMALL]
    OPTIONS [ [HOT FUDGE]
    [SUGAR CONE] ] ]
```

The first of these is a list of three words. The second is also a list with three members, but the first is a word and the others are lists. The third example shows that numbers can be included. The last example demonstrates that a list can contain a list that contains a list.

The last example is a special kind of list, called a property list. If this property list were associated with the name ICECREAM, the Logo statement

```
PRINT GPROP "ICECREAM "SIZE
```

would print:

LARGE SMALL

(GPROP stands for Get PROPerty.) Property lists are a convenient way to group related information. Imagine, for example, a Spacewar game program with several ships, each with a property list. The properties might be the ship's position, velocity, shape, remaining energy, and so on.

The reason that some languages restrict you to using arrays is that, being uniform and of fixed size, they are more efficient to deal with. The restrictions on arrays mean that if the computer knows where the beginning of some array is located in memory, the location of the *n*th element of the array can be calculated easily, no matter what values the elements actually have.

With a list, the size of each element is variable. Therefore, lists are stored in a more complicated way. As a result, to find the fourteenth element, you have to start with the first one, figure out where the second one is, then figure out where the third one is, etc. Since this is all done automatically by the Logo interpreter, lists aren't hard for the programmer to use, but it's somewhat slower than finding something inside an array.

Among the major languages, LISP uses lists much like those in Logo. (In fact, the data structures in Logo are based on those of LISP. LISP's name stands for LISt Processing.) APL uses a data structure that is like lists in that it is not fixed in size, but is like arrays in that it is uniform in composition. In other words, an APL vector can grow or shrink, but it has to be all numbers or all characters. Pascal and C don't have lists, but they have pointer variables that can be used along with records or structures to build the equivalent of lists. FORTRAN and BASIC don't have dynamic storage allocation – you can't make something bigger in the middle of the program – so there is no way to create lists in them.

Logo is not typed

In BASIC, if you want a variable to contain a character string, you put a dollar sign at the end of its name. If you don't use the dollar sign, the variable must contain a number, not a string. (Some versions of BASIC have a third type: a variable whose name ends with a percent sign contains a integer, or whole number.) In Pascal and C, the *type* of a variable must be given explicitly in a declaration. In FORTRAN, variables can be declared as in Pascal; if a variable isn't declared, its type depends on the first letter of its name. The letters I through N indicate integer variables.

In Logo, as in LISP and APL, variables are not typed. Any variable can take on any value. The same variable can be an integer at one point in the program and a character string (called a word in Logo) later on.

Originally, variable typing wasn't a matter of language-design philosophy. Variables were typed to make life easier for the people who wrote compilers. Since different machine-language instructions are used, for example, to add integers and to add numbers with fractional parts, it's easier to translate 'A + B' into machine language if you know ahead of time whether or not A and B are integers.

More recently, some language designers have taken the position that variable typing is a good thing, apart from implementation issues, because it disciplines the programmer to use a variable for only one purpose. In rejecting typing, the designers of Logo did not mean to encourage the haphazard use of variables for different purposes; rather, they built a procedural language in which variables are attached to a particular procedure, rather than being available to the entire program. This encourages the same discipline in a different way.

As an example in which typed variables are awkward to use, listing 3 illustrates the common problem of writing a program that reads some numbers entered by the user, performs some calculation with them, and repeats the process until the user signals that there are no more problems to do.

```
TO ADDLOOP
PRINT [TYPE TWO NUMBERS TO ADD.]
MAKE "NUMBERS READLIST
IF FIRST :NUMBERS = "DONE [STOP]
PRINT SENTENCE [THE SUM IS] (FIRST :NUMBERS) + (LAST
      :NUMBERS)
ADDLOOP
END
```

Listing 3 – Logo variables are nontyped. The variable NUMBERS contains whatever the user enters. First, it is examined as a list of words and tested to see if it contains the value DONE; next, it is used as a list of numbers and added.

This program has been written so that the user can enter the word DONE when no more numbers are left to add. In a typed language, the numbers would have to be read into a numeric-type variable, not a string type variable. Entering a non-numeric word would be an error. FORTRAN programs used to be full of instructions to the user like 'type 9999 to indicate that you're done'. Pascal programs face the same difficulty.

Logo is extensible

Every computer language has certain built-in, or primitive, operations. Most languages, for example, include arithmetic operations on numbers, and some way to print the results. Procedural languages allow the programmer to create new operations, extending the capability of the language. In that sense, most languages are extensible. But 'extensible' is used by language designers in a special sense.

An *extensible* language is one in which user-defined procedures 'look like' primitive procedures. This is partly a matter of notation and partly a matter of real power. In most languages, the primitive arithmetic operations can be applied to several different types of variables (integer and real, for example) with appropriate results for each type. In most languages, however, user-defined procedures must specify in their definition one particular type of variable to which they apply. This restriction violates the principle of extensibility.

Extensible languages are particularly valuable for teaching because a teacher can provide language extensions and teach them as if they were primitives. LISP, Logo, APL, and FORTH are extensible, with some minor restrictions in some cases Logo violates pure extensibility, for example, in that some of the primitive arithmetic operations are represented in infix form (with the operation symbol between the two operands, as in 3 + 2), while user-defined procedures can be represented only in prefix form (with the operation symbol before the operands, as in SUM 3 2). Almost all Logo primitives are used in prefix form.

As an example of the use of extensibility in Logo, most versions do *not* have primitive procedures for iterative looping, like the FOR, DO, or WHILE constructs in other languages. But it is very easy to define these procedures, if you want them, so that they look syntactically similar to the IF command that is a Logo primitive.

LOGO AS A LEARNING LANGUAGE

Among respectable languages, you may have noticed two groupings. Logo, LISP, and APL are interpreted, list-oriented, and untyped. Pascal and C are compiled, array-oriented, and typed. (All respectable languages are procedural, by definition.) These groupings reflect historical accidents, implementation convenience, and language design philosophy. For example, C and Pascal are very similar because they are both derived from an earlier language, ALGOL, that established a style followed by many newer languages.

Compilers have a much easier time with typed languages, while interpreters are just as happy with untyped ones. The list-oriented languages were all invented by people who are primarily mathematicians, rather than computer programmers.

Within each group, though, the differences tend to reflect the particular use each designer had in mind. For example, C is different from Pascal largely because C was designed as a language for systems programming. In the list-oriented group, LISP was developed for use in artificial-intelligence research, and APL was developed to teach algebra and the mathematical topics, like calculus, that depend on algebra. Logo, though, was developed as a learning language, not for a specific branch of mathematics, but for problem-solving behavior. Logo is meant to appeal particularly to younger students than APL does, although Logo has also been used successfully with college physics students at MIT.

From the point of view of the 'pure' computer scientist, Logo is LISP. The developers of Logo, in fact, have been artificial-intelligence researchers for whom LISP is second nature. The differences between the two languages are all based on the specific intent to make Logo particularly useful as a learning language. Logo's special properties from this point of view will be described next.

Logo is 'tuned' for interesting applications

Probably the most famous aspect of Logo is the idea of *turtle geometry*. This approach to computer graphics has been added to other languages, such as Pascal and PILOT, but it originated with Logo.

Most approaches to computer graphics are based on Cartesian coordinates (the 'x,y' system you learned for graphing equations in high school – see Fig. 3). In this approach, each line you want to draw is specified in terms of the specific positions of the endpoints, relative to a fixed-coordinate system. Using Cartesian coordinates, it's not too hard to draw an upright square in a known position, but if the square is tilted, its coordinates must be calculated using trigonometry. The power of turtle geometry is that lines are described not in terms of absolute position in a coordinate system, but relative to the position and direction of the turtle, a conceptual animal that moves around the TV screen. In this system, you don't say where the turtle starts or ends, just how far it moves and in what direction:

```
TO SQUARE :LENGTH
REPEAT 4[FORWARD :LENGTH RIGHT 90]
END
```

For our purposes, what's important is that the use of this powerful approach makes graphics programming possible for beginners the first time they use the computer.

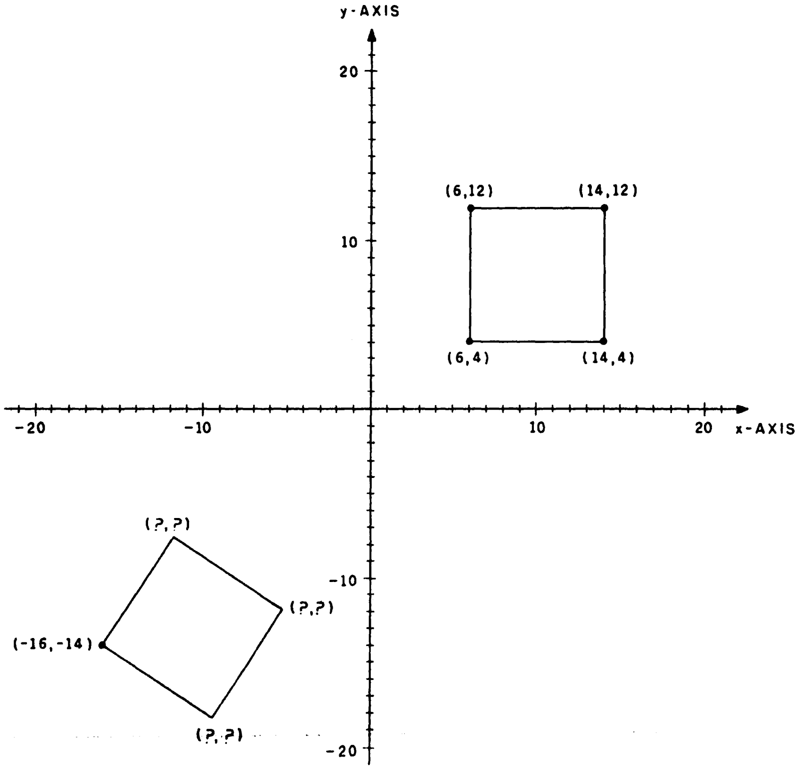

Fig. 3 – The difficulties involved in graphics using Cartesian coordinates. A square is simple to draw when its sides are parallel to the axes, but trigonometry is necessary when other orientations are used.

In the past, computer programming has appealed to only a small number of people because there has been a real lack of problems that are both interesting and easy enough for beginners, Traditional programming courses have been heavy in algebraic problems ('Write a program to solve quadratic equations.'). Therefore, they have not attracted people who don't like the traditional mathematics curriculum.

Turtle geometry is not the only special application built into Logo. Another one is language processing. Letters, words, and sentences are a natural hierarchy of Logo objects. (In most programming languages, by contrast, a sentence is not a list of words, but a string of characters. If you want to deal with the words in the sentence, you have to write a complicated program just to look for spaces in the string to divide the words.) As a simple example, listing 4 is a Logo program to translate a sentence into pig Latin. PLWORD is used as a subprocedure to translate a single word based on this rule: if the word starts with a vowel, add AY at the end. If not, move the first letter to the end and try again.

```
TO PIGLATIN :SENT
IF EMPTYP :SENT[OUTPUT[]]
OUTPUT SENTENCE (PLWORD FIRST :SENT) (PIGLATIN
    BUTFIRST :SENT)
END

TO PLWORD :WORD
IF VOWELP FIRST :WORD [OUTPUT WORD :WORD "AY]
OUTPUT PLWORD WORD BUTFIRST :WORD FIRST :WORD
END
TO VOWELP :LETTER
OUTPUT MEMBERP :LETTER [A E I O U Y]
END
```

In the program, WORD and SENTENCE are procedures for joining two objects into a larger object; FIRST and BUTFIRST separate an object into its component parts. The primitive procedure FIRST, when applied to a sentence, produces the first word of the sentence. When applied to a word, it produces the first letter. No other programming language deals so neatly with this hierarchy of objects in human language.

Logo is user-friendly

A language for learners has to be designed to deal with problems that are less important in a language meant for experienced programmers. For example, when you make a mistake, you should get a detailed, helpful error message. Languages that say things like SYNTAX ERROR or ERROR NUMBER 259 are not encouraging to a beginner. Logo has messages like:

+ DOESN'T LIKE HELLO AS INPUT

This means that you tried to add a nonnumber, the word HELLO, to something. When you see the message

I DON'T KNOW HOW TO FRIST

you have used a procedure, FRIST, that you haven't defined. The message

NOT ENOUGH INPUTS TO MAKE

means that the procedure MAKE needs two inputs, and you gave only one. If the error happens during the execution of a procedure, Logo also prints the name of the procedure and the line containing the error.

Since the beginning of time (in 1954), programming students have been getting confused about common programming statements such as X = X + 1, a frequently used assignment construct that seems to go against one's algebraic intuition. Pascal's use of := instead of the unadorned equal sign is somewhat of an improvement, and APL's ← is even better. Even so, the notation doesn't make it obvious that X ← 3 has an effect very different from X + 3 or X – 3, which look very similar. In Logo, the assignment is done this way: MAKE "X :X + 1. Although less terse than a single-character symbol for assignment, the word MAKE conjures up much more vividly the notion that something is being changed, not just used in a calculation.

There are many more ways in which Logo makes explicit things that many languages leave hidden. For example, Logo uses the colon (which Logoites call 'dots') to mean 'I want the value of this variable'; the same word without the dots names a procedure. In LISP, the notorious parentheses make it possible to distinguish procedure calls from variable references without the dots notation; most other procedural languages simply prohibit using the same word for both purposes. (That solution would be awkward in Logo because some words like WORD are not only popular variable names, but also names of primitive procedures.)

In any case, according to the design philosophy of Logo, the dots notation is a good thing, apart from its technical necessity, because it calls attention to the fact that a variable's value is different from its name; it also points out that a variable is different from a procedure. For example, in the X = X + 1 situation, the two identical-looking appearances of X have different meanings. The second represents the old *value* of X, whereas the first merely *names* the variable being given a new value. In the Logo version, these two meanings are distinguished by the notation. The first is called "X; the second is called :X.

Another example of a distinction that is explicit in Logo and not in some other languages is the division of procedures into commands and operations. An operation is a procedure that computes some value that becomes the output of the procedure. For example, the arithmetic operations are in this category. A command does not have an output, but instead has an effect: it prints something, moves the turtle, or changes the value of a variable.

The same distinction is made in Pascal, in which operations are called functions and commands are called procedures. FORTRAN calls them functions and subroutines. LISP, APL, and C, however, are less fussy. C treats all procedures as operations, but allows an operation to be used as if it were a command; the result of the operation is ignored in that case. In LISP and APL, the result of such a 'top-level' operation is printed. (In LISP, every procedure has an output and every top-level command prints something. In APL, some procedures don't have output and, therefore, don't print anything.) In Logo, using an operation without a command is considered an error; if you want something printed, you must use the PRINT command.

The use of infix arithmetic in Logo is a concession to the habits of the users. All other Logo procedures are used in prefix form, with the procedure name before the inputs. Arithmetic can also be expressed in prefix form. The two Logo expressions 3 + 2 and SUM 3 2 are equivalent.

The infix form seems more natural to people accustomed to doing arithmetic outside of the Logo environment. The prefix form, however, is better in some ways. For example, it eliminates the need for precedence of operations (i.e., where division is always done before addition, etc.). Also, it eliminates the need for parentheses to indicate grouping. In LISP, only the prefix forms are used.

Another user-friendly aspect of Logo is its facility for interactive definition of procedures. Early versions of Logo used a line-numbering technique: within each procedure, lines were numbered and could be replaced much as the lines of a BASIC program can be replaced. Current implementations of Logo use a display editor in which special control characters are used to move the cursor around the display screen to change individual characters anywhere in a procedure definition.

Logo has no threshold and no ceiling

This means that Logo is easy enough for anyone to use, but it is powerful enough for any project; it's not a 'toy' language. Logo is best known as a language for elementary school children, but it's designed for learners of any age and any level of sophistication.

How young can a Logo learner be? Well, very young children might have trouble with typewriter keys and with the spelling of procedure names. Several years ago, however, Radia Perlman at MIT built a series of special keyboards with large buttons labeled with pictures instead of words. With this special hardware, she taught the ideas of turtle geometry to 4-year-olds. This project even included the idea of procedures, with buttons called 'start remembering' and 'stop remembering' to delimit a procedure definition, and one called 'do it' to execute the procedure. Multiple procedures could be named by using buttons in different colors.

How old can a Logo learner be? Professors Harold Abelson and Andrea diSessa have been using Logo to teach physics to MIT undergraduates. They use Logo simulation programs to demonstrate not only simple Newtonian mechanics but even the general theory of relativity. Their book, *Turtle Geometry: The Computer as a Medium for Exploring Mathematics* (Cambridge, MA: MIT Press, 1981), demonstrates their approach, which has also been used successfully with high school students.

Logo has also been used for a special group of learners, those with severe handicaps. In the past, many children of normal or superior intelligence, but with impaired ability to communicate, have been diagnosed as retarded. Computers can be used with such children both as a communication prosthesis and as a field of interest in which the handicapped learner can exhibit autonomy in pursuing goals. The use of Logo in education for the handicapped is explored in Dr. E. Paul Goldenberg's book *Special Technology for Special Children* (Baltimore: University Park Press, 1979).

Other languages designed with students in mind are BASIC, Pascal and APL. (I omit PILOT, which was designed not so much for students as for teachers; in its original design, students were supposed to use computer-aided-instruction programs written in PILOT, rather than PILOT itself.) How do these languages compare with Logo in their applicability to education?

BASIC was designed as a modification of FORTRAN for beginners. By far the most important advance in BASIC was its interactive approach. This was much more of a pioneering step than it now seems because people are now accustomed to inexpensive personal computers with this feature. In the early days of BASIC, the only computers were huge, expensive ones. Although timesharing, which allowed several people to use the big computer at once, had recently been invented, many people objected to it because it used the precious time of the huge computers inefficiently. (The response of timesharing advocates was that it was more efficient in the use of human time). An interactive language was even more time-consuming than timeshared use of the old, compiled languages. For John Kemeny and his colleagues at Dartmouth to move against the general worship of efficiency was very brave.

Besides adding interaction, BASIC removed some of the most difficult parts of FORTRAN. For example, the INPUT and PRINT statements in BASIC don't require a detailed specification of the format in which information is presented, as FORTRAN does with its FORMAT statement. (As an example, FORTRAN requires the user to specify the number of digits before and after the decimal point in the printed form of a number.) Of the modern languages, only C uses primarily format-directed input and output. Unfortunately, the important ideas of procedures and local variables were also left out of BASIC.

This means that easy problems are very easy to solve in BASIC, but hard problems are close to impossible. Any large BASIC program is bound to be an unreadable maze of GOTOs. The designers of BASIC, after all, intended it as a language for beginners (i.e., Beginner's All-purpose Symbolic Instruction Code). FORTRAN was supposed to be used for more difficult programs.

The advent of personal computers has pushed BASIC into a more extended role, not because it's easy for the programmer, but because it's easy for the computer! The Logo interpreter, like the Pascal compiler, barely fits in an Apple II computer with 64K bytes of memory. BASIC interpreters are used with 8K-byte machines at a much lower cost. The result is that computer magazines are filled with long, complicated BASIC programs that are far from basic in their readability.

Pascal, on the other hand, was designed to include the most advanced ideas of computer science in recent years. Although intended as a first language, it was meant primarily for college students, particularly those interested in computer science as a career. That helps to explain why it is compiled and typed, two strong barriers to the unsophisticated student. Even the simplest Pascal program is rather complicated to write, enter into the computer, and run. That's why, in practice, Pascal is often taught to students who have already used BASIC and FORTRAN.

BASIC and Pascal were both designed to teach computer programming *per se*. APL was designed to teach mathematics, especially at the high school level. Its inventor, Kenneth Iverson, used it for several years as a blackboard language without any intention of actually implementing it on a computer. That helps explain his willingness to use special symbols not then found on any actual computer printer. Anything can be drawn on the blackboard!

In its intended use, APL is very powerful. Many computations that require iterative loops and auxiliary variables in other languages can be done in one step in APL. Most people see this power mainly as a matter of terseness; APL is famous (or notorious) for its one-line programs. The real virtue of APL's approach is that it allows the student's attention to be focused on the mathematics of a problem, rather than on the needs of the computer. APL was designed to be used not in a special programming course or a special unit stuck into another math course, but casually throughout an algebra course, just as you'd use a calculator.

Logo's goal is different from all these. It isn't supposed to be an easy introduction to something else, it's not specifically for computer-science majors, and it isn't a tool for teaching the same math curriculum people are already teaching. Instead, it's a door into the territory of the computer as an object for intellectual exploration. To return to the theme stated at the beginning of this article, Logo is for learning learning.

WHY LOGO?

In his book *Mindstorms: Children, Computers, & Powerful Ideas* (New York: Basic Books, 1980), Seymour Papert says, 'It is not true to say that the image of a child's relationship with a computer I shall develop here goes far beyond what is common in today's schools. My image does not go beyond: It goes in the opposite direction'. Logo isn't just a programming language; it's also a philosophy of education. Papert's book is the best explanation of that philosophy, but what follows is a briefer summary.

A child learns partly by picking up specific facts and skills. Much of existing formal education is about facts and skills: reading, spelling, and the multiplication table. But a more profound kind of learning is the skill of learning itself, which involves the building of mental models of the world, of oneself, and of the learning process. These models are developed through intellectual exploration. That exploration may begin in a weak, haphazard way, but a good learner develops strategies for purposeful exploration. The more one learns, the better the model of learning, and the more able one becomes as a learner.

In this process of growth, it doesn't really matter what particular aspect of the world you explore. In the introduction to *Mindstorms,* Papert mentions that at age 2 he fell in love with automobile gearboxes. When I was in junior high school, I fell in love with hypnotism. The point about using computers in education is not that everyone must know something about computers, but simply that for many people, computer programming can be the arena for this general process of learning to learn. Because the computer is such a general-purpose machine, it can appeal to many different interests. It can draw pictures, make music, write stories, or move robots.

'I want a job as a computer programmer. Why should I learn Logo, and not something useful like COBOL?' This is a common question. There are two possible answers to it. The first is that Logo, as explained earlier, is designed to make explicit many of the fundamental ideas of computer programming. Someone who learns Logo is likely to have a very clear idea of the nature of variables, procedures, and most other programming constructs. So Logo may be a better basis even for learning COBOL than simply starting with COBOL itself. But the second answer is that Logo's purpose isn't to train computer programmers. Logo isn't meant to replace all other programming languages.

Logo is generally associated with children because most people have a model of the learning process in which children learn and adults don't. This model is unfortunate. Logo can be useful to people of any age, but it will be most useful to you if you approach it in a playful, exploratory way.

It's important to distinguish between the Logo language and any particular implementation of Logo. Some things can't be done in the Apple and Texas Instruments versions of Logo simply because the machines aren't big or fast enough or because the implementation doesn't include some capabilities. For example, no microcomputer version of Logo has a good way of storing data on disk, although all versions can store procedures on disk.

The Logo interpreter barely fits in a 64k-byte Apple II, and the implementation favors the features needed for education, not those needed for practical data processing. But in principle, Logo is a good language in which to develop any application because of its interactive debugging and its procedural style.

Do you want to write a video-game program? It'll probably run too slowly in Apple Logo, unless it's a simple one. But it might be worthwhile to develop it in Logo, playing around with different ideas for your game in an environment that permits quick, easy modification of your program, and then rewrite it later in some other language. The advantage of Logo can be described partly in purely technical terms like 'interactive'. Another way of looking at it, however, is that Logo encourages the playfulness you need to design the best possible game. If all you want to do is make an exact copy of Asteroids, the benefits of Logo are less important.

In summary: Logo is a LISP-like language, and a laboratory for loose, lifelong learning about learning.