Second Rewrite

Tinker's Misunderstanding of Logo

Brian Harvey

This is a reply to the article "Logo's Limits: Or Which Language Should We Teach?" by Bob Tinker, published in volume 6, number 1 of *Hands On!* (the newsletter of Technical Education Research Centers). In his article, Tinker recommends the use of Logo for young students, fourth through sixth graders. But he suggests that certain weaknesses in Logo make it unsuitable for use with older or more advanced students.

The strengths Tinker sees in Logo are turtle graphics and extensibility. The weaknesses he lists are "its use of lists, the assignment statement, and the absence of an index looping construct." Because of these perceived weaknesses, Tinker thinks that other languages, like BASIC or Pascal, are better suited for use with older students.

Why Respond to Tinker?

The idea that Logo is good for little children and not for older people is, unfortunately, a common one. Many school districts are setting up programs in which Logo is used in the elementary schools, and BASIC is used in the secondary schools. Therefore, the importance of this issue goes beyond Tinker's personal opinions. This reply could have been presented as a general position paper without reference to any specific version of the elementary-only view of Logo.

Nevertheless, I think it's worthwhile to consider a specific example of this position. One reason is that Tinker's article was detailed and technical in its critique of Logo. That praiseworthy level of detail may help me avoid a debate in vague generalities. Another reason, though, is that we Logoites are often in the position of saying "BASIC rots your mind" and not being believed. It will turn out that the details of Tinker's views can be explained in terms of his own experiences with BASIC.

In responding to Tinker's ideas, I don't want to suggest that Logo is the perfect, universal language for all time, without any flaws. There are certainly new ideas in computer science (like the object-oriented programming of Smalltalk) which are not yet represented in Logo, and there are gaps in the particular implementations of Logo for current microcomputers. But the things Tinker lists as Logo's weaknesses are actually among its greatest strengths! Tinker misunderstands Logo because he does not approach Logo on its own terms. Instead, just like many young people whose first programming experience has been in BASIC, Tinker writes BASIC programs in his head (or sometimes on paper, in the article) and transliterates them into Logo, line by line. Naturally, the

results are just as ugly as the original BASIC programs.

Table of Contents

The bulk of Tinker's article discusses specific technical questions about the Logo language. I shall address these questions in this reply, but I must emphasize that such questions don't exist without a context. We have to consider the broad question of programming language design for education on several levels. These levels constitute the major divisions of this paper:

- A. Educational Goals First, what educational goals is computer education supposed to serve? Tinker does not address this question explicitly, but he does have ideas about it, which he sums up in the phrase "computer literacy curriculum." This general question includes the subordinate but important specific question of what should be taught at each age level.

- B. Intellectual Content Second, what is the intellectual content of computer programming in general, and of any particular programming language? For example, one of Tinker's specific objections to Logo is the way in which it assigns values to variables. Tinker's discussion is almost entirely in terms of details of the surface syntax: he objects to Logo's use of quotation marks and colons. But underlying the syntactic form is the substantial issue of how to understand the naming and binding of variables conceptually. We'll have to consider this question and others like it.

- C. Tinker's Specific Objections Finally, within the framework provided by this understanding of the goals and concepts of computer education, we can consider the technical details of the syntax of Logo and other languages. This is the largest section of the paper; it's divided into four parts dealing with four specific points raised by Tinker:
    1. Lists
    2. The "Assignment Statement"
    3. Indexing
    4. Not Enough Memory

- D. General Remarks In addition to replying to Tinker's paper, I hope to provide a brief critique of Logo and BASIC from the point of view Logo represents. This section includes a very brief comparison of the two languages as problem-solving tools, a more detailed discussion of one particular difference between them as an example, and some ideas about future developments in Logo and Logo-like languages.

A. Educational Goals: What Does "Computer Literacy" Mean?

Tinker starts his article with this statement: "There seems to be unanimous agreement that computer literacy should be an important part of every curriculum from kindergarten through college and that this should involve not only teaching about computers but teaching how to program." There is certainly widespread (though not unanimous) agreement that computers should be used in education somehow or other. But the agreement is not very deep; there are many different ideas about *how* and *why* computers should be used. Tinker's feeling of unanimity, I think, is largely based on a general unspoken agreement among educators not to ask embarrassing questions.

When people first started using the phrase "computer literacy," what they meant was an awareness of the social role of computers, and some idea of how they work. Programming, at first, was still thought to be something which only a few specialists would learn. Later, the phrase was tied to job requirements: since everyone's job will involve computers, according to this approach, we have to teach people how to operate them. Still later, people (like Tinker) extended the idea of "literacy" to mean that every child should learn the skill of computer programming. But why?

Clearly the decision about what software to use depends on the goal you're after. For example, if the point of computer programming courses in high school is to get a head start on the computer programming courses in college, then it may make sense to use Pascal, which is becoming the favorite language for introductory programming courses in the colleges. If the point is to prepare students for programming jobs directly out of high school, then perhaps FORTRAN or COBOL is what you want.

The idea behind Logo is different from these. Briefly, it is this: In the process of programming a computer to carry out projects, people can learn powerful mathematical ideas.

One example is the idea of ***debugging***. Most computer programmers consider it normal that a program may not work completely the first time you try it. A program which is not doing exactly what you wanted is not necessarily a total failure. Instead, it's considered a mostly okay piece of work, which still needs to be refined. This idea of refining one's work is very different from what usually happens in school. There, you turn in an assignment once and for all. It comes back marked up in red ink, with a C+ at the top, and the message you get is that you are a C+ sort of person. Mistakes are shameful and permanent. To a Logo programmer, a mistake is not only fixable, but may even be a source of new ideas. "That isn't what I had in mind, but it's pretty neat!" is a

common thing to hear in a Logo center.

More generally, mathematics is a *process*, not a fixed body of knowledge. Mathematicians don't spend their time doing exercises in textbooks; they invent *new* mathematics. Computer programming is a medium in which people who are not experts can create mathematics.

Another example is the idea of *modularity*. This means dividing a large problem into smaller pieces. This idea isn't new with Logo; it is one of Polya's problem-solving strategies, for instance. But in Logo, it is expressed in a very concrete form, because a Logo program is organized as a group of *procedures*. Each procedure is a small program itself, which deals with one aspect of the overall project. Procedures communicate with one another in a well-defined way, encouraging the programmer to maintain a clean division between different parts of a problem. Pascal also has a procedural structure; BASIC does not.

None of these ideas is unique to Logo. In fact, none is even unique to programming–they are important precisely because they're valuable in other contexts also. But Logo differs from other languages in that its design *emphasizes* these ideas.

The philosophy behind Logo's use of computers in education is very different from "computer literacy." The point is not to teach about computers, or to teach programming as a skill for its own sake. Instead, the goal is to *use* computer programming as a tool for learning mathematics, learning autonomy, learning the spirit of intellectual play.

Not every high school student will end up programming computers later in life. Not every student *needs* the specific skills of computer programming. But many students, and other people, can benefit from the experience of doing real mathematics by carrying out projects using the computer. Why using the computer? Because it is a tool which provides concrete, manipulable mathematical objects. The Logo turtle is an example. What is compelling about the turtle is not simply that it produces snazzy pictures, but that it does so in a mathematically rich way.

B. The Intellectual Content of Computer Programming

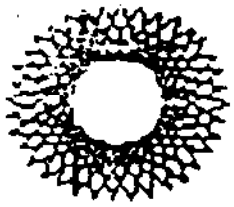
Consider this Logo procedure:

```
TO LRPOLY :SIZE :LANGLE :RANGLE
FORWARD :SIZE
LEFT :LANGLE
FORWARD :SIZE
RIGHT :RANGLE
LRPOLY :SIZE :LANGLE :RANGLE
END
```

This procedure is a slight elaboration of the POLY procedure which is often used in Logo to draw regular polygons. This version draws zigzags, like this:

LRPOLY 40 70 130

LRPOLY 70 50 194



LRPOLY 40 70 140

LRPOLY 40 70 140 (stopped partway)

These are pretty pictures. More interestingly, they can encourage mathematical exploration in several directions. One possibility is to think about *symmetry*. Why does one of these figures show sixfold symmetry and another fivefold? Can you find inputs to LRPOLY which aren't symmetrical at all?

For people unfamiliar with Logo, it is probably worth a few paragraphs to explain the structure of this short Logo program. The first line introduces the procedure, indicating that its name is LRPOLY and that it has three *inputs*, named SIZE, LANGLE, and RANGLE. An input is some piece of information which is given to a procedure when you use it. For example, the command

```
LRPOLY 40 70 130
```

gives the first input the value 40, the second input the value 70, and so on. The names

give the procedure a way to refer to its inputs. In the instructions which make up the LRPOLY procedure, the notation :SIZE means "the thing stored in the box named SIZE." These boxes are variables, but their values come from the inputs given to the procedure, rather than from explicit assignment. This is how the idea of variable is generally introduced in Logo teaching.

It's important that in the title line of the procedure, its inputs are given *names*; when the procedure is actually used, the inputs are given *values*. Inside the definition of the procedure, the colons used in front of the variable names represent the operation of *evaluation* of the variable. These fine points may seem obvious to an experienced programmer, but they're often a problem for beginners. We'll see later that Logo and BASIC take very different approaches to helping the student understand these issues.

By the way, the word TO which introduces the definition of LRPOLY represents the idea "Here's how *to*..." or "I'm going to teach you how *to*...." This metaphor of programming as teaching the computer is an important part of Logo's style.

The LRPOLY procedure is invoked by typing its name, and specifying values for its inputs. Similarly, the instructions which make up the definition of LRPOLY invoke other procedures. The line

```
LEFT :LANGLE
```

invokes a procedure named LEFT, and gives it as its input the thing stored in the box named LANGLE. (Logo knows that LEFT is a procedure, not a variable whose value we want, because the word LEFT is used without punctuation. If it were :LEFT Logo would look for a variable of that name.) The procedure LEFT happens to be a *primitive* procedure, one which is built into Logo. (It turns the turtle to the left by the number of degrees given as input.) But the way you invoke a primitive procedure is no different from the way you invoke a user-defined one.

So far, we've used only numbers as inputs to procedures. Numbers are often used in graphics programs, representing either angles or lengths of lines to draw. But we'll see later that Logo can understand other kinds of things as inputs also.

There are some subtle issues in computer science lurking in this example. One such issue is that when we invoke LRPOLY, we need to know that it requires three inputs, and what those inputs mean. But we *don't* need to know the names of the variables associated with the inputs within LRPOLY. In fact, those variables only exist while LRPOLY is active. They are created when LRPOLY is invoked, and they disappear when it finishes. We call them *local* variables. Among other reasons, this locality of variables is

better than BASIC's *global* variables because it helps ensure that one part of a program doesn't mess up another part. If **LRPOLY** is used as part of a larger group of procedures, and another procedure also uses a variable named **SIZE**, the two don't interfere.

In discussing this short procedure, I've referred briefly to the ideas of *symmetry*, *inputs*, *naming*, *variable*, *metaphor*, *evaluation*, *invocation*, and *locality*. These are examples of the intellectual content with which Logo concerns itself. Some of these ideas are specific technical parts of programming itself, like *inputs* and *invocation*. Others are more general mathematical ideas, such as *symmetry* and *locality*. Still others are ways of thinking about thinking itself, like *metaphor* and *debugging*. Later in this paper, we'll investigate some of these ideas in more detail.

What's Easy and What's Hard

Several times in his article, Tinker refers to some aspect of Logo as being powerful for sophisticated programmers, but hard for beginners. Yet he recommends Logo for young beginners, and not for more advanced students! This doesn't make sense.

One problem is that it's possible to confuse a few different meanings of the word "easy." Consider riding a bicycle. It's *easy* to learn to ride, in the sense that just about anyone can do it. You don't have to be unusually smart or well coordinated. But it's *hard* to learn, in the sense that you can't just do it right away. You have to invest a good deal of effort, and fall down a few times. But this investment is not onerous to kids learning to ride. It's fun! The effort is also educationally valuable in its own right; it teaches self-discipline and planning for long-term goals. But kids wouldn't undertake the effort if the result (being able to ride a bike) weren't worthwhile to them.

The difficulty of learning to ride a bike is inherent in the task; it's not anyone's fault. On the other hand, if the pedals were on the handlebars, bike riding would be harder, for no good reason. Part of good bike design is to avoid putting *unnecessary* difficulties in the way of the learner.

Another way things can be hard is that they can conflict with what you're already used to. For example, the Linotype machine used for typesetting (when it's not done by computer) has a special keyboard which is designed to be very fast to operate. But it's very different from the standard typewriter keyboard. It's not much harder to learn the Linotype keyboard than the typewriter keyboard, but it's almost impossible to learn both of them.

Some of the things Tinker considers hard in Logo really *are* hard, in the same way

that learning to ride a bike is hard. The whole idea of assigning values to variables, for example, is hard for any beginning programmer, in any language. But many of the specific details Tinker calls "hard for beginners" *aren't* hard for beginners at all. Instead, they're hard precisely *for experienced programmers steeped in BASIC!* It's like the analogy of the two keyboards. It is only because Tinker tries to identify Logo mechanisms with the style of programming he himself understands best that he finds Logo difficult.

The Use of Metaphor

One of the important insights behind the design of Logo is the use of metaphor in learning. We've seen the example of the word TO which is used to conjure up the metaphor of programming as teaching the computer. I'd like to introduce another metaphor, in which the computer is described as being full of little people, each of whom knows how to carry out some particular procedure.

To be specific, consider this example. Things which are underlined here are printed by the computer.

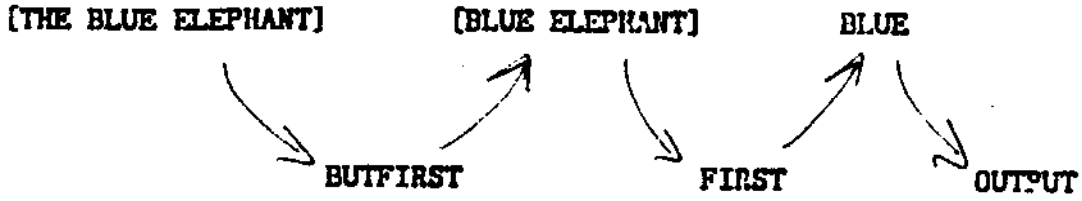
```
TO SECOND :THING
OUTPUT FIRST BUTFIRST :THING
END

?PRINT SECOND [THE BLUE ELEPHANT]
BLUE
```

The procedure SECOND outputs the second element of its input. In this example, the input to SECOND is a *list* of words, and the procedure gives us the second word in the list. This is our first example of the fact that procedures can have an *output* as well as having inputs. "Output" doesn't mean "print"; the output from a procedure is used as an input to another procedure. In the example, the computer prints the word BLUE because the output from SECOND is used as the input to the PRINT procedure.

It may be helpful to think of an instruction like this as a sort of bucket brigade, in which the little person who carries out each procedure passes something on to the next one in line. So in this case, SECOND hands some information to PRINT. The PRINT procedure, at the head of the line, doesn't pass its bucket on to anyone else, but does something with it--throws the water onto the fire. The metaphor is not perfect, though, because each procedure in the line doesn't just pass on the same information it is given. Instead, the procedures in the brigade *process* the information in some way. In this example, SECOND passes on as its output only the second element of its input.

A similar bucket brigade is at work within the SECOND procedure itself. The primitive procedure BUTFIRST gives as its output everything *but* the *first* element of its input. The primitive FIRST outputs the first element of its input. In this example, the bucket brigade works like this:



At the head of the brigade, OUTPUT makes the word BLUE be the output from the procedure we're defining, SECOND.

This bucket brigade, by the way, is precisely the idea which is called *composition of functions* in algebra classes. This important mathematical idea is, for some people, more compelling in the context of Logo programming. For one thing, the invocation of a procedure is a process which happens at a particular time, so it's natural to think about a sequence of invocations. Algebraic functions aren't taught as a computational process, but as timeless facts. Also, the domain of words and sentences is more concrete and familiar, to some people, than the more abstract domain of numbers.

C. Tinker's Specific Objections

We are now ready to look at Tinker's specific objections to Logo.

1. Lists

Tinker's first example of a difficulty in Logo is the use of lists. "The full power of lists is awesome, but also far beyond the reach of beginning programmers. Even though the Logo implementation of lists is simple and straightforward, beginning students must still grapple with the underlying concepts."

Here is the underlying concept: A list is a bunch of things. For example, the list

```
[VANILLA CHOCOLATE STRAWBERRY]
```

is a list with three elements, each of which is a word. The list

```
[COFFEE [CHOCOLATE CHIP] [ROCKY ROAD]]
```

is a list of three elements, two of which are themselves lists. Is that difficult to understand?

Here is a Logo program which uses lists like these:

```
TO PRAISE :FLAVORS
IF EMPTYP :FLAVORS [STOP]
PRINT SENTENCE [I LOVE] FIRST :FLAVORS
PRAISE BUTFIRST :FLAVORS
END
```

And here is how the program might be used. What the computer types is underlined.

```
?PRAISE [[ULTRA CHOCOLATE] [CHOCOLATE CINNAMON RAISIN] GINGER]
I LOVE ULTRA CHOCOLATE
I LOVE CHOCOLATE CINNAMON RAISIN
I LOVE GINGER
?
```

This procedure is not terribly different from the procedure SECOND we looked at above, but I don't want to explain in detail all seven of the primitive procedures used within it. Instead, let me briefly explain the overall effect of each line of PRAISE. The first says that if the input is empty, stop this procedure. (The importance of this step will become clear in a moment.) The second says to print a sentence combining the words I LOVE with the first element of the input. So, for example, the first thing that happens in the example shown is that Logo prints I LOVE ULTRA CHOCOLATE. Finally, the third instruction invokes the PRAISE procedure again, but with an input containing all but the first element of the original input. Each time PRAISE is invoked, its input is smaller by one element, until finally the input is empty. Then the first line comes into play, terminating the program.

Suppose you wanted to do what this program does in BASIC. Since BASIC doesn't have lists, you'd have to set up an array of character strings. If you want to be able to type in all the flavors at once, you'd have to write a fairly complicated program just to split up a large string into smaller strings, one for each flavor. You would have to decide in advance how many flavors to allow for, and how many characters to allow in the name of each flavor. Logo's use of lists makes things so simple! You want a list of three flavors? Just type it in. If you want 31 flavors next time, no problem.

Logo's emphasis on lists is there for a reason. List processing has proven to be a very powerful tool for the manipulation of human language. A sentence is a group of words, not a string of characters some of which happen to be spaces. Language was chosen as a central application for Logo because it is something with which everyone is familiar. We don't all do algebra, but we all speak English or some other natural language. We play with words, whether by writing poetry or by speaking Pig Latin. In his article, Tinker gives this problem as an example of something which is easy to do in BASIC: "Suppose

one wanted to print out the tenth through the twentieth odd numbers." Sure enough, this is exactly the sort of thing which is easy to do in BASIC. It is *not* the sort of thing which any human being would find interesting!

It's quite true that the same lists which make the PRAISE program so simple can be used for more complicated purposes. Indeed, one of the design goals of Logo is expressed in the slogan "no threshold, no ceiling." This means that the same mechanisms which are available to beginning programmers also meet the needs of the most sophisticated experts.

Extensibility

Actually, though, when Tinker gives specific examples of what he finds difficult about lists, his difficulty isn't really about lists at all. It has to do with the idea of variable typing--that is, restricting particular variables to contain only a certain kind of object.

Tinker points out, correctly, that Logo variables are not typed, but that there are two types of objects in Logo, words and lists. (He identifies numbers as a third type, but this is incorrect. A number is simply a word which happens to be full of digits instead of letters. There is no special numeric object type.) He prefers the typed variables of BASIC, in which a dollar sign as part of a variable name indicates that the variable contains a character string.

The irony of Tinker's position is that untyped variables contribute to the extensibility which he praises in Logo. *Extensibility* is not a single well-defined property of a language, like list processing or procedural organization. Instead, it is an ideal to which languages come more or less close. The general idea is that users should be able to extend the power of the language in a smooth way. That's quite vague, so I'll try to give some examples.

Every programming language is a little bit extensible, simply because writing a computer program means extending what you were able to do before. But the extent to which the extension is well-integrated with the original power of the language varies. For example, in BASIC, there are about a dozen named things the language knows how to do (LET, INPUT, GOTO, and so on). But when a BASIC user invents a new thing, she can't give it a name, and create a new BASIC command. Instead, the new thing must be referred to by a line number, using the GOSUB command. Saying GOSUB 4360 is not mnemonic in the way that PRINT is mnemonic.

Logo is not perfectly extensible, but it does better than BASIC. For example, the

primitive procedure FIRST outputs the first element of its input. The procedure SECOND we defined earlier outputs the second element of its input. Although one is a primitive and one is a user-defined procedure, they are invoked in exactly the same way. Each one can be used wherever the other could be used. This similarity between the two procedures is an example of what I mean by "smoothness."

The primitive procedure FIRST can accept either a word or a list as its input. If the input is a word, the output is the first letter of the word. If the input is a list, the output is the first element (word or sublist) of the list. Logo's elegant handling of the hierarchy of sentence, word, and letter makes it an ideal medium for exploring human language.

The point I'm leading up to is that SECOND, too, can accept either a word or a list as input. That input is assigned to the variable named THING. (Look back at the procedure definition.) If Logo had typed variables, as Tinker recommends, we would have had to specify the variable THING as being either of type *word* or of type *list*. That would mean that the SECOND procedure would work only on words, or only on lists. We'd have to write two separate procedures to handle the two cases. Pascal, a typed language, suffers from exactly this problem.

One way in which Logo is *not* extensible is that it provides infix notation for arithmetic procedures (2+3 as distinct from the prefix form SUM 2 3), but users can't define their own infix operators. Another way is that some primitives will accept a variable number of inputs, but user-defined procedures must have a definite, fixed number of inputs. These are real weaknesses of Logo, not mentioned by Tinker.

Objects Needn't Live in Variables

In the background of Tinker's position on variable typing is the fact that he is accustomed to programming in BASIC, in which a datum rarely exists without being assigned as the value of a variable. In Logo, data are used as inputs and outputs of procedures, as in the bucket brigade examples earlier.

This difference explains Tinker's confusion about the specific question of keyboard typein. Tinker is thinking in terms of the INPUT command in BASIC. This command actually combines two purposes: it reads something from the keyboard, and it stores the result in a variable. The BASIC INPUT command, therefore, requires a variable name as part of the command syntax. BASIC interprets what it reads differently for different variable types.

Tinker is trying to interpret Logo's typein facility as if it worked like BASIC's. But in Logo, the reading of keyboard typein is separate from the assigning of values to variables. There is a primitive procedure called READLIST (abbreviated RL), which reads a line from the keyboard, and outputs a list containing whatever the user types. Another primitive, READCHAR or RC, reads only a single character from the keyboard and outputs it. That primitive is generally used for something like a video game, where a single keystroke has some immediate effect.

In earlier versions of Logo, the READLIST primitive was called REQUEST, a name which might leave room for doubt as to what kind of object is read. The change to READLIST is one example of the fact that Logo's designers continue to think about how the language can be improved. But if Dr. Tinker has trouble remembering that READLIST reads a list, I don't know what we could do to make it any easier for him.

It is the natural thing, in Logo, to represent a typed line as a list. The user can type anything on the line, including several words. The list is Logo's way of dealing with data aggregates. By accepting anything the user types, Logo makes it possible for a program to allow flexibility in the format of user typein. This flexibility is in sharp contrast with the situation in BASIC. For example, suppose you want to write a program which will read several numbers from the keyboard and average them. You'd like the program to work on any number of numbers. So when the user has entered all of them, she must have some way to indicate the end of the numbers. How about just a blank line? Or the word "done"? Sorry, those aren't numbers. BASIC won't let you read them with an INPUT command which is expecting to read a number. The restrictive typing of input in BASIC leads to ugly solutions like "type 9999 when done."

Sometimes, you want to write a program which expects to read only one word on a line. Therefore, one of the standard procedures everyone learns quickly is READWORD:

```
TO READWORD
OUTPUT FIRST READLIST
END
```

This procedure does not "convert [a list] to a number" as Tinker describes it. What it does is to extract the *first* element from the list the user types. If that element is a number, fine. But it's not *converted* to a number. It is Tinker's mental image of the type-constrained INPUT of BASIC which makes him consider this situation hard to understand.

User Interaction Through Procedure Inputs

More importantly, the situation he describes is fairly rare in Logo programming. BASIC programs do a lot of reading numbers from the keyboard, because BASIC lacks the idea of procedures with inputs. His example is about drawing a box of a specified size. He envisions writing a program something like this:

```
TO BOX
TYPE [WHAT IS THE SIZE?]
MAKE "SIZE FIRST RL
REPEAT 4 [FORWARD :SIZE RIGHT 90]
END
```

This is a Logo procedure in the sense that Logo can understand it, but it is *not* a Logo procedure, in the more profound sense that a Logo programmer wouldn't write it this way. Instead, we would write this:

```
TO BOX :SIZE
REPEAT 4 [FORWARD :SIZE RIGHT 90]
END
```

Instead of invoking the BOX procedure and then, separately, telling the procedure how big to make the box, a Logo user writes the procedure to take an input, so that the size is specified in the invocation itself. Tinker has transliterated a program he might write in BASIC, instead of considering the Logo style of programming.

Letting "Natural" Mean "Just Like BASIC"

Many people, not only Tinker, seem to have a vague but compelling notion of the "naturalness" of a programming language. It's as if all existing programming languages were approximations to some ultimate language, which we'd all understand by instinct. To each person, what seems "natural" is, of course, simply what that person happens to know best. I think that Tinker is afraid of lists because they're "unnatural" to him in this sense. (The advertisers who promote software as "English-like" are taking advantage of the same notion.)

I think this is a bad aesthetic for programming language design, because there really is nothing absolute about the sense of naturalness. I'd like to propose a better criterion: *coherence*. A well-designed programming language is one which hangs together *in its own terms*.

There is a specific illustration of this point which is related to the use of lists as procedure inputs, about which Tinker complains. There have been two different syntactic forms for conditional execution of a command in Logo. The older syntax was

this:

```
IF :X=0 THEN STOP
```

The word THEN is an optional keyword here. The IF procedure, in this syntax, takes one input, an expression whose value must be either the word TRUE or the word FALSE. In this case, the expression is :X=0. (We call such an expression a *predicate*.) If the predicate is TRUE, the rest of the line is executed. If it's FALSE, the rest of the line is ignored.

The trouble with this syntax is that it is the only one in Logo in which a procedure (IF) takes note of "the rest of the line." All other Logo procedures can take inputs, but the STOP command in the example above isn't exactly an input to IF. If it were, it would have to be evaluated before the IF procedure is invoked, just as :X=0 is evaluated. But then the procedure would stop before allowing IF to test the predicate. No, the STOP isn't an input to IF; it is just held waiting, and IF magically allows it to be carried out, or doesn't allow it.

To eliminate this incoherence in the syntax of IF as compared to the rest of Logo, later versions have used this new IF syntax:

```
IF :X=0 [STOP]
```

In this form, IF is a straightforward Logo procedure like any other. It takes two inputs. The first is a predicate, as before, and the second is a list containing Logo instructions. The effect of IF is that if the first input is TRUE, then the second input is executed as an instruction to Logo.

When this new syntax was introduced, even some expert Logo programmers objected to it as "unnatural." And yet, most versions of Logo have a procedure with this syntax:

```
REPEAT 4 [FORWARD 40 RIGHT 90]
```

The REPEAT procedure takes two inputs, a number and a list. It executes the list as Logo instructions repeatedly, the number of times specified by the first input.

Nobody considers REPEAT "unnatural." But it could have been invented this way:

```
REPEAT 4 TIMES: FORWARD 40 RIGHT 90
```

This would be more "English-like." But it has never been part of Logo, and nobody has ever suggested it.

Why is REPEAT perfectly "natural" the way it is, while the virtually identical IF is widely rejected? Simply because BASIC has an IF statement, but not a REPEAT statement! The hidden definition of "natural" is "just like BASIC." Even some people

intimately involved in the design of Logo fail into this trap. It's a particularly dangerous trap for someone trying to compare two languages, because it easily leads to the hidden circularity of saying, in effect, that BASIC is the best language because it's the most BASIC-like.

2. The "Assignment Statement"

Tinker's second complaint is about the MAKE procedure in Logo. "A second major stumbling block in the language is the distinction made in variable use between the value of the variable and the variable itself." This is not quite right; the distinction is between the value of the variable and the *name* of the variable. But more importantly, Tinker is quite mistaken in considering this a special problem of Logo.

To understand the issue here, we have to go back to the analogy about learning to ride a bicycle. There are some inherently difficult ideas in programming, and the use of variables is one of them. The difficulty exists in all programming languages, no more so in Logo than in any other. It has nothing to do with syntax; it's a real problem about the meaning of a variable.

Part of the problem is that the word "variable" is often used in algebra to represent something which is not variable at all, but rather is a *constant* value in a problem, which we happen not to know yet. Although we don't know its value, we do know that its value is fixed; it isn't suddenly going to change half way through solving the equations. In programming, variables are really variable.

But what a programmer must understand about variables, before anything else, is that there are two things you can do to them: you can put something in, or look at what's already in. The first of these is variable *assignment*; the second is variable *evaluation*. Tinker calls this distinction "somewhat arbitrary," but it is nothing of the sort. That's like suggesting that the distinction between reading a book and writing one is insignificant.

I don't think this distinction is "difficult to remember," as Tinker suggests. Nobody thinks that putting something in a box is the same as taking something out. But if Tinker is worried that it might be hard to remember, *all the more reason* to make the distinction more apparent by being explicit about the process of variable evaluation. That's what the colon does.

It is BASIC which creates a stumbling block, by blurring this distinction. In the BASIC assignment statement

```
LET X=Y
```

the two variables X and Y are being used in very different ways. Y is being evaluated, and X is being assigned to. The paradoxical case X=X+1 which Tinker mentions is a boon, if anything, because it rubs the beginning programmer's nose in the fact that he doesn't understand the X=Y case *either!*

The issue really isn't about the precise form of the assignment statement at all. BASIC blurs the distinction between assignment and evaluation in all contexts. For example, you can say

```
PRINT X
```

or

```
INPUT X
```

and these two statements look very similar. But you can say

```
PRINT X+1
```

and yet you *can't* say

```
INPUT X+1
```

Why not? The beginning BASIC programmer has to figure this out, and the language makes it harder by trying to avoid a distinction which is inherent in the nature of programming.

BASIC is designed around the idea that programming should be "easy" in the sense of *seeming* easy. Let's bury hard concepts under the rug, so we don't scare people away. The difficulty is that when the problem gets out from under the rug, as it inevitably must, the BASIC user is unprepared. It's as if someone invented a bicycle with *invisible* training wheels, so that the rider couldn't fall over, but wasn't aware of the need to balance. Such a rider might be convinced that bike riding is "easy," but the conviction wouldn't survive her first ride on a real bike.

Logo, on the other hand, is based on the idea that an "easy" language means, in part, one which brings the important ideas out in the open.

In Logo, the expression :BAZ means "the value of the variable named BAZ." If what you want is the name itself, that's just a normal Logo word, represented as "BAZ. Consider this Logo instruction:

```
PRINT SE "NUMBER :NUMBER
```

The procedure SE (for SENTENCE) takes two inputs and puts them together to form a list. In this case the first input is the literal word NUMBER, and the second is the thing stored in the variable NUMBER. The result of executing this instruction might be to print this:

NUMBER 17

It should be clear, in this example, that two syntactic notations are used (quote and colon) because two different ideas are being represented.

Exactly the same thing is happening in the case of

```
MAKE "WHERE :POSITION
```

The procedure MAKE takes two inputs. The first is the name of a variable, and the second is the thing you want to put in that variable. In this example, we are copying the thing found in another variable, POSITION. But we could also have said

```
MAKE "WHERE "POSITION
```

which would mean something quite different: put the word POSITION *itself* into the variable named WHERE. Or we might have said

```
MAKE "WHERE POSITION
```

which would mean a third thing: invoke a *procedure* named POSITION and put whatever it outputs into the variable named WHERE.

In this regard, there is nothing special about variable assignment. The three examples might just as well have been

```
PRINT :POSITION
PRINT "POSITION
PRINT POSITION
```

with the same three interpretations of POSITION.

The Power of Consistent Notation

In the vast majority of cases, the first input to MAKE is a quoted word, like "WHERE in the examples above. Why didn't Logo's designers save people a little typing by letting them leave out the quotation mark in the particular case of the "assignment statement"? Tinker says, "It is difficult to understand the rationale for having the first argument a quoted variable."

The point is that there is no such thing as the "assignment statement" in Logo. Again, Tinker is thinking in terms of BASIC. That language has about a dozen distinct kinds of "statement," each with its own *ad hoc* syntax unrelated to the others. For example, the BASIC PRINT statement can include several expressions to be printed. The expressions can be separated by commas or by semicolons, with slightly different results. (In fact, expressions separated by semicolons are more closely connected than those separated by commas, which BASIC prints further apart. This is the opposite of the way these punctuation marks are used in English, in which the semicolon is a stronger

separator than the comma. To me this seems "somewhat arbitrary and often difficult to remember." Two can play at this game.)

Logo does not have a separate *ad hoc* syntax for every possible instruction to the computer. Instead, there is one syntax, used both for primitive commands and for user-written ones. This fact is the most fundamental basis for the extensibility Tinker praises. The syntax is that to invoke a procedure, you say its name, followed by the inputs you want to give it. Logo first evaluates the inputs, and then invokes the procedure. (The evaluation of the inputs may involve the use of other procedures. This is the bucket brigade situation we discussed earlier.) It doesn't matter what the procedure is; the rules for evaluating its inputs are always the same. If you want to use a literal word as an input, you quote it. MAKE is not a special case.

The syntax of MAKE is what it is to maintain consistency and clarity. A Logo programmer who understands how Logo works in general will understand how MAKE works without having to memorize awkward exceptions. This clarity is, as I've said, part of the extensibility of Logo. That's why MAKE is the way it is, and not to make possible the "clever, but... confusing" indirect assignment which Tinker shows in his article. But that power is an added bonus. Consider this procedure:

```
TO INCREMENT :VARIABLE
MAKE :VARIABLE (THING :VARIABLE)+1
END

?MAKE "COLOR 88
?PRINT :COLOR
88
?INCREMENT "COLOR
?PRINT :COLOR
89
?
```

As you can see in the example, the input to INCREMENT is the name of a variable. The procedure adds 1 to the value of that variable, and makes the sum be the new value of the variable. The MAKE command inside INCREMENT does not assign a new value to the variable VARIABLE, but instead to the variable whose name is in VARIABLE. In the example, it is the variable COLOR which is incremented.

This is an advanced use of Logo. It's not something we would expect a beginner to use, or to understand. (Readers should be aware of the dilemma I face in picking examples for this article. I am trying to rebut the claim that Logo is only good for beginners, and I can't do that without using advanced examples. But I am writing for an

audience which includes people who are not Logo experts. So if you don't understand an example, that's not because Logo is "hard" in Tinker's sense, only because Logo *is* suitable for studying advanced ideas!)

The "Dummy Variables" Smokescreen

In discussing the use of variables in Logo as inputs to procedures, Tinker refers to these inputs as "dummy variables." This is not a term used by Logo designers or teachers. It's rather a frightening term, and perhaps it's worth a paragraph to dispel the fear by explaining what it means.

The name "dummy variable" is a holdover from the first FORTRAN compilers of the 1950s. In FORTRAN, and also in BASIC, most variables are "really" names for a particular location in the computer's memory. When you say X, the computer knows that you really mean memory location number 437. The earliest FORTRAN programmers were accustomed to machine language programming, in which it's the programmer's job to know exactly where everything is stored in memory. It turns out that the variables used for procedure inputs were called "dummy" variables because they do not, like "real" variables, correspond to a particular memory location.

The fact is that *no* Logo variable represents a particular memory address in this way. But a much more important point is that the user of a programming language should not have to worry about any of this! The whole point of inventing high-level programming languages is to allow people to think about the problems they want to solve, and *not* about what is stored where in memory. No child who hasn't been ruined by BASIC has any trouble understanding that procedures can have boxes associated with them, called inputs, and when you use a procedure you have to put things in the boxes first. This is only "very abstract" to Tinker because it is not the way *he* thinks about programming.

3. Indexing

Tinker's third complaint is that Logo lacks an equivalent to the FOR-NEXT loop in BASIC. There is a deep issue behind this question, but it is not the issue he sees. The issue is one of dealing with data aggregates.

In BASIC, the data aggregate is the array. An array is a group of objects. The number of objects in an array is fixed in advance, and the objects must all be of the same type. Each object in the array is identified by a number. That is, if `X` is an array, then `X(3)` is the third thing in the array.

In Logo, the data aggregate is the list. A list, too, is a group of objects. But the size

of a list is not fixed in advance, and the elements of a list may be of different types. Since the size of a list is not fixed in advance, the FOR-NEXT mechanism is not appropriate for the problem of doing something to each element in turn.

To make this clearer, consider this fragment of a BASIC program:

```
10 DIMENSION X(10)
   ...
50 FOR I=1 TO 10
60 PRINT X(I)
70 NEXT I
```

Lines 50 to 70 print all the elements of the array X. Because we know in advance that there are exactly 10 elements, it is simple to refer to the range of index numbers in line 50.

A somewhat similar Logo procedure might be written this way:

```
TO ONE.PER.LINE :LIST
IF EMPTYP :LIST [STOP]
PRINT FIRST :LIST
ONE.PER.LINE BUTFIRST :LIST
END
```

In writing this procedure, we don't know in advance how many elements are in the list which is its input. So instead of counting them, we use a different mechanism: We take the list elements one by one, starting from the beginning, and continue until we have emptied out the list. This approach is certainly different from the array-index idea. But it isn't any harder. In fact, in one sense it is easier; we have avoided introducing the *auxiliary variable* I which was required in the BASIC program.

Is Recursion Hard?

We've already seen several examples of one procedure invoking other procedures. The particular case in which a procedure invokes *itself* is called recursion.

Here, more than in any other issue, we have to think about what "easy" and "hard" mean in terms of our educational goals. There is no doubt that recursion is a "hard" concept, in the sense that it takes some struggle to understand it fully. The difficulty isn't arbitrary; it's not like the bike pedals on the handlebars. Instead, it reflects the tremendous mathematical power of the idea of recursion: to solve a problem, it sometimes helps to solve a smaller version of the problem as the first step.

Logo learners don't have to understand the full power of recursion to be able to perform repeated actions in Logo. Beginners can start with a procedure like this:

```
TO POLY :SIDE :ANGLE
FORWARD :SIDE
RIGHT :ANGLE
POLY :SIDE :ANGLE
END
```

This is how you draw a regular polygon in Logo turtle graphics. POLY takes two inputs: the length of a side, and the angle to turn between sides. It draws a side, turns once, then invokes the same POLY procedure with the same inputs. This starts the procedure again from the beginning.

As a programming tool, this procedure has a severe limitation. It keeps running forever, retracing the same polygon. So it can't be used as part of a bigger program which also does other things. But it has the advantage that everyone understands how it works.

The next step in understanding recursion might be to change the values of the inputs in the recursive invocation:

```
TO SPIRAL :SIDE :ANGLE
FORWARD :SIDE
RIGHT :ANGLE
SPIRAL :SIDE+1 :ANGLE
END
```

This procedure SPIRAL is just like POLY, except that it uses :SIDE+1 instead of :SIDE as the first input to itself in the recursive instruction. This means that if the SIDE input was, say, 20 when SPIRAL is called the first time, it will be 21 the second time, 22 the third time, and so on. The effect is that instead of a polygon, it draws an outward-growing spiral.

There are several more things to learn about recursion in Logo, and I don't have room to teach them all here. One example is how to get a recursive procedure to stop. But we've actually already done that several times, most recently in the ONE.PER.LINE procedure. In any case, the only point I want to make now is that beginners *can* use recursion, in simple ways, while more advanced programmers can learn to benefit from the power which it has, which iteration does not have.

Just because a bicycle *can* do wheelies doesn't mean you *must* do wheelies the first time you ride one. But knowing that you can learn to do wheelies later helps make even the first awkward practice sessions appealing.

Is Iteration Easy?

I've explained that the recursive structure of Logo is well-matched with its list processing capability, because a variable-length list can't be handled easily by the FOR-NEXT style of fixed-count iteration.

But is FOR-NEXT so easy? Tinker writes as if everyone were born already familiar with that notation. But it has its own pitfalls. How do you know, when you see a FOR statement, what range of statements is to be repeated? The corresponding NEXT may be later on the same line, or buried 400 lines down. In the Logo approach, the range of instructions to be repeated forms an entire procedure. What happens if the final value given for the index variable is less than the initial value? The answer to this question is "somewhat arbitrary and often difficult to remember."

The short BASIC program fragment I gave earlier works well enough when we know how big an array is, and we want to print all of it. But suppose in a real situation we don't know how many elements we want to print until we've examined some or all of them? As soon as the task gets at all complicated, the simple-looking FOR-NEXT structure becomes *too* simple to do the job. Recursion is flexible enough for any task.

Tinker's actual example, though, was not so sensible as printing the elements of an aggregate. Instead, he uses an artificial example of printing out a range of odd numbers, for no particular reason. In this example, the variable I is not auxiliary. It is actually the thing he wants to print. So for this artificial example, the BASIC style does seem natural. But the example is a cheat--that's not what people really want to do with their computers.

You Can Have Iteration If You Want It

If you do want the equivalent of a FOR-NEXT loop in Logo, it is easy enough to invent one. What we want is a procedure with four inputs: the name of the index variable, its first value, its last value, and the thing to do repeatedly:

```
TO FOR.NEXT :VAR :FIRST :LAST :COMMAND
MAKE :VAR :FIRST
IF :FIRST > :LAST [STOP]
RUN :COMMAND
FOR.NEXT :VAR :FIRST+1 :LAST :COMMAND
END
```

Tinker's example would then be written this way:

```
FOR.NEXT "I 10 20 [PRINT 2*:I-1]
```

Now, the FOR.NEXT procedure is not something a beginner would write on the first day

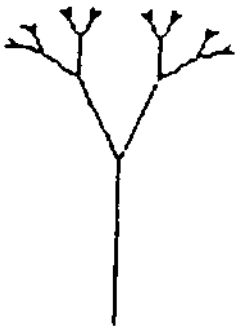
of using Logo. But it certainly is something a teacher could provide, if it seems really to be the natural way of solving an actual problem. However, the problems Logo programmers undertake to solve are not usually ones for which this is a good style.

The FOR-NEXT loop is only one of many possible iterative control structures. More powerful languages which share BASIC's iterative approach provide things like WHILE (do this while some arbitrary condition remains true), UNTIL (do it while the condition is false), and a more general sort of FOR (allow any updating instruction between iterations, not just incrementing a variable). Because Logo instructions can be manipulated as data, using the list processing ability of the language, and then executed with the RUN command, any of these iterative structures can be implemented straightforwardly in Logo.

When Iteration Won't Do

Consider this turtle graphics procedure:

```
TO TREE :SIZE :DEPTH
IF :DEPTH=0 [STOP]
FORWARD :SIZE
LEFT 25
TREE :SIZE/2 :DEPTH-1
RIGHT 50
TREE :SIZE/2 :DEPTH-1
LEFT 25
BACK :SIZE
END
```



```
TREE 100 5
```

This is a somewhat more advanced use of recursion than POLY or SPIRAL. It is based on the fact that the left and right branches of the tree are each smaller versions of the tree itself. A program structure like this one *cannot* be built out of the simple iteration BASIC provides. To do this in BASIC, you would essentially have to invent recursion yourself. The program would be much longer, and extremely hard to understand. In

Logo, it's not "easy" to understand, but it is no harder than it has to be in light of the complexity of the task it performs.

4. Not Enough Memory

Tinker complains that the existing implementations of Logo are too slow and too small. He's quite right. The current generation of 8-bit microcomputers can barely support a powerful language like Logo.

But he's less right than he was a year ago. The two versions of Logo for the Apple, written a few years ago, require 64K bytes of memory. Most of that is taken up by the interpreter itself. Atari Logo, just released, fits in an Atari 400 computer with only 16K bytes for the interpreter itself, leaving another 16K for user procedures and data. On the larger Atari models, there is much more user memory.

Logo is also becoming available for the IBM PC. This 16-bit processor has an ample memory for just about any imaginable Logo program. The IBM is the first of a growing family of 16-bit machines. Soon they will be the standard, and the memory problem will disappear.

Even on existing machines, the space problem is not so terrible. We've seen many complex and interesting projects done on 8-bit microcomputer versions of Logo.

The space shortage in 8-bit micros affects not only the user workspace but the language itself. Some implementations lack primitives like MEMBERP or ITEM because there wasn't enough room to include them. Older research versions of Logo included things like arrays as well as lists, which are not in any currently available implementation. It would be nice to include even more advanced features, like a fast pattern matcher. Hardware improvements should make these extensions to Logo possible within a couple of years.

D. General Remarks

In this concluding section, I'd like to move beyond Tinker's objections to Logo. I'd like to compare Logo with BASIC from my own point of view, and then move on to consider ways in which the philosophy of Logo can be extended in further language development.

The Forest and the Trees

Here are two criticisms I've made of BASIC earlier in this article:

- BASIC is not procedural. It does nothing to encourage a modular programming style.

- The use of commas and semicolons in the BASIC `PRINT` statement is backwards from their use in English.

Both of these criticisms are valid, I think, but they are certainly not equally important. The first is a fundamental property of the language. It pervades every aspect of BASIC programming. It is, I think, a strong objection to the use of BASIC by anyone for any purpose. The second objection, although a real one, is a detail. It could easily be fixed. Even if not fixed, it is not a real reason to avoid BASIC or to prefer a different language.

I say all this because I'm afraid that in answering all Tinker's detailed points about things like quotes and colons, I may have encouraged you to lose track of what is really important about Logo and its competitors. The essential thing to remember is that Logo is a powerful language, which implements a consistent philosophy of programming in a way which is accessible to people at all levels of expertise. BASIC is a toy language which is okay for listing odd numbers, but inadequate to any more interesting project.

Not long ago I wrote a video game program in Atari Logo. The program displays four spaceships on the screen. When two ships collide, the program notices, and issues sound effects and flashing lights. Each ship can be steered either randomly, by the program, or under control of a joystick, by a human player. There can be any number of players from zero to four. The program steers all the ships when it starts, but it notices the operation of a joystick, and relinquishes the associated ship to the player using the joystick. The ships are shaped like spaceships, and move against a background of stars.

All of this fits in 65 lines of Logo. It took me about four hours to write and debug it. There are 10 procedures in the 65 lines, each with a specific job to do.

I don't think I could have written this program in BASIC at all. Certainly it couldn't be so small and elegant, and it would take weeks rather than hours.

Several years ago, a colleague of mine wrote a very sophisticated pattern-matching program in Logo. It took ten lines. I'm not sure anyone could write the same program in BASIC in any length of time.

To say that Logo has limits is to say the obvious. Logo is the finite product of fallible human minds. But to say that it is suitable only for beginners in elementary schools is ludicrous.

Player-Missile Graphics: A Case Study

As a concrete illustration of the profound difference in design philosophy between Logo and BASIC, I'd like to discuss the support of the special-purpose animation hardware of the Atari Home Computers in the two languages. The point of the example is not that the hardware details are important in themselves, but that the way in which each language interprets the hardware is revealing about the languages.

The hardware in question is designed to allow small, animated objects to move quickly across the TV screen against a more stationary background. This is a useful capability for video games, which are populated by spaceships or asteroids or gorillas who move in this way. The hardware was designed to be adequate and inexpensive, rather than elegant. There are four "players" and four "missiles"; the difference is that players are eight dots wide, while missiles are only two dots wide. Both are, in effect, columns as tall as the TV screen. Typically, though, the shapes they are used to display are not so tall, but are more nearly square in extent. To move the player or missile horizontally requires only changing one number in a horizontal position register, but to move it vertically requires the programmer to copy the numbers representing the actual shape from one place in computer memory to another place. Moving diagonally requires the combination of both operations. Moving *smoothly* diagonally is quite complicated!

Before a programmer can use player-missile graphics, he must first allocate a large block of computer memory. This block can't start at any old memory address; it must be on a "page" boundary. (Never mind if you don't know what that means. The point is simply that it's a detail which must be attended to, and one which is of no intellectual interest.) Then the block must be filled with zeros, except for the parts of each column where the desired shape is filled in.

To do all this in Atari BASIC, the programmer must study many arcane details. For example, there are particular magic addresses in memory where the programmer puts things like the horizontal position information. Allocating the block of memory for the shape information is mysterious, especially because of the special requirements on where it can be located. Once all this is done, moving a shape horizontally is pretty easy, but moving it vertically requires copying the shape information with a slow and complicated FOR-NEXT loop. The program which results is full of PEEKs and POKEs.

This complexity meant that for a long time, hardly anyone managed to use the animation hardware except for the professional game programmers at Atari. But when Atari introduced a version of Microsoft BASIC for their computers, the advertising announced with great fanfare that "Microsoft BASIC supports player-missile graphics." What does this mean? It turns out that, of all the complexity involved, what the Microsoft designers had noticed was the slowness of the FOR-NEXT loop to move a shape vertically. So they invented a new command to copy a block of memory from one place to another. They did *nothing* about the complexity of the hardware. Microsoft BASIC users still need to know all the arcane magic numbers, and their programs are still full of PEEKs and POKEs. Vertical motion, while faster than before, is still done very differently from horizontal motion.

The designers of Atari Logo took a very different approach. The four players are used as Logo turtles. For the Logo programmer, they can be moved in any direction, horizontally, vertically, or diagonally, with the same straightforward commands. What the programmer says represents the desired motion in an understandable way. The programmer can focus her attention on the intellectually interesting part of the problem she is trying to solve, rather than concentrating on arcana.

Logo achieves this simplification by restricting itself to only part of what the hardware can do. It ignores the missiles altogether. The shape editor, used to create new shapes for players, thinks of them as boxes 8 dots wide and 16 high, not as columns the entire height of the TV screen. But programmers almost always want to use the players in the way Logo understands.

The moral of the story is that for BASIC designers, "support" means only making the program run faster. They do not provide *intellectual* support. For Logo designers, support means not only speed but also providing a metaphor, a framework in which the programmer can use the hardware in a way which makes sense.

New Directions for Logo

Logo excels as a programming language because it provides powerful control structures (procedures) and data structures (lists). These structures were the state of the art in computer science when Logo was first developed 15 years ago. Today, Logo is still the most powerful language widely available to nonprofessionals, but it is not the state of the art.

In data structures, one important new idea is the object-oriented programming of the

language Smalltalk. Instead of the traditional notion of "smart" procedures manipulating "stupid" objects, Smalltalk introduced "smart" objects which in some sense "know how" to carry out particular tasks. The Logo TELL command, used to direct graphics commands to one of several different turtles, is inspired by the object metaphor. But true Smalltalk-like objects have not yet been part of Logo. An experimental system called qLogo, being developed by Gary Drescher at Atari Cambridge Research, introduces the power of object-oriented programming while retaining most of the traditional flavor of Logo.

In control structures, probably the most important new development is that of multiple concurrent processes. This capability is important both as a good metaphor to describe projects which aren't strictly linear in nature, and also to reflect current hardware developments, which have made processors inexpensive. The *collision demons* of Atari Logo are a primitive form of multiprocessing. A demon is a process which simply waits for some event to occur; in Atari Logo, demons wait for collisions among the animated turtles. When a collision happens, the demon executes some Logo instruction specified by the programmer. Demons were an early form in which the idea of multiprocessing was introduced in artificial intelligence research. More flexible forms of multiprocessing are possible, and will probably be added to Logo in the future.

Inventing New Languages

Tinker's solution to the problems he sees in Logo is to invent new languages of his own, and to call them Logo. He has, of course, every right to invent new languages, and I hope he succeeds in making programming easier for beginners with them. But it is extremely misleading, not to say dishonest, for him to call his languages "Logo" if they embody a philosophy of computing which is the opposite of what the real Logo represents.

Summary

This has been a long discussion. Here's a brief summary of what I've been trying to say:

- Logo isn't perfect.
- But it's a damn sight better than BASIC!
- Tinker's specific objections to Logo are based on misunderstandings of what Logo is all about.

- One reason for these misunderstandings is that every programming language teaches a style; BASIC programmers have trouble breaking out of the BASIC style of thinking about programming.

- What is really important is not the detailed syntax of a language, but its intellectual content. For Logo, this content includes modularity, recursion, and the use of metaphor.

- The right reason to teach programming in the first place is to teach that intellectual content, not just to breed programmers or "computer literates."