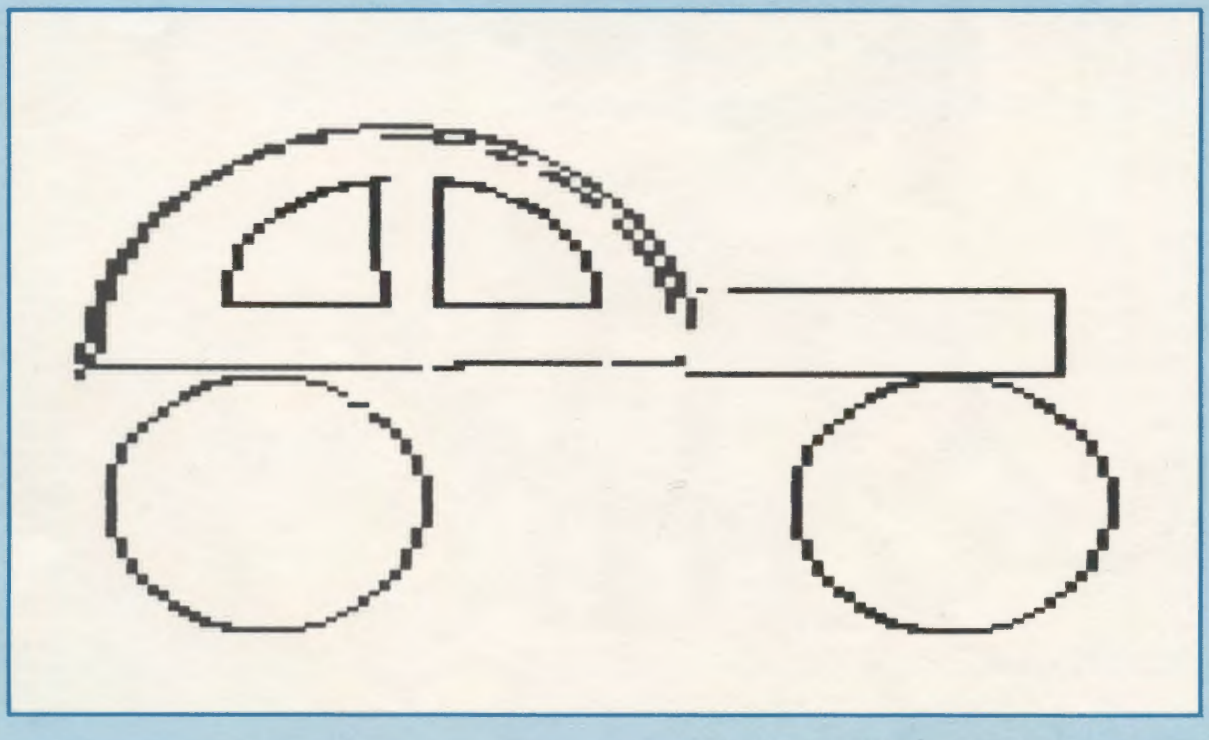
Journal of the ICCE Special Interest Group for Logo-Using Educators

# LOGO EXCHANGE

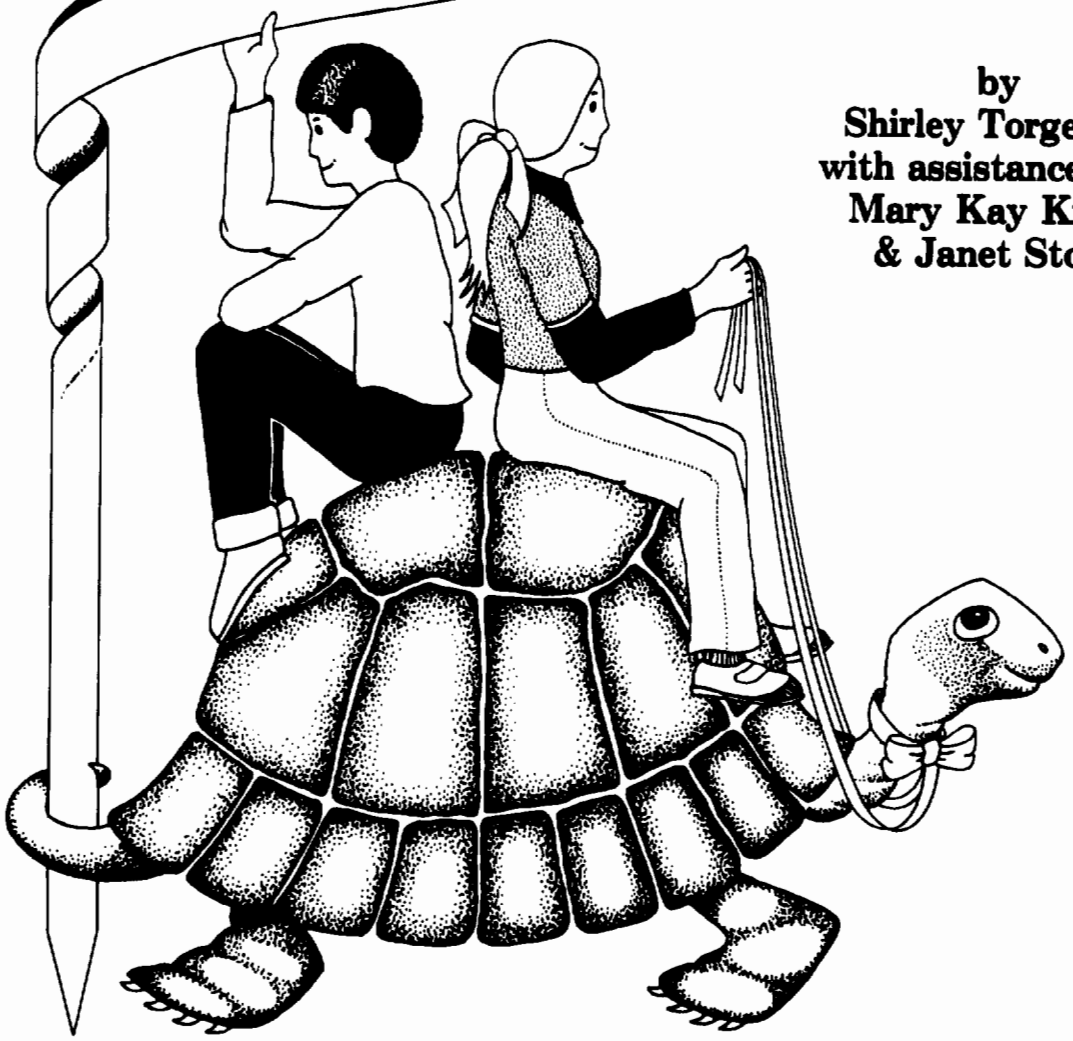
MARCH 1988 VOLUME 6 NUMBER 7



International Council for Computers in Education

ICCE Publications

# Logo in the Classroom



by
Shirley Torgerson
with assistance from
Mary Kay Kriley
& Janet Stone

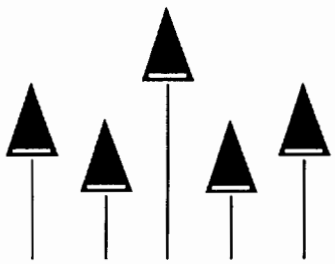
*Logo in the Classroom* integrates Logo into your elementary curriculum. Twenty lessons were developed in a classroom setting as a response to "How can Logo work in a classroom where computers are in short supply?"

Detailed teacher information is given for each lesson along with copyable practice sheets, transparency masters and 12 charts. Useful for teacher inservice.

**$13.00 (US)** plus $2.50 shipping.

INTERNATIONAL COUNCIL FOR COMPUTERS IN EDUCATION ICCE

University of Oregon 1787 Agate St. Eugene, OR 97403



# LOGO EXCHANGE

VOLUME 6 NUMBER 7 Journal of the ICCE Special Interest Group for Logo-Using Educators MARCH 1988



Logo Exchange is the journal of the International Council for Computers in Education Special Interest Goup for Logo-using Educators (SIGLogo), published monthly September through May by ICCE, University of Oregon, 1787 Agate Street, Eugene, OR 97403-9905, USA.
POSTMASTER: Send address changes to Logo Exchange, UofO, 1787 Agate St., Eugene, OR 97403.

## CONTENTS



SIGLogo Membership (includes *The Logo Exchange*)

| | U.S. | NON-U.S. |
|---|---|---|
| ICCE Member | 24.95 | 29.95 |
| Non-ICCE Member | 29.95 | 34.95 |
| ICCE Membership (includes *The Computing Teacher*) | 28.50 | 31.50 |

Send membership dues to ICCE. Add $2.50 for processing if payment does not accompany your dues. VISA and Mastercard accepted.



The *Logo Exchange* is produced on a Macintosh SE and Laser Writer donated by Apple Computer, Inc. (LOCAL "PAGE)

# From the Editor

## Logo is Dead...Long Live HyperCard?

**Wait! Don't stop! Read on...**

Each fall and spring, computer conferences seem to cluster into a few short months, keeping some of us "on the road" for weeks at a time. I have just returned from attending a series of such conferences. At each of these conferences, someone said to me "I hear that Logo is pretty much dead." While they don't continue with "but HyperCard lives," the flavor is still there: HyperCard sessions are packed, those who haven't tried HyperCard are apologetic, and one can overhear much chatter between sessions and in exhibit areas about the wonders of HyperCard. Even the *Logo Exchange* has included several articles about HyperCard.

The "hype" about HyperCard reminds me of some of the things that we have heard over the years about Logo. To listen to the chatter, one would think that HyperCard is going to solve the problems of education, not to mention the problems of the world. Although HyperCard is a fascinating product with a great deal of potential, I find myself wondering if the "HyperCard community" is going to turn this new software into a "cause" in much the same way as the Logo community has at times "oversold" the virtues of Logo.

Research on Logo is simply not confirming many of the glowing claims made by Logo leaders and Logo teachers. Some feel that most of these studies are flawed or that they fail to examine facts that we as Logo teachers know are important. Nonetheless, they *are* affecting the educational community's attitude towards Logo.

Why hasn't Logo fulfilled its promises? Why aren't more people using Logo? Why aren't research studies confirming the value of Logo? Certainly, no one has easy answers to these complex questions. However, in this issue of *LX* we are taking one small step towards addressing one of these concerns. A frequently stated reason for teaching Logo is that it will enhance the problem-solving abilities of its users. Few research studies bear this out. The article "Powerful Ideas in Problem Solving and Logo" addresses this issue and attempts to point towards a solution. We hope you will find it of interest.

Is Logo dead while HyperCard lives? I doubt it, and most likely so do you loyal readers of *LX*. But what *should* Logo become? More like HyperCard? More powerful? Faster? Broader (e.g., Lego Logo)? More like existing applications packages? I'd be interested in your thoughts...let's get a "Letters to the Editor" column going.

Long live Logo!

*Sharon Burrowes Yoder, Editor*
The Logo Exchange
*ICCE, University of Oregon*
*1787 Agate St.*
*Eugene, OR 97403-9905*

# Monthly Musings

## Hyper Hype II

### by Tom Lough

Last month, I wrote about my initial reaction to the HyperCard program recently released for the Macintosh. Since then, I have had the opportunity to explore this software in a little more detail. The more I worked with it, the more I sensed a kinship with Logo.

At first glance, HyperCard appears to be a sophisticated information manager which uses a stack of index cards as its organizational metaphor. Backgrounds, fields, and buttons can be combined with graphics to organize and present many different kinds of information. Users can control the environment of a stack by writing scripts with the HyperTalk language.

HyperTalk consists of English-like commands with reasonably sensible syntactical organizations. Thus, reading a HyperTalk script is not a totally befuddling experience. This self-documenting characteristic reminds me of Logo. [It is possible to put comments with a HyperCard script, nevertheless.]

The scripts perform in a manner similar to the WHEN demons of Atari Logo. In that particular implementation, Logo checks continuously the state of a specified condition, such as whether two particular turtles have collided or whether a particular turtle has encountered a line of a specified color. WHEN the condition is true, then a list of specified Logo instructions is carried out. Thus, a typical WHEN statement might be:

```
WHEN 17 [ PRINT "POW" TOOT 1 440 7 6]
```

Here, condition 17 corresponds to checking whether turtle #1 and turtle #3 have collided. WHEN this happens, Atari Logo prints "POW" and uses voice #1 to TOOT the tuning note A (440 cycles per second) at a medium volume (7 on a scale from 0 to 15) for 6/60ths (one-tenth) of a second.

Instead of using "WHEN," each HyperCard script begins with the word *on* and a specified condition, such as whether the mouse button has been pressed or released. This is followed by a set of HyperTalk instructions. When (or, *on* the occasion that) the condition is true, then the instructions in the script are carried out. A common script is the following:

**Cover:** Andy Houck, grade 3, is in the Resource Enrichment program at the Conover Road School in Colts Neck, New Jersey. He designed the cover art using the arc and circles on his own without using the tool procedures.

## Monthly Musings — CONTINED

```
on mouseUp
   go to the next card
end  mouseUp
```

This script is activated by the mouse button. The mouseUp condition refers to a message sent by the computer to indicate that the button on the mouse, having been depressed, has been released. On that occasion, HyperCard then goes to the next card in the stack and displays it on the Macintosh screen.

Although the WHEN demons of Atari Logo might help one understand better the general operation of HyperCard scripts, there is a fundamental difference. The WHEN demons are global. That is, the WHEN command is given only once. Regardless of what Atari Logo is doing, it is always checking the specified condition, ready to carry out the associated instructions. In HyperCard, however, the scripts are local. That is, each script is associated with a particular object, such as a card, a stack, a background, a field, or a button. When an object is active, the scripts associated with it can carry out their instructions when the appropriate conditions are met. This allows different scripts to be written for different buttons, fields, or other objects.

In addition, there is an established hierarchy of objects which channels signals (or messages) upward for activation of scripts. This hierarchy has HyperCard itself at the top, and proceeds downward through the Home Stack, other stacks, backgrounds, and then cards. Both buttons and fields are subordinate to cards. Thus, if a button is clicked, then HyperCard searches first through the button's scripts to see if there is anything to do on mouseUp. If it finds an on mouseUp script, then it will carry out the instructions. For example, if the script above were found, then the next card would appear on the screen. If there is no on mouseUp script associated with the button, then HyperCard searches the scripts of the card, then the background, then the stack, the Home Stack, and finally HyperCard itself. At any point, if it finds an on mouseUp script, it would carry out the instructions and return to the user. If none is found, then nothing is done.

Although the local nature of the scripts and the consideration of the hierarchy were something I had to learn more about, the similarity between the script idea and the WHEN command of Atari Logo was somewhat comfortable for me. But I still felt incomplete. I wondered, "Is there anything in HyperTalk which is comparable to a Logo procedure?" As it turned out, there is: the HyperTalk function.

Although HyperTalk comes with many predefined functions [similar to primitive procedures], it is possible for you to define your own. Just type "function" followed by the selected function name and the names of any inputs. On the next lines, type the instructions, and finish up with "end" and the function name once again. Once defined, the function names can be used like other HyperTalk instructions. Here is an example adapted from Danny Goodman's *The Complete HyperCard Handbook* (Bantam Books) of a function to calculate and return the value of a factorial of an integer given as input:

```
function factorial n
   if n < 2 then return 1
   else return n * factorial (n - 1)
end factorial
```

The corresponding Logo procedure might look like this:

```
TO FACTORIAL :NUMBER
 IFELSE :NUMBER < 2
   [OUTPUT 1]
   [OUTPUT :NUMBER * FACTORIAL(:NUMBER - 1)]
END
```

Once again, the similarity with Logo helped me to understand the functions of HyperTalk. The general organization of the definition of each is about the same. The factorial example also shows the recursive ability of HyperCard (although when I used numbers greater than 17 or so for the input, I got an interesting error message, protesting "too much recursion.").

To call the Logo procedures, you could type:

```
PRINT FACTORIAL  5
120
```

Because the FACTORIAL procedure is an operation which provides an output as a result, you must provide a place for the result. Here, it was used as input to the PRINT command.

To call the HyperTalk function, you could type:

```
put factorial (5) in field 1
```

HyperTalk places the value of 120 into field number 1 of the specified card.

Logo procedures, like the WHEN statements, are stored in a global workspace, and are accessible from practically anywhere within Logo. On the other hand, HyperTalk functions, like the "on" scripts, are stored with particular objects (buttons, fields, etc.), and are local to those objects and those lower in the object hierarchy. Through this hierarchy, you can make functions available to specified groups of objects. For example, if you define a function which will be used from all levels of Hyper-

## Monthly Musings — CONTINUED

Card, then you could associate the function with the Home Card.

Although the function capability of HyperCard is made for operations, it is possible to write functions which perform as commands.

The similarities between Logo and HyperCard have helped me to learn more about this exciting new computer tool, and I commend it to you. I believe that, with the attitudes and experiences developed during the past several years with Logo, many educators will be able to recognize and capitalize on the educational power of HyperCard. As always, I would be interested in your work.

```
FD 100!
```

*Tom Lough*
*PO Box 5341*
*Charlottesville, VA 22905*

**LOGO CONNECTIONS:**

A Two-Week Logo Retreat will be offered at the University of Virginia in Charlottesville, VA, August 1 - 12. The intent of the retreat is to bring together teachers across the country who are interested in connecting with each other and learning how to create "Logo Connections." From its beginning, Logo was designed as a language which could be connected to objects in the outside world, such as switches, motors, lights, and other sensors. In addition, Logo work has always emphasized the importance of the social side of learning through connections with colleagues and friends. Both of these aspects of connections will be emphasized at the retreat in an informal, hands-on environment.

The workshop faculty of Glen Bull, Gina Bull, Judi Harris, Tom Lough, and Cheryl Wissick have a wide range of interests and expertise, ranging from pendula in physics to switch inputs for handicapped users. Among them, they have written numerous books and several hundred articles on Logo.

The two-week retreat will be limited to a maximum of twenty participants. Air conditioned dormitory rooms will be available at summer rates for those who require housing. Three hours of graduate credit will also be available through the university, on an optional basis. A wide variety of fun-filled social experiences throughout the two-week period will emphasized the importance of personal relationships in teaching with Logo. Participants must have intermediate Logo programming skills prior to the retreat. For an application or more information, write to:

**Logo Connections c/o; Tom Lough ;Curry School of Education; University of Virginia; 405 Emmet Street; Charlottesville, VA 22903 or call Peggy Marshall at 804-924-7471**

# Logo Ideas

## Using SETH: When, Why and How?

### by Eadie Adamson

SETH is a command which, together with a number, sets the turtle to an absolute direction. You can think of it as using compass points.

SETH 0 is *always* pointing to the top of the screen
SETH 90 *always* points directly to your right
SETH 270 *always* points directly to your left
SETH 180 *always* stands the turtle on its head

How and when should you teach it?

* * * * *

An important aspect of a good Logo environment is placing the student in control of his own learning. One of the reasons I really like *LogoWriter* is that, by putting a program on disk, the child can take over. Taking over, however, does not always bode well for success. There still may be countless problems to solve which are sometimes beyond a particular student's ability. It becomes important to provide simple tools or ideas which they can use as they encounter difficulties.

One tool which I find particularly useful to give students early in their learning is SETH, or SETHEADING (different versions of Logo allow you to use both terms, *LogoWriter* accepts only the abbreviation). SETH, or SETHEADING, changes the direction the turtle faces based on degrees. The orientation is directly related to compass points students may have already encountered in map-reading in their social studies classes: 0 corresponds to north, 90 to east, 180 to south, and 270 to west (see Figure 1).

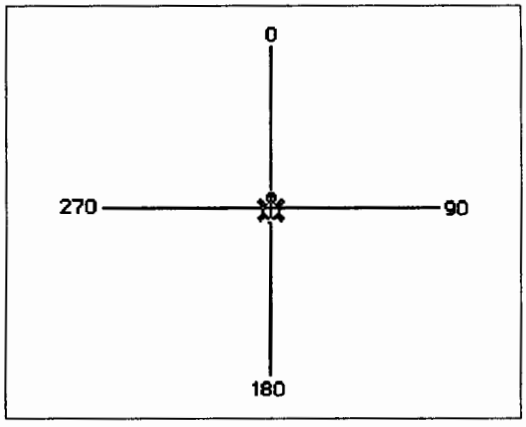



**Figure 1**

## Logo Ideas — CONTINUED

Just as these directions are orienting directions on map, they are also important anchors for turtle graphics. SETH is a convenient tool to use when the turtle is "lost," since it is an absolute direction as opposed to the relative directions of LEFT and RIGHT. For example, suppose you've been turning the turtle, perhaps have accidentally typed RIGHT 990. If you now want the turtle to face straight up, type SETH 0.

Can teaching SETH early get in the way of learning the right-left commands? My own experiences with teaching SETH have indicated that if students have these anchor points to use, they will use them when necessary and that giving them just the four compass points does not really interfere with their learning about and using RIGHT and LEFT. It may, in fact, give their confidence a boost, since they know how to get themselves untangled when necessary.

When are children ready to learn SETH? As with most new Logo ideas, I believe they are ready *when they need it*. This can mean that even a kindergarten child might learn to use SETH. Several years ago, when I was working with kindergarten children in a fairly large and active class, Damaris was in the process of drawing a house. She needed to draw a roof. I couldn't stay with her while she worked everything out, so I made her a small drawing with SETH commands and arrows showing their direction. By the end of the class Damaris finished her house! That was not the end of the story, however. During the next class a week later, I suddenly noticed Damaris teaching her neighbors how to use SETH! Damaris not only used the tools I gave her, she remembered them because they were useful to her and proceeded to share the new information with her peers. (Perhaps this means that we as teachers need to look carefully at our decisions about when and what to teach. Are we holding back on teaching something a child can really use because of our own misconceptions about what that child can understand?)

It can be helpful to put little stickers around the monitor: 0 at the top, 180 at the bottom, 270 on the left, and 90 on the right. Once SETH has been introduced, these numbers act as a good reminder when children are *turtling* about the screen. If a child has typed RIGHT 990, for example, when the intended command was RIGHT 90, a quick switch to SETH 90 will solve the problem without need for teacher intervention. For those using SETH more actively, the numbers around the monitor serve as additional visible cues, from which the intervening directions may be derived. Setting the heading of the turtle somewhere between 0 and 90 will get you started for drawing a tree, perhaps.

When using shapes rather than the turtle (with *LogoWriter*, Sprite and Atari Logo, for example), SETH is extremely important for setting direction. SETH 90 will always turn the turtle to the right, even though the shape itself will not turn. It is also far less confusing to use SETH 90 than to start a procedure with RIGHT 90, only to run the procedure again and discover that suddenly your turtle, in whatever shape, is now heading to the bottom of the screen. SETH is more reliable!

If you use the LogoWriter Activity cards in your classes, I suggest you change the ones which move shapes (there are several) so that students are directed to use SETH 90 instead of RIGHT 90. This will save you—and your students—considerable distress.

Sometimes I give my students little diagrams which they keep in their notebooks. The diagram (see Figure 2) gives a number of useful directions which can be used with SETH (or with RIGHT if you place the diagram so that 0 is aligned with the current direction of the turtle). This is similar to many "Turtle-Turner" devices which are on the market, but the same little diagram can be used to produce transparencies which a child can place on the screen to help determine direction.

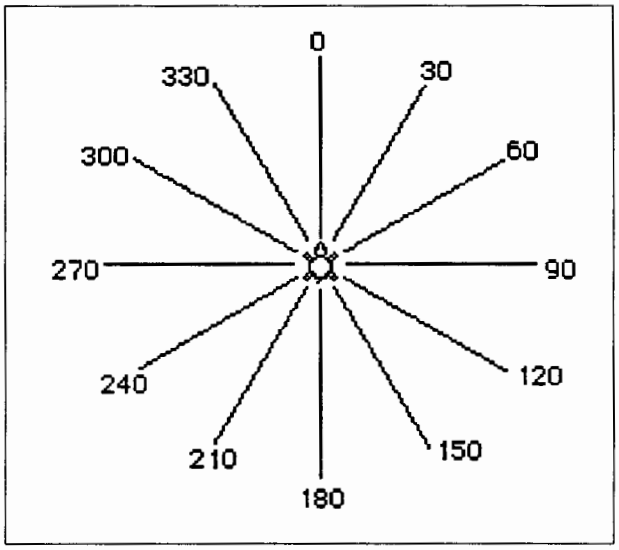



**Figure 2**

The diagram is a nice idea, but a better idea might be to help your students make their own. If you use *LogoWriter*, help them to construct one which has at least the four basic points of 0, 90, 180 and 270. Let them work out the others. You might start off with:

```
SETH 0
FORWARD 50
BACK 100
FORWARD 50
```

Finish the diagram, then LABEL the points. Challenge your students to make a diagram of their own which includes some intervening points as well. Let them print their own diagrams! Not only do they have a tool they can use, but they have

**Logo Ideas — CONTINUED**

ownership of the tool because they created it themselves. You can make transparencies for them from their own diagrams, too!

Another way I have introduced SETH is to ask my students to pretend the turtle can only respond to FORWARD, BACK and SETH—no RIGHT or LEFT allowed. Then I give them a few simple challenge designs to try using only these three commands (see Figure 3).

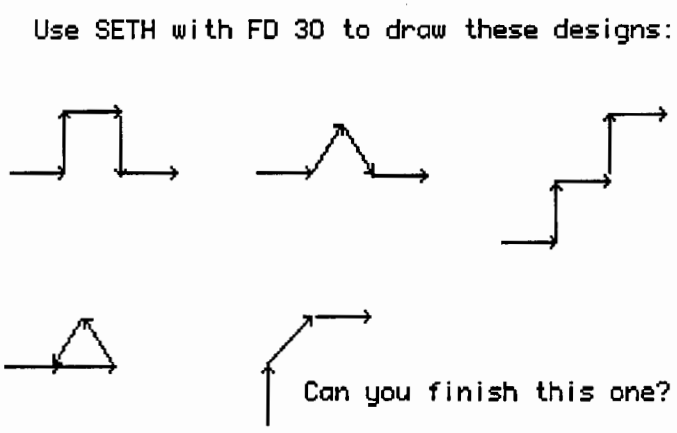



**Figure 3**

Some students may think of programming the four headings by their compass names. This is presents a good opportunity to teach about OUTPUT as well. Rather than programming EAST to be SETH 90, introduce OUTPUT so that EAST will OUTPUT 90.

```
TO EAST
 OUTPUT 90
END
```

Then SETH EAST becomes the command to use (and EAST can also be an input to LEFT or RIGHT if the student wants to use it that way!). Not only does this make a more direct connection with compass points learned elsewhere, but it will help the student remain aware of what is actually happening when EAST is used, namely setting the heading of the turtle. Since SETH is a primitive, I think it's important not to allow children to lose sight of it by burying it in a procedure. Encourage them to use it instead!

How to explain OUTPUT? One simple approach which I have used with my fourth grade classes is to hand someone a disk jacket. Since OUTPUT passes something on to be used in another command or procedure, ask the student to OUTPUT the jacket to his neighbor. Make it a simple demonstration in which each OUTPUTS to the next until it gets back to you, when you—finally!—*use it* to store a disk. The analogy may not be perfect, but it gets across the idea of passing something on, which is precisely what OUTPUT does. Then suggest writing the procedures to OUTPUT the heading. Perhaps someone will think of OUTPUT as useful for shapes and colors too. For example:

```
TO BLUE
 OUTPUT 5
END

TO CAR
 OUTPUT 26
END
```

allows you to type SETC BLUE and SETSH CAR. SETH EAST, after EAST has been written, will set the car in the correct direction for moving across the screen.

If your students are already using inputs, pose this problem:

1. Write a square procedure with inputs. Use only SETH to turn.
2. What can you do with inputs so that you can use SQUARE to do a SPINSQUARES?

One might think this is an insoluble problem, but it can be done by adding an input, INCREMENT, to SETH: SETH 90 + :INCREMENT. Don't forget to add it to the name of the SQUARE and SPINSQUARE procedures as well.

If you want to try this, it might look something like this:

```
TO SQUARE :SIZE :INCREMENT
 SETH 0  + :INCREMENT
 FD :SIZE SETH 90 + :INCREMENT
 FD :SIZE SETH 180 + :INCREMENT
 FD :SIZE SETH 270 + :INCREMENT
 FD :SIZE
END

TO SPINSQUARES :SIZE :INCREMENT :ANGLE
 SQUARE :SIZE :INCREMENT
 SPINSQUARES :SIZE (:INCREMENT + :ANGLE)
      :ANGLE
END
```

SQUARE 50 0 will simply draw a square; SQUARE 50 45 will draw a square turned on its tip; SPINSQUARES 50 45 45 will spin the squares! (The increment and the size inputs are passed to SQUARE; ANGLE is the turn for SPINSQUARES.) Now, how about adding a STOP rule for SPINSQUARES?

*Eadie Adamson*
*Allen-Stevenson School*
*132 E. 78th St.*
*New York, NY 10021*

# Powerful Ideas in Problem Solving and Logo

by Dave Moursund and
Sharon Burrowes Yoder

## Introduction

The creators of Logo had a vision of young children becoming better problem solvers through immersion in a Logo environment. A large number of researchers into Logo and other computer programming environments have diligently searched for improvements in the overall problem solving skills of students that could be attributed to programming experiences. Such researchers are seeking evidence of **transfer** of problem-solving skills from the computer programming environment to other domains (most specifically, to non-computer domains).

For the most part, these researchers have been disappointed. Most of the research studies suggest that the problem-solving transfer that occurs from students learning to program, be it in Logo, BASIC, or Pascal, is modest at best. Most studies report no significant differences.

Researchers into problem solving would have predicted this lack of transfer. The subject of problem solving and transfer has been studied for many years and there is a substantial body of knowledge in this area. (Dewey, 1910; Fredericksen, 1984; Minsky, 1986; Moursund, 1987; Polya, 1945). Quite a bit is known about when transfer will occur and how to increase transfer. We even know quite a bit about how to teach for transfer.

Logo is an excellent vehicle for creating a rich problem-solving environment. But the Logo environment, all by itself, is not sufficient to produce the desired increases in general problem-solving skills that we would like to have occur. If transfer of problem-solving skills and knowledge is a major goal when having students use Logo, there are many things the teacher can do to enhance transfer. This article is the first in a series that will address how to increase transfer of problem-solving knowledge and skills from a Logo environment.

## Transfer of Learning

Learning theorists talk about *near transfer* and *far transfer*. For example, suppose that I learn to tie my left shoe, and I am practicing with brown shoes that have nylon shoe laces. Chances are that the knowledge and skills I gain will transfer to tying the right shoe of a pair of black shoes that have cotton shoe laces. This is near transfer. It is something that most human minds do quite well, quite automatically. The human mind is "wired" to facilitate near transfer.

On the other hand, mastering shoe tying does not readily transfer to tying a bow tie. That is a far transfer. Of course, the terms *near* and *far* are relative. Most of us are familiar with the case of the student who studies and masters metric measurement in a math class, walks down the hall to the next period science class, and claims complete ignorance of any metric knowledge in the science class. Many students find that the transfer from a math class environment to a science class environment is quite difficult—much to the exasperation of teachers. In summary, we know that far transfer does not easily or automatically occur but appropriate practice and instruction can help it. We also know that a number of measures of intelligence are, in essence, measures of transfer. Less gifted students have more trouble making transfers of their problem-solving skills from one environment or domain of discourse into another.

We know how to increase far transfer. Two techniques that will be stressed in this series of articles are:

1. Help students to have a very explicit understanding of the problem solving skills and techniques that they are learning and practicing.

2. Help students to find examples from a wide variety of disciplines where these same problem-solving skills and techniques are applicable. Facilitate students gaining practice in applying their new problem-solving skills and techniques in these disciplines.

## Some Things We Know About Problem Solving

We know quite a bit about problem solving. In this section we list a few of the known facts. The reader should be aware that these facts are not 100% guaranteed to hold in all instances. However, there is sufficient research underlying the basic ideas to strongly support curriculum design and instruction based on the ideas. Some examples:

1. Problem solvers who talk about the steps they are taking to solve a problem do better than those who do not describe their work.

   Talking about the problem-solving steps one is taking or contemplating seems to help increase understanding of how these steps relate to each other and to their intended outcome. All teachers recognize that they have learned a great deal through their attempts to explain things to students.

2. How we think about or represent a problem is a better indicator of a problem's difficulty than any quality

intrinsic to the logic of the problem.

Two people, with essentially equivalent backgrounds and experiences, may view a particular problem in entirely different ways. For one the problem may be trivial, while for the other it may seem impossible. This suggests that we should give specific instruction and practice in viewing (thinking about, representing) the same problem in a number of different ways. Much of the work of de Bono (1970) focuses on this idea.

3. Even with well defined problems, people tend to frame small subgoals and may not be able to explain why they did so.

   This is essentially a statement that solving hard problems is hard. It is easy for the human brain to essentially be overwhelmed by a problem. In the absence of careful training as to ways one might proceed in this case, most people tend to select some subgoal and work on it—often with little understanding of why they are selecting the subgoal or how success on it relates to solving the original problem. This suggests that we should give specific instruction to students on ways to get started when a problem seems overwhelming.

4. The typical person has a few basic general problem-solving strategies which they use for dealing with a variety of problem situations.

   This fact suggests that adding even a few problem-solving techniques to students' repetoire may contribute substantially to their ability to solve problems.

5. Precise thinking (processing) is one of the keys to strong problem-solving ability.

   Precise thinking can be emphasized in every discipline. Some disciplines, such as computer programming, tend to place greater emphasis on precise thinking than do other disciplines. Precise thinking and precise representation of one's thinking are, of course, closely related.

6. Experts outside of their domain of expertise do no better than novices. Good problem-solving ability in one area does not automatically carry over to problems in another area.

   Research into problem solving often focuses on *domain-free* and *domain-specific* knowledge. Expertise in a particular field requires a huge amount of domain-specific knowledge. We cannot expect to make students into expert problem solvers in all domains merely by helping them to learn general-purpose problem-solving techniques in a particular domain (such as in computer programming). However, the value of increasing a student's domain-free problem-solving knowledge and skills is self evident.

**Powerful Ideas**

Careful analysis of the literature on problem solving and the above ideas has led to the identification of a number of *powerful ideas* Here are some powerful ideas that occur in Logo and in a number of other problem-solving domains:

1. Break big problems into a collection of smaller problems.

2. Build on the previous work of yourself and others.

3. Verbalize when problem solving. (For example, it is often quite helpful to explain to someone else what it is that you are doing.)

4. Look for patterns and make use of repetition.

5. Draw a picture, a diagram, a graph, or some other picture-like representation of the problem.

6. One's procedures for solving problems often contain bugs. Learn how to detect and correct bugs.

7. When solving a complex problem by breaking it into smaller problems, it is important that steps used to solve one of the smaller problems not change the other problems or affect steps one is using to solve them. (In Logo, this is closely related to the idea of State Transparency —returning the turtle to its original location and heading at the end of a procedure.)

**Criteria for Powerful Ideas to be Included in this Series of Articles**

In this series of articles we will examine and illustrate a number of powerful ideas. Ideas to be illustrated must meet three criteria:

1. The idea must be powerful (strongly applicable) in a number of problem-solving domains. However, it need not be universally applicable or equally powerful in all domains.

**Powerful Ideas in Problem Solving and Logo — CONTINUED**

2. The idea must occur naturally in Logo and be easily illustrated, even to quite young children, in the Logo environment.

3. The idea must be easily illustrated in a number of other problem-solving domains, at a variety of grade levels and in a variety of subject matter areas.

**Reader Input Requested**

We intend to write a substantial series of articles for the *Logo Exchange*, each focusing on one or more powerful ideas. Reader input is welcome. What are your favorite powerful ideas on problem solving? How do you illustrate them to your students? Please send us some examples.

**References**

de Bono, E. (1970). *Lateral Thinking: Creativity Step by Step.* New York: Harper and Row Publishers.

Dewey, J. (1910). *How We Think.* Boston: D. D. Heath.

Fredericksen, N. (1984). Implications of Cognitive Theory for Instructional Problem Solving. *Review of Educational Research 54*: 363-407.

Minsky (1986). *The Society of Mind.* New York: Simon and Schuster.

Moursund, D. (1987). *Roles of Computers in Problem Solving: An Independent Study Course.* Eugene, OR: International Council for Computers in Education.

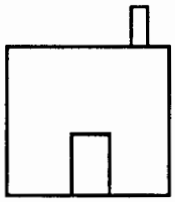
Polya, G. (1945). *How to Solve It.* Princeton, NJ: University Press.

## Breaking Big Problems in Small Pieces

A common thread running through the Logo literature is the idea of using procedures and subprocedures. Examples abound showing teachers and students alike how to write procedures and to use them to break drawings into component parts. How many of you have seen an example showing how to write a SQUARE procedures and then use that procedure to write a SPIN.SQUARE procedure? Or, perhaps you've encountered the more common example, showing how to use SQUARE and TRIANGLE procedures to draw a house?

These classic examples seldom go beyond breaking drawings into component polygons in a rather obvious manner. Further, little is made of the need to assist students in looking for ways to subdivide their projects into meaningful chunks. Too often it seems that the teacher simply points out that the house is a triangle and a square and doesn't encourage the student to see it differently—for example, as a *u* shape with a triangle on the top.

Below is a drawing of a not-so-typical house that can easily be done in Logo. Before you read further, take a few moments to think about dividing this simple picture into components. Try to take your thinking out of the typical Logo context. Think about drawing the picture with pencil and paper. Does that give you new insights? Think about the actual programming process. Do you see patterns in what is needed to produce this drawing?

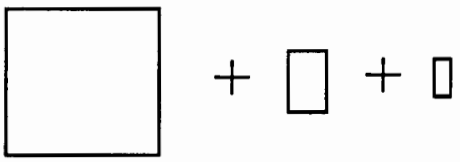


**Solution #1:**

To solve this problem, you might simply use the turtle to trace the outline of the figure. A key idea here is that you will need to go over some lines more than once (e.g., the bottom of the door and chimney). You might trace the outline of the house clockwise or counter clockwise or you might go around the door and or chimney clockwise or clockwise. If you think about it for a minute, you can see that there are quite a few different solutions that will produce this drawing using only a series of FORWARD's, BACK's, RIGHT's and LEFT's.

**Solution #2:**

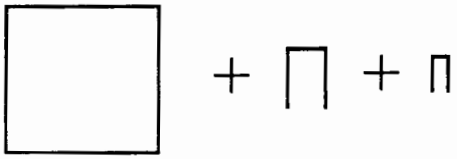
Here the house is viewed as a square with two added rectangles:



**Solution #3:**

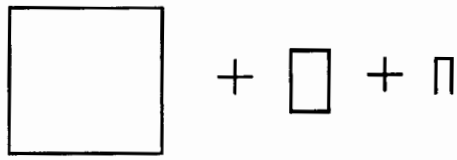
Here the house is viewed as a square with two upside down U-shapes:

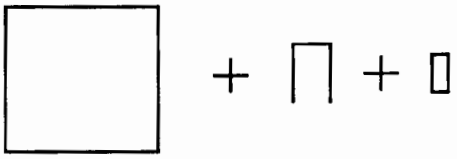
## Powerful Ideas in Problem Solving and Logo — CONTINUED



**Solution #4:**

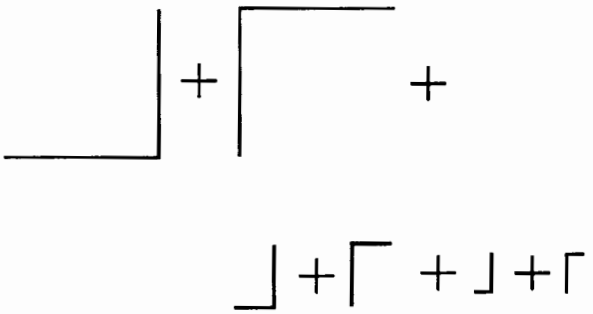
Here we see two combinations of #2 and #3:





**Solution #5:**

Yet another approach is to view the entire picture as made up of L's. Only one solution using this idea is given below. There are a number of possible combinations of such L's that might be used to create the house.



Did you think of some of the solutions given above? Did you see still other solutions? Were they quite similar to those shown above or were they quite different?

In writing a computer program to solve a problem such as drawing this house, there are two possiblities. First, if you already know how to accomplish the task, you simply do it. That is, if you are given a picture of a square to draw, and you *know* that REPEAT 4 [FORWARD 50 RIGHT 90] draws a square, no further thought is required. However, if the necessary tools are not at your fingertips, then you need to break the problem into parts. In the final analysis, the parts must meet two criteria:

1. Each piece is a problem that you can solve.

2. Doing all the pieces solves the overall problem.

So, in order to draw the house, you needed to break it into pieces (since it is unlikely that you have in mind exactly how to draw a house like the one shown.) If you think about drawing this house *in Logo* then you will likely choose a breakdown that meshes with what you know how to do in Logo. If you are comfortable drawing rectangles, then the square + two rectangle decomposition is the one you are likely to choose to implement. If drawing L shapes has been part of a recent project, then you might choose to decompose the house into a shape for which you have relevant Logo code "at hand."

There are a number of ways that you can practice this idea of decomposing drawings into subparts with your students. You might present simple drawings or designs and discuss with your class different ways to decompose them. You might divide your class into small groups and have each group discuss different approaches and then share their work with the class. Yet another approach is to use student sketches of their project ideas. You can discuss possible decompositions with them individually, in small groups or with the class as a whole. The important idea here is to provide a variety of practice for your students and to make quite explicit that there are many different ways to solve any particular problem. Be sure that you don't discount a student's solution to such a problem because it is not the way you would solve it. Let students discuss the advantages and disadvantages of their solutions, both in and out of the context of using Logo.

The house example discussed above illustrates one of the fundamental issues in teaching problem solving. One can help the student to acquire more and more building blocks—specific problems that they know how to solve. The student will be able to solve more and more problems by merely recognizing that a new problem is one that has been previously encountered and mastered. Or, one can give students lots of practice in breaking big problems into small problems, with the goal of breaking a problem into small problems that one knows how to solve.

## Powerful Ideas in Problem Solving and Logo — CONTINUED

How much instructional time and learning effort should be spent on the former task, and how much on the latter? Does an appropriate division of instructional and learning effort vary widely with the student? If so, can we individualize instruction to accomodate these major differences in learning style, and help the student to learn how to learn?

**Transfer Examples**

As we indicated in the introductory section of this article, teaching a problem solving concept using Logo is only one step in the process of helping students to become better problem solvers. It is necessary to actively teach for transfer of newly learned knowledge and skills in problem solving.

Sometimes the computer teacher is a specialist and does not teach other subjects to the students. Sometimes the classroom teacher is also the Logo teacher. In either case, there are a number of ways to help students to transfer a newly learned technique to other disciplines.

The Logo teacher can point out ways in which an idea learned in Logo applies to another subject. The students can be asked to give examples from other disciplines. Work can be done in large or small groups or individually.

If the person teaching Logo does not teach other subjects, it is especially important that he or she communicate with subject-area teachers the topic being studied so that they can incorporate the new ideas into their teaching. It will be most effective for the learning of students if the connections with Logo are made spontaneously as part of the teaching of another subject. For example, in a math class, the teacher might be discussing how to find the area of a complicated shape by breaking it into subparts. The teacher could include in the discussion that this technique is "like breaking problems into parts using procedures and subprocedures in Logo," and then ask students for some examples. The subject area teacher need not be a Logo expert to assist students in making these connections.

It is very important that students discover for themselves examples where their Logo problem-solving ideas are applicable in other disciplines. The role of the Logo teacher and other teachers is to facilitate this discovery process. The teacher may find it useful to have quite a variety of examples in mind when undertaking this facilitation process. Here are some examples where breaking big problems into smaller problems may be useful.

1. Getting dressed to go to school. This problem can be subdivided in a number of ways. The first consideration might be the weather or the season. Once that is determined, then such matters as color, style, and what matches what need to be examined.

2. Running a series of errands. Presumably the person running the errands knows how to complete each errand on the list. Here the process of breaking into subparts might include grouping the errands by location, ordering them by importance that they be completed, or perhaps grouping them by when would be the best time to do each one.

3. Doing homework. Homework might consist of straight-forward tasks such as "do problems 1-10 on page 23" and more complex tasks such as "write a book report." The student needs to subdivide the larger tasks into "doable" subtasks and then organize the available time in order to complete that work that is due the next day.

4. Cleaning your room. For many a youngster, this is a formidable task, especially as the teen years approach. Subdividing the job into such tasks as clean up the clothes, take care of the records and tapes, organize the books, etc. may make it more manageable. However, if the room is simply too awful, then more subdivision may be necessary: put the dirty clothes in the laundry, put the clean clothes in the drawer, hang up the clothes on the closet floor, and give to your younger brother the outgrown clothes!

5. Cooking a meal. Preparing a meal of more than one dish requires planning. First, the steps to prepare each item must be examined. Then the overall plan for timing the preparation of the meal must be made. The preparation of each item requires a number of steps which might include finding the recipe, getting the ingredients, combining the ingredients in the proper order, and cooking for the required period of time.

6. Learning a new skil! in physical education. Often learning to play a new game requires putting together pieces of kinesthetic knowledge into a new "move." For example, most children know how to bounce a ball, but dribbling a basketball requires learning the component parts and putting them together so that the ball can be moved down the court.

7. Decoding a word. When students encounter a new word, they are often told to sound it out. This amounts to breaking the word into subparts that they know.

## Powerful Ideas ... — CONTINUED

8. Writing a story. Writing can seem to be an overwhelming task to most students. Drawing a parallel between writing a Logo program and writing a story may relieve some anxieties. They can learn to see that creating the beginning, middle, and end of a story is not unlike creating the parts of a picture in their computer class.

9. Learning a new song. Whether memorizing a song to sing or learning to play a part on an instrument, few students try to tackle the whole job at once. Music is naturally procedural, with phrases bearing a close resemblance to individual Logo procedures. (In fact, if your version of Logo includes music, the writing of music is an excellent way to make using small procedures very natural.)

10. Searching for information in a library. Locating the needed sources in a library can be an overwhelming task without some subdivision. Note that this task requires that you understand the subdivisions of the library, e.g., periodicals, books, microfilm etc., as well as an understanding of how to locate the information within a subidivsion, e.g., card catalogue, computer terminal, searching through stacks, etc.

This list is, of course, by no means exhaustive. No doubt you (and your students) can—and should—think of many more. You might even want to keep a list of new ideas on a bulletin board in your classroom when you are specifically discussing procedures and subprocedures and breaking problems into parts.

*David Moursund and Sharon Burrowes Yoder*
*ICCE*
*University of Oregon*
*1787 Agate St.*
*Eugene, OR 97403-9905*

**LogoWriter Sound Contest**

Walker Junior High School gives a first prize of $50 to the school that sends the best sound procedure. Procedures must be original and use TONE. Entries must include the name of school, grade level, teacher, procedure, and name of sound simulated. Entries will be judged on how closely they resemble the chosen sound.

Send entries to:
Walker Computer Club; c/o Edwina Walsh; Walker Junior High School; 8132 Walker St.; La Palma, CA 90623-2097.

# Teaching Tools

## "Programming with Style"

**by**
**Glen Bull and Gina Bull**

Students in computer science are responsible for meeting coding standards. Part of their grade may be based on how well their programs meet these standards. Why programming standards? Procedures written by professional programmers are frequently reused in other programs. If a procedure can be reused, it is possible to amortize the investment of effort across several projects. Therefore in computer science a great deal of attention is devoted to construction of procedures which are well written.

As you may have guessed, this is the concept which underlies a software tool. The title of this column, "Teaching Tools," stems from this idea. There are other reasons for creating well-written programs. A well-written program is less likely to contain errors. If there are errors in the program, properly constructed procedures are easier to debug.

If a procedure is well written, it may be easier to see patterns in the program. This encourages the "aha!" phenomenon — instances in which the individual suddenly sees a relationship. This is a particularly important part of problem-solving. Well-constructed procedures are easier to modify and change. A well-written program also is easier for someone besides the programmer to understand. This makes it possible for other people to use your procedures. An entire branch of computer science, called "software engineering," is devoted to the study of how to write good programs.

You do not need a degree in computer science to write good programs. A few common-sense ideas will help ensure that your programs will run the first time, every time ... Well, probably not every time. However, the percentage of procedures that do run right away will be higher.

You may be asking whether "software rules" will stifle creativity and exploration? Perhaps this is why very little is found in the Logo literature about ways of writing good programs. Good programming techniques will not interfere with exploration and tinkering. In fact, they will increase the chances of interesting discoveries. Here are four strategies for constructing good Logo programs.

### Procedure Length

The single most important strategy is to write lots of short procedures, rather than a single long procedure. Often people

## Teaching Tools — CONTINUED

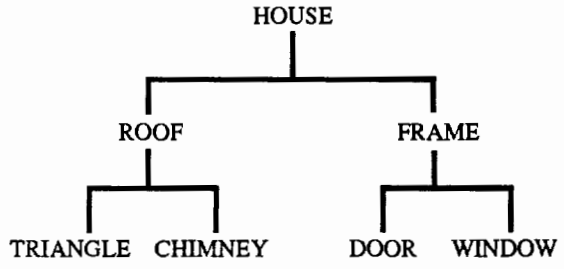
who first learn to program in BASIC write long procedures in Logo too. However, in BASIC it is not possible to create independent procedures—subroutines are the closest equivalent. Logo makes it possible to break a program up into many small parts—using short procedures takes advantage of the strength of the language.

How short should a procedure be? One rule of thumb is that if a procedure is longer than a screen (25 lines), it is too long. Another rule of thumb is that code which is used in more than one place should be written as a subprocedure. For example, if you are creating a house, and have several windows, it is probably a good idea to create a general-purpose WINDOW procedure.

Why are short procedures important? The most important reason is that it is easier to spot mistakes if procedures are short. If a procedure is only seven lines long, there are not many places where the error can be. Of course, it is important to test and debug each subprocedure before using it in a larger procedure.

Using short procedures also encourages good problem-solving techniques. Papert refers to the process of breaking ideas into "mind-sized" chunks. Logo provides an excellent laboratory for experimenting with problem-solving strategies, because it is immediately possible to determine whether a solution works or not. However, providing a child with Logo does not automatically confer these benefits. It is necessary for the teacher to point the way, and suggest effective techniques for overcoming problems. Using short procedures also makes it more likely that some of the procedures will be usable in other programs.

Sometimes it is helpful to construct a procedure tree showing the relationship of the procedures to one another. For example, here is the procedure tree for a house. (Our son asked for a tree house, but our closest approximation was a HOUSE tree!)




The procedure tree makes a useful debugging tool. This procedure tree shows that the HOUSE procedure is three levels deep. At the first level, HOUSE is the master procedure. It is possible to see that HOUSE consists of only two subprocedures: ROOF and FRAME.

```
TO HOUSE
 ROOF
 FRAME
END
```

Of course, it is not necessary to create a procedure tree for a program as short as HOUSE. However, as a program becomes more and more complex, aids such as this make it easier to see how a program is structured.

**Use Meaningful Names**

Use of meaningful names is the second most important strategy that you can adopt to make better programs. As Papert notes in *Mindstorms*, early versions of BASIC only permitted names which were one or two letters long. As a consequence, people who learn BASIC before using Logo sometimes have a tendency to use names such as X1 and X2. Logo permits a name to be any length. Use of meaningful names takes advantage of the strength of the language. (Some of the more modern versions of BASIC now permit meaningful names as well.)

Of course, it is possible to use any name you want. The computer doesn't care. If you like, you can call a procedure that draws a house RUTABAGA. However, your procedure will be easier to understand if meaningful names are used. Compare the following three procedures that draw the frame of a house.

```
TO FRAME
 WINDOW
 DOOR
 DOORKNOB
END

TO A
 B1
 B2
 B3
END

TO RUTABAGA
 ARDVAARK
 APRICOT
 ENDICOT
END
```

Even though all three procedures do the same thing, the first one is much easier to understand. The second procedure uses numbers and letters instead of names. The third procedure uses names which are not related to the purpose of the subprocedures. Therefore it is difficult to follow what the program does. In a

simple program like HOUSE meaningful names are not as important. As programs grow more complex, it is easier to understand how they function if meaningful names are used.

Meaningful names should be used for procedure names as well as input and variable names. Consider the following Logo procedure that draws an arc.

```
TO ARC :DEGREES :DIAMETER
 REPEAT :DEGREES / 10 [RIGHT 5
      FORWARD :DIAMETER * PI / 36 RIGHT 5]
END

TO ARC2 :X :Y
 REPEAT :X / 10 [RIGHT 5
      FORWARD :Y * PI / 36 RIGHT 5]
END

TO PI
 OUTPUT 3.1416
END
```

In the first procedure, ARC, it is immediately apparent that the first input, :DEGREES, represents the number of degrees in the arc, and that the second input, :DIAMETER, represents its diameter. In the second procedure, ARC2, the user must examine the procedure to determine that :X represents degrees and :Y represents diameter. In this short program, it is relatively easy to identify what the two inputs represent. As programs become more complex, it is more difficult to decipher the purpose of variables.

Use of meaningful procedure names and variable names helps make code self-documenting. It is not necessary to have a separate document which explains that :X is degrees and :Y is diameter if the actual names :DEGREES and :DIAMETER are used instead. This makes it easier for the programmer to follow the logic of the program as a procedure is developed. It also makes it easier for others to understand what inputs to use when they run the program or modify it.

### Formatting

Formatting is another area which is typically considered in the construction of programs. In programming languages such as Pascal, attention is usually given to indentation of lines and the physical appearance of procedures. Formatting guidelines of this kind are intended to improve the readability of the code.

Most versions of Logo do not permit indentation, and therefore this is a moot point. However, there are a couple of other factors which can be considered. It is a good idea to group similar commands so that patterns in the program are visible. Consider these two procedures.

```
TO SQUARE1
 FORWARD 50 RIGHT 90 FORWARD 50
 RIGHT 90 FORWARD 50 RIGHT 90 FORWARD 50
 RIGHT 90
END

TO SQUARE2
 FORWARD 50 RIGHT 90
 FORWARD 50 RIGHT 90
 FORWARD 50 RIGHT 90
 FORWARD 50 RIGHT 90
END
```

In SQUARE2 it is evident that there is a pattern in the procedure. It is also possible to discern this in SQUARE1, but it is not as apparent. The pattern leaps out in SQUARE2.

It is also a good idea to limit the length of a line to no more than a couple of commands. After more than one or two commands are placed on a line, the procedure becomes difficult to read.

### State Transparency

If a software tool will be used in many different programs, it should ideally leave the programming environment in the same condition in which it found it. This idea is known as *state transparency*. Here are couple of examples.

A programming tool which we frequently use is the procedure OVER. OVER just moves the turtle over.

```
TO OVER :DISTANCE
 PU
 RIGHT 90
 FORWARD :DISTANCE
 LEFT 90
 PD
END
```

OVER is useful, because it substitutes a single command in place of five. However, the version of OVER shown above could be improved. The difficulty is that OVER puts the turtle's pen down at the end of the procedure. A better version of OVER would record the condition of the turtle's pen at the beginning of the procedure, and return the turtle's pen to the same condition at the end of the procedure.