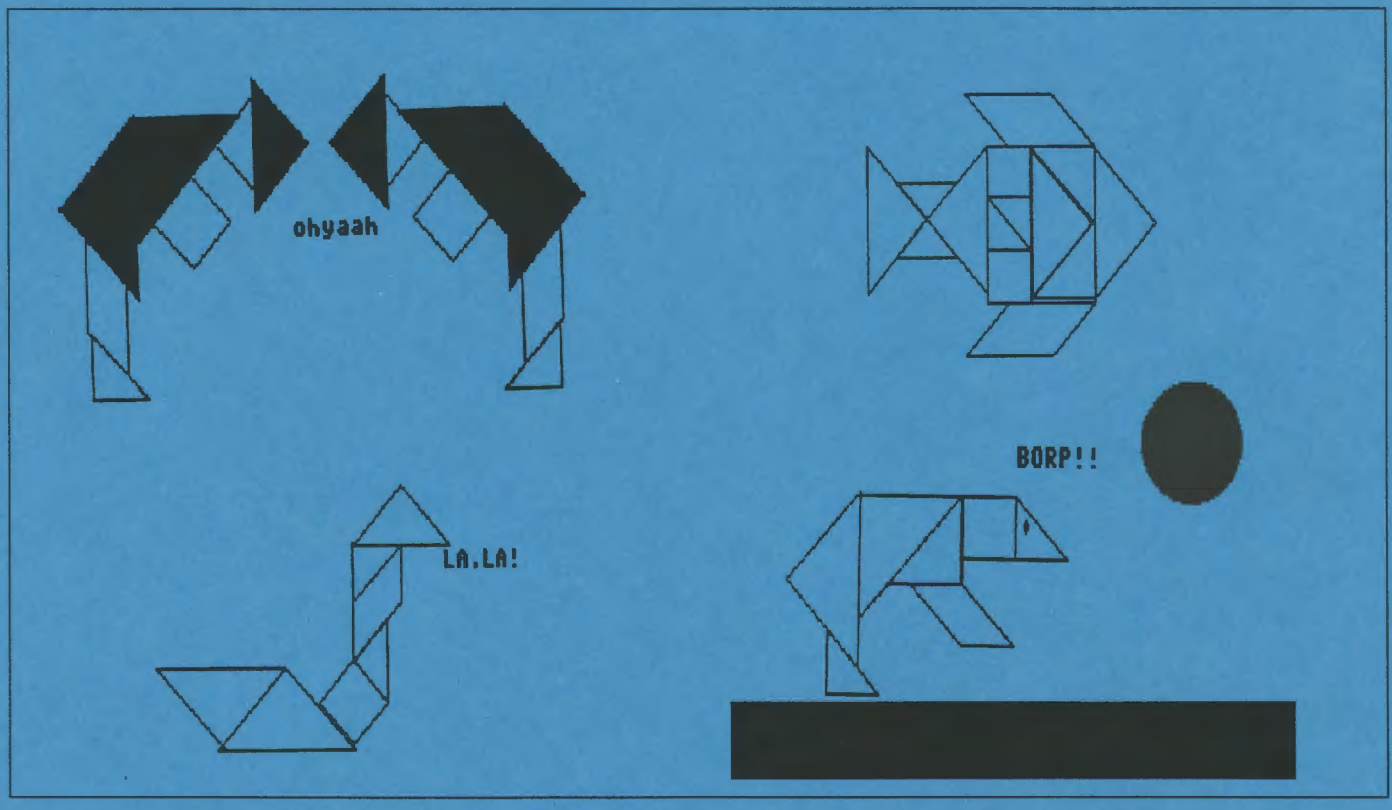
Journal of the ISTE Special Interest Group for Logo-Using Educators

LOGO EXCHANGE

March 1990

Volume 8 Number 7



International Society for Technology in Education

ISTE Publications

ef•fec•tive\i-'fek-tiv\adj (14c)

1 a : producing a decided, decisive, or desired effect b : IMPRESSIVE, STRIKING
2 : ready for service or action

Computer-Integrated Instruction: Effective Inservice

Dave Moursund's comprehensive series on inservice training for computer using educators has grown. ***Effective Inservice for Secondary School Mathematics Teachers*** and ***Elementary School Teachers*** are joined by texts for ***Secondary School Science Teachers*** and ***Secondary School Social Studies Teachers.***

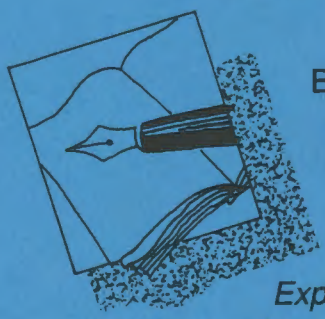
Based on a National Science Foundation project, these volumes bring you the latest research on effective training. Each work contains specific activities and background readings that enable you to hold inservices that result in positive, durable change at the classroom level.

If you design or run computer-oriented inservices, ***Effective Inservice for Integrating Computer-As-Tool into the Curriculum*** will help you develop a sound program through theory and practice. Sample forms for needs assessment and formative and summative evaluations are included.

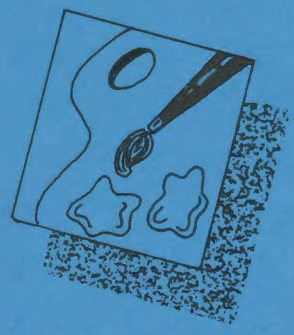
Each of the five volumes comes in a three ring binder that includes both hard copy and a Macintosh disk of the printed materials. Individual Math, Science, Social Studies, and Elementary School volumes are $40 each ($3.70 shipping per copy). Computer-As-Tool is $25 ($3.70 shipping per copy). The complete set of five is available for the discounted price of $150 ($7.50 shipping per set).

ISTE, University of Oregon, 1787 Agate St., Eugene, OR 97403-9905; ph. 503/346-4414.

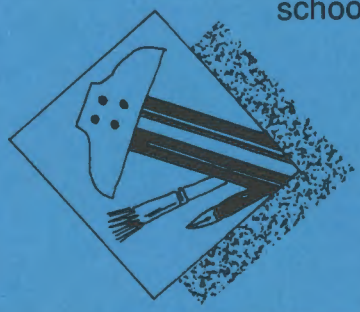
Make your desktop publishing software earn its keep.



By now you have discovered that there is more to desktop publishing than mastering the keystrokes and commands of the software. *Exploring Graphic Design* teaches you how to plan and produce letterhead, posters, newsletters, manuals and books.



Exploring Graphic Design is a concise and thorough overview of essential design principles and their application to practical problems. It complements any desktop publishing program. Perfect for secondary school or university classes, or as a helpful reference for adults.



Get your money's worth from your desktop publishing software by *Exploring Graphic Design.*
$9.95 plus $2.65 shipping

ISTE, 1787 Agate St., Eugene, OR 97403; ph. 503/346-4414.

ISTE

LOGO EXCHANGE

Volume 8 Number 7 Journal of the ISTE Special Interest Group for Logo-Using Educators March 1990

Logo Exchange is the journal of the International Society for Technology in Education Special Interest Group for Logo-using Educators (SIGLogo), published monthly September through May by ISTE, University of Oregon, 1787 Agate Street, Eugene, OR 97403-9905, USA; 503/346-4414.

POSTMASTER: Send address changes to Logo Exchange, U of O, 1787 Agate St., Eugene, OR 97403. Second-class postage paid at Eugene OR. USPS #000-554.

Contents

ISTE Membership

	U.S.	Non-U.S.
	28.50	36.00

SIGLogo Membership (includes *The Logo Exchange*)

	U.S.	Non-U.S.
ISTE Member Price	25.00	30.00
Non-ISTE Member Price	30.00	35.00

Send membership dues to ISTE. Add $2.50 for processing if payment does not accompany your dues. VISA and Mastercard accepted. Add $18.00 for airmail shipping.

From the Editor

Are You a "Language Chauvinist?"

This quarter I'm teaching a new course here at the University of Oregon called "Programming Languages for Educators." This course is designed to expose graduate students in computer education to important computer science concepts as well as to help them expand their knowledge of programming in at least two different programming languages. Because this is an education class, we will be discussing the issues surrounding the teaching and learning of computer programming while we ourselves are engaged in the process of teaching and learning.

I have been preparing for this course by rereading Bruce MacLennan's excellent book *Principles of Programming Languages* (Holt, Rinehart, and Winston). This book again struck me with the importance of viewing programming languages in the context of their history. While this book is indeed a computer science book aimed at teaching readers how to *develop* a programming language, it can easily be viewed at another level, providing the reader a deep understanding of why each major programming language or generation of programming languages has the characteristics that it does. When I first read the book some years ago I found that it put into perspective much of what I already knew about programming in a way that made me much more tolerant of the diversity of programming environments. As I reread it, I again find myself struck by the importance of understanding historical context, even in a field as new as computer science.

In the Logo community it is quite common to hear extremely disparaging remarks about other programming languages, especially BASIC. Those of us in love with Logo can't seem to imagine how anyone would want to program in any other language. If you talk to those outside of the Logo community who have different "favorite" programming languages, you'll find that they too think that *their* programming language is the only one to use, be it Pascal, C++, or Prolog.

Historically, however, each language or generation of languages has had its role to play. FORTRAN brought us away from assembly language programming and allowed us to use fairly standard mathematical notation. Algol laid the groundwork for most significant future programming languages. Pascal brought us simplicity and a move towards the needs of the applications programmer. Some of the early languages have been extremely successful; others have merely provided the foundation for future evolution. But each has made its contribution. It should be particularly interesting to *LX* readers that MacLennan's only mention of Logo is a reference to its influence on Alan Kay's development of Smalltalk. In this context, Logo is primarily viewed as having a small place in computer science history!

But what does all of this computer science "stuff" have to do with those of you using Logo in the classroom?

First and foremost, I think that it should remind us not to be "language chauvinists"; not to think that "our" language is the only language – or even that our version of Logo is the only one. Even your *LX* editor uses a variety of programming languages. I write programs in BASIC, Pascal, DBASE, Logo, and Hypertalk. Each has its place; each has its reason. Logo is not *always* the best tool for my needs.

More importantly, perhaps, let's be careful not to get stuck insisting that Logo not grow and change. Among some Logophiles there was a great deal of resistance to the LogoWriter interface when it was first introduced. Somehow it wasn't "real" Logo. I've heard quite a few people complain about some of the new features in LogoWriter and Logo PLUS. These detractors seem to think that Logo should remain pure to its traditional form. A proliferation of such attitudes will soon leave Logo as nothing more than a mention in the history of computer education. Educators will move on to more flexible and powerful tools that make use of the capabilities of new hardware and the needs of a changing population of computer using students.

And what of other software besides programming languages? In this issue we again have an article by Glen and Gina Bull on using *HyperCard*. Why? Because Glen and Gina (and many others) see Logo as part of a larger class of software that is "learner based"; software that puts the learner rather than the computer in control. No doubt you have noticed more articles along these lines in *LX*. These articles can help remind us all that it is generally the Logo environment that most of us value. For most of us, Logo is clearly more than just a programming language.

So where will the future take those of us who use Logo today? Hopefully towards better and better "learner based" environments; environments that don't control the student; environments that allow all users to make the computer a powerful tool in their lives. Let's hope that the developers of Logo versions continue to provide us with increasingly sophisticated versions of Logo that can compete successfully with the other wonderful software tools that are available for today's newest hardware.

Sharon Yoder, SIGLogo/ISTE
1787 Agate Street
Eugene, OR 97403

Monthly Musing

Do It Again!

by Tom Lough

During February and March of last year, I reported the responses of many readers to the question, "What is your favorite Logo command and why?" One of the most frequently mentioned commands was REPEAT. I got to thinking about that the other day. What about REPEAT would make it so popular?

The need for a REPEAT command is spontaneous and universal, as far as I can tell. One of the first questions I hear in a Logo workshop is "Isn't there some way to get the computer to do this again without having to type it [or scroll back up to it] each time?" We really seem to have this craving for an easy method of getting the computer to do the same thing over and over again. Molly Watt once told me that REPEAT was not included in the original MIT Logo version, but was added as a result of this expressed need. Hearing this makes me rather proud of REPEAT's special heritage.

Another special aspect of REPEAT is its ability to provide surprises. There is something heady about not being able to predict the outcome of several Logo commands repeated a bunch of times. I'm sure that nearly all LX readers have observed the astonishment and delight of a new Logo user who typed

```
REPEAT 500 [ several Logo commands
   here! ]
```

But REPEAT has some surprises in simpler (and more elegant) expressions, too. Remember an encounter similar to the following?

```
REPEAT 3 [FORWARD 50 RIGHT 60]
```

Oooops! Now, why doesn't that make an equilateral triangle?

I like the way REPEAT seems to invite experimenting with something resembling controlled conditions (related to the so-called scientific method). REPEAT makes it easy to change one aspect of a group of commands while keeping all others the same. For example, when students attempt to discover how to draw a five-pointed star, they often go through a process such as

```
REPEAT 5 [FORWARD 50 RIGHT 72]
REPEAT 5 [FORWARD 50 RIGHT 120]
REPEAT 5 [FORWARD 50 RIGHT 150]
REPEAT 5 [FORWARD 50 RIGHT 145]
```

whereupon they might stop, if the resulting figure seemed to be good enough. Varying the angle alone gives them some very special immediate feedback.

Here is a variation that has led to some interesting discussions about what affects size and what affects shape.

```
REPEAT 4 [FORWARD RANDOM 100
   RIGHT 90]
REPEAT 4 [FORWARD 100 RIGHT
   RANDOM 90]
REPEAT 4 [FORWARD RANDOM 100
   RIGHT RANDOM 90]
```

And, sooner or later, students get the idea of trying to put one REPEAT command inside another. This can lead to some really complicated results, especially if other Logo commands are sprinkled among the REPEATs.

```
REPEAT 5 [FORWARD 20 REPEAT 10
[RIGHT 38 BACK 40 REPEAT 4 [RIGHT 65
   FORWARD 70] LEFT 20]]
```

What about the one below? Care to make a prediction before you type this in?

```
REPEAT 1 [REPEAT 1 [REPEAT 1
   [REPEAT 1 [REPEAT 1
   [FORWARD 1]]]]]
```

Then try

```
REPEAT 2 [REPEAT 2 [REPEAT 2
   [REPEAT 2  REPEAT 2
   [FORWARD 1]]]]]
```

These aspects of REPEAT add up to a rather nice package. Once your students are hooked on REPEAT, then the stage is set for you to show them some really powerful stuff! I'll show you what I mean next month.

PS: What is the smallest value you can get REPEAT to accept for the number of repetitions of a list of commands and not produce an error message?

```
REPEAT 100 [ FD 1 ]
```

Tom Lough
Founding Editor
PO Box 394
Simsbury, CT 06070

Logo Ideas

"Driving" the Turtle
by Eadie Adamson

In the October 1989 issue of LX Diane Miller wrote about "The Turtle As Car," in which her student used a recursive procedure to move a car. Diane suggested that there were methods of steering a car as well. (See, for example, Tom Lough's Monthly Musings in May 1988, LX, page 3)

For the past several years I have been working with a group of boys developing motion games. One of their first tasks is to get the turtle moving as Diane's student did:

```
repeat 999 [forward 1]
```

proceeding next to a recursive procedure

```
to drive
forward 1
drive
end
```

and then to a recursive procedure with inputs.

```
to drive :speed
forward :speed
drive :speed
end
```

A turtle following a set path, however, did not satisfy my students for long. They knew they needed to control direction if they were ever going to adapt the procedure to a game!

How Can You Steer It?

There are several ways of adding "steering" to the simplest drive procedure, each with its own advantages and/or disadvantages. Here's one we used that does not require changing the original procedure.

First, we wrote procedures to output the cardinal directions, using numbers for headings:

```
to north
output 0
end

to south
output 180
end

to east
output 90
end

to west
output 270
end
```

Turning a turtle north, simply use seth north. For south, seth south. That much works fine for setting up a turtle to move, but how can you use this without changing drive? It's simple, actually!

Control Is the Answer

LogoWriter has 10 keys that can be programmed so that when the Control key is held down while one of these keys is pressed, a given command or series of commands or procedures will be activated, interrupting whatever else is in process on the screen. Programmed Control keys can even work independently of other procedures. Further, you can use them in the immediate mode as well, almost as if they were procedures. Puzzled? Read on....

The ten "programmable" keys are N O P Q R and V W X Y Z . To program a Control key use this form:

```
when "key [   here is where a task
   goes   ]
```

By typing an instruction such as

```
when "n [forward 50 right 90]
```

in the Command Center and then pressing Return, the "N" key is activated for as long as the computer remains on, or until the command **clearevents** is used. Pressing Control-N causes the turtle to go **forward 50 right 90.**

My students first explored this idea by activating the keys in the Command Center and then trying them out. However, a more efficient way of using these keys is to write a procedure that sets up the keys to be used. The idea that is sometimes hard for students to grasp is that if the instructions that define the actions of the Control keys are in a procedure, the *procedure must be invoked* to activate the keys. Simply writing a procedure on the flip side does not make the Control keys active.

The keys procedure can eventually go in a startup procedure on the page, but it is important to take the time to be sure the students understand that programming the keys alone is not what activates them. Spend some time setting up Control keys in a keys procedure, typing keys to activate the keys, testing them out, then typing **clearevents** to clear the keys, then running the keys procedure again to reactivate them.

With a "drive" and a "keys" procedure, the Control keys can be used to actually control the turtle's movement. What is fun for students in this context is perceiving the enormous extension this makes for their driving procedure without requiring them to reprogram **drive**. Somehow writing a second short procedure (or more) seems easier to many students!

Another Direction

One of the first things we tried when we first used LogoWriter was a project just like this. The first task was to create scenery, then get an object to move with a drive procedure. Instead of using Control keys for direction, however, we turned them into keys which changed the speed of the turtle and eventually made a kind of speedometer that reported the speed. How? Read on...

Changing Speeds

Students needed to have a drive procedure with inputs for the speed. Then I explained about the Control keys and showed them how to program a turtle to move faster. Pressing a Control key was used to alter the input to drive.

First, think about what occurs when something goes "faster." The drive procedure has an input, **speed**, that represents the "speed" in turtle steps. If "faster" is defined in terms of speed, you are increasing – adding to – the speed. The **speed** in the drive program is the input to forward, representing the number of turtle steps, or the distance, to be advanced each time the drive program is called.

Prior to working with Control keys, the method many students used to make their cars go faster was to stop the procedure and restart with a larger number for input. Faster, then, meant increasing the input for drive. How can this be expressed with a Control key? Like this:

```
when "z [make "speed :speed + 1]
```

What we are saying is: "Take the value of speed the turtle is currently using (a number) and add one to it. Call this the speed." We increment the speed by 1. This takes only a fraction of a second to take effect. Pressing Control and the letter z while **drive** is operating makes the turtle go faster; pressing the two keys again increases the speed again.

Slow Down or Stop!

Eventually the question arises: "How can I slow down the turtle?" There already is a model for making the speed faster. What makes a speed slower? Subtract instead of add! Most students have no difficulty figuring this out and choosing another key to program for slowing down the turtle.

Soon another question inevitably comes up: "How can I make the turtle stop without stopping the procedure?" Stopping means the speed is 0. We want to make the speed nothing, not less than the current speed or more than the current speed, but 0. That change can be expressed as:

```
make "speed 0
```

All we want is for the speed to be 0: nothing, no speed, not moving.

Once these three options are programmed, the other keys can be used for changing directions, changing "lanes" if the cars are on a highway, and so on. Some students wrote a "pass" procedure as well.

Speedometers

Earlier I mentioned speedometers. How can you get your program to tell you the "speed" each time a Control key is pressed to change the speed? Here's one way to handle the problem.

First, write a speed procedure. This procedure will display the speed in the Command Center, rather than on the screen. Since there may be commands in the Command Center, the procedure should clean that up first, then show the current "speed" of the turtle, the value of the variable speed. It might look like this:

```
to speed
cc
type:speed
end
```

(I prefer to use the command **type** here, rather than **show**. While **show** *will* display numbers without brackets, it poses a potential problem: if we add any text: the phrase will then appear in brackets. **type**,

Logo Ideas - continued

on the other hand, displays text without brackets. The fact that the cursor is left at the end of the line can be taken care of in two ways: either add one more line: **type char 13** (char 13 is the ascii value for the Return key) – or simply clear the commands as we did here each time the procedure was invoked.

Some students chose to print the speed on the screen. They needed to change the speed procedure to clear the text, **ct**, in order to avoid a string of numbers running down the left side of their scenery.)

More complex versions of a speedometer might use a multiplier and report a relative miles per hour:

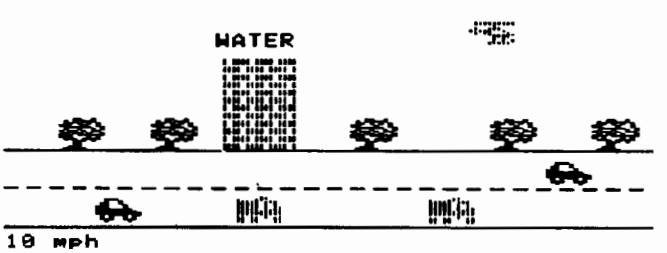
```
show sentence :speed * 10 "m.p.h.
```

This speed procedure is then used with each speed-altering Control key. Insert the word speed after the commands to change the value of speed. If there are three keys programmed to alter the speed, be sure to add the word speed just before the end bracket in each command. Here's an example:

```
when "z [make "speed :speed + 1
   speed]
```

Be sure to type the word **keys** (or whatever procedure name has been used to set up Control keys) before trying this. Simply making a change in the keys procedure on the flip side does not have any effect on the Control keys until the procedure is invoked again.

Type **drive 1** to begin. Watch the Command Center as keys are pressed to speed up and slow down or stop. Each time the speed is changed, the **speed** procedure will also be called into action. Voila! a speedometer!



Putting It All Together

The final important step is to write a superprocedure that sets up the keys and starts **drive**. Most of my students like to be able simply to type "go" and have things begin. Later the "go" procedure could also contain a setup procedure that put the vehicles into starting position. A simple "go" procedure would look like this:

```
to go
keys
drive 1
end
```

More ideas on driving at a later date. Meanwhile, enjoy the ride!

Eadie is a computer coordinator at The Allen-Stevenson School, an independent school for boys in New York City.

Eadie Adamson
1199 Park Avenue, Apt. 3A
New York, N.Y. 10128

Logo and Tangrams

by Lani Telander

As a teacher, I'm always looking for activities that will keep my students interested and my objectives accomplished at the same time. I have been doing a Tangram Unit with fifth graders for the past three years and each year I get more excited about the results.

My objectives for the Tangram Unit are to have the students work in a cooperative-learning activity, effectively using LogoWriter as a programming tool, and end up with a finished product of which they can be proud.

I begin by giving teams, usually two to three students to a team, the seven tangram puzzle pieces. They use the first couple of sessions to come up with a design the team agrees upon, using all seven pieces (some teams use more!) Each team then traces around the pieces on paper so they have their drawing from which to work. All of this work is done away from the computer.

The first step when the students begin working on the computer is to write a procedure for each tangram piece. This amounts to writing five procedures, since the tangram pieces consist of 1 square, 2 small triangles, 1 medium triangle, 2 large triangles, and 1 parallelogram. We use the following procedure for the square as the base upon which to write the other procedures:

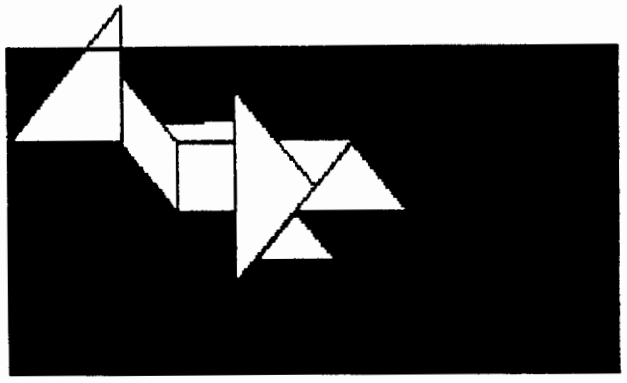
```
TO SQ
REPEAT 4 [FORWARD 30 RIGHT 90]
END
```

By using FORWARD 30 the tangram designs fit nicely on the screen without wrapping.

After the above procedures have been written, the challenging work of positioning the turtle begins before the next shape can be called up.

by Susan Bowers and Emilie Hitch

These fifth graders worked eagerly on this unit right up to the last session. They made use of the word processing feature of LogoWriter to write captions, TONE to enhance their designs with music, and, of course, color. In addition, this has been the most effective way for me to teach the Total Turtle Trip Theorem and SETHeading.



by Jason Grais and Geoffrey Nadler

This past year, the conclusion of our Tangram Unit coincided with Grandparents' Day at our school. We had the Computer Lab open so the fifth graders could share their work with their grandparents and parents. The enthusiasm and pride they showed for all their hard work was contagious!

Lani Telander
The Blake School
Highcroft Campus
301 Perry Lane
Wayzata, MN 55391

About the Cover

This month's cover shows the work of Lani Telander's students, as indicated below:

Ashleigh Cashman and Cassie Powell

Ben Mackay and Trevor Rusin

Robin Kreiser and Cindy Sher

Erin Clarkson and Andrew Steiner

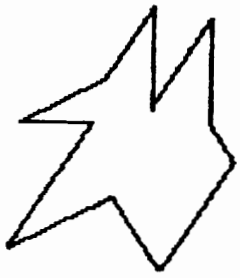
Beginner's Corner

Pencil & Paper or the Turtle?
by Dorothy Fitch

Have you noticed that sometimes the pictures that you plan are much easier to draw with a pencil and paper than to create using the turtle? And other times, it seems much easier to use the turtle than to attempt the same design with a pencil? This month, we'll take a look at different ways of approaching Logo designs.

Last month's column had a lot to do with stars. It reminded me of another stellar (and true) example of how there is often more than one approach to a problem in Logo.

Cathy, age 6, used Kinderlogo, a single keystroke Logo program in her first grade classroom once a week. The only commands available to her were F (to move the turtle forward 10 turtle steps), R (to make a 30 degree right turn), and L (to make a 30 degree left turn). Priscilla Flanagan, her teacher and Kinderlogo co-author, was present when Cathy created her star designs and related these fascinating observations to me.



With just three commands to use, Cathy drew this design one fall day. The intent was obviously a star and Cathy asked if her picture could be saved, although she didn't seem totally pleased with the result. But, her allotted time at the computer was up. We all know how frustrating that can be!

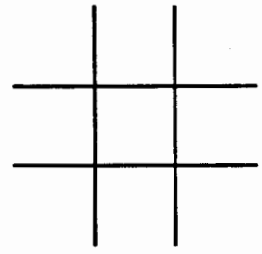
Later the same day, Cathy asked her teacher if she could have another turn at the computer. Although this was not normally allowed, her teacher sensed that this was somehow important and let her have some additional time. Cathy's next star looked like the star to the right:

What happened between the time Cathy drew the first star and the time, just a few hours later, when she drew the second star, we can only imagine. It is clear, however, that she must have spent some amount of time during the day thinking about the star and how she could improve on it. She must have known that she needed to use a different approach. Did the solution come in a flash, like a cartoon light bulb, or did she spend time doodling with a pencil, trying out possibilities? At some point she must have reached the conclusion that she should perhaps draw the star in Logo the way she would draw it with a pencil, not necessarily the way she'd seen stars drawn in books, with just the outline showing. Cathy was pleased with her second attempt at a star.

It really is quite exciting to see such concrete evidence of a thought process. Having witnessed this one isolated incident, it makes us wonder how often similar problem-solving occurs when Logo isn't there to provide "hardcopy."

Epilog: Cathy created this final star in the spring of the same year—a new variation on a familiar theme, and a neatly symmetrical one at that!

In this example, drawing the star in the same way you would with a pencil seemed to be the best approach. But that is not always the case in Logo. Consider a typical tic-tac-toe design, made up of just four straight lines.



If you were to draw this in Logo the same way you would with a pencil, you would have to pick the pencil up several times. This is the easiest way for humans, though not necessarily for turtles. Although in Logo you can determine the exact spots for placing the pen, your program might end up rather long, inflexible and hard to debug, like one of these rather gruesome examples:

```
TO TICTACTOE
PENUP
FORWARD 45
PENDOWN
BACK 90
PENUP
FORWARD 90
RIGHT 90
PENUP
FORWARD 30
LEFT 90
PENDOWN
BACK 90
PENUP
FORWARD 60
LEFT 90
PENDOWN
BACK 30
FORWARD 90
RIGHT 90
PENUP
BACK 30
LEFT 90
PENDOWN
BACK 90
END
```

```
TO TICTACTOE
PENUP
SETXY -15 45
PENDOWN
SETXY -15 (-45)
PENUP
SETXY 15 45
PENDOWN
SETXY 15 (-45)
PENUP
SETXY -45 15
PENDOWN
SETXY 45 15
PENUP
SETXY -45 (-15)
PENDOWN
SETXY 45 (-15)
END
```

These procedures work, but is either one the best way to draw it in Logo? Take a closer look at the tic-tac-toe design. Can you see any ways to repeat a pattern to draw the design? Remember that anytime that you can reduce a design to a repeated pattern, you gain a set of instructions that are:

- more efficient
- easier for you to debug or modify
- easier for others to read and understand
- more powerful and flexible
- more easily adapted for use in other projects that you develop

Here are some ideas for other ways to draw the tic-tac-toe design. You wouldn't choose any of these ways to draw the design with a pencil and paper, but they are rather suited to the turtle.

1. The C pattern

Picture the tic-tac-toe design made up of 4 squared-edged letter C's that look like this:

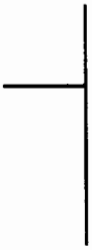
Just repeat the pattern four times for the tic-tac-toe design.

```
TO C
LEFT 90
FORWARD 30
BACK 30
RIGHT 90
FORWARD 30
RIGHT 90
BACK 30
FORWARD 30
END
```

```
TO 4C
REPEAT 4 [C]
END
```

2. The L shape design

In this version, the tic-tac-toe design is created using four double-L shapes, like this:

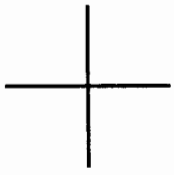


```
TO L
FORWARD 90
BACK 30
RIGHT 90
BACK 30
END
```

```
TO 4L
REPEAT 4 [L]
END
```

3. The letter T

Here is a version of the tic-tac-toe design that uses a repeated T-like shape:



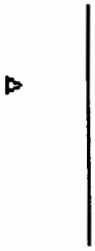
```
TO T
FORWARD 60
BACK 30
RIGHT 90
FORWARD 30
BACK 60
FORWARD 30
END
```

```
TO 4T
REPEAT 4 [T]
END
```

Beginner's Corner - continued

4. The jump, turn and draw method

Finally, this version causes the turtle to perform a pirouette midway through each repetition of the pattern. It goes forward, then hops over to the beginning point of the next line. It is similar to the L pattern above, but is actually a little quicker. Repeat the pattern four times to complete the tic-tac-toe design.



```
TO JUMP
FORWARD 90
LEFT 135
PENUP
FORWARD 42.43
PENDOWN
LEFT 135
END
```

```
TO DRAW.AND.JUMP
REPEAT 4 [JUMP]
END
```

The forward number 42.43 in the procedure JUMP (above) is the length of the hypotenuse of a right triangle whose sides are of length 30. The length is computed using the Pythagorean Theorem: $c^2 = a^2 + b^2$, or c (the hypotenuse) = the square root of $a^2 + b^2$. In Logo, you can find the length of the hypotenuse by typing:

```
PRINT SQRT (30 * 30 + 30 * 30)
```

Other Challenges

You can probably come up with additional ways of drawing the tic-tac-toe design. Why not challenge your students to write one or more REPEAT statements to draw the design?

Think of how you would transfer other pencil doodlings to Logo. Sometimes, as in the case of the star, the pencil approach may be quite satisfactory. For other designs, you may have to use a non-traditional approach. It is fun to try to draw the same design in as many ways as you can!

How would you draw letters of the alphabet using the turtle? I'll bet that you draw them differently with the turtle than with a pencil! Try drawing some letters exactly as you would with a pencil. For example, for the letter H you would first draw two vertical lines (picking the pen up in between), then connect them with the horizontal bar.

It becomes quite interesting to compare pencil techniques with turtle techniques, doesn't it?

A former education and computer consultant, Dorothy Fitch has been the Director of Product Development at Terrapin since 1987. She can be reached at:

Terrapin Software, Inc.
400 Riverside Street
Portland, ME 04103

Cooperative Creations

Third Grade Dynamic Poetry
by Jandy Bird

Since LogoWriter provides an excellent opportunity for cooperative learning, we decided to design a project for third graders in the computer lab that utilized the "jigsaw" technique of cooperative learning in a LogoWriter project. (See Slavin, 1988, Maring and others, 1985.) The jigsaw technique is quite simple in concept, but it takes planning to organize it. The idea of jigsaw is that a group is given a project to do cooperatively. Then the group is temporarily split up and each member of the group gets training in different specific skills necessary for completion of the project—a piece of the puzzle. Each member becomes an "expert" in something, and it is each expert's job to return to the group and teach the skill to the other group members. The project we chose was the translation of a poem into a piece of dynamic writing using LogoWriter. Time did not permit the students to write their own poems for this project, so the poem selected was "Shapes"by Shel Silverstein, a favorite poet of most of the students.

The first step was to demonstrate dynamic writing so the students could understand what the project was about. The beginning of a simple story was shown to the students, simply printed on the page.

> Once upon a time there was a cat who lived by himself in a house at the edge of the forest. One day, the cat looked out of the window and saw the most beautiful white rabbit in the yard.

(The story used the cat, rabbit, house and tree because the shapes were already available and familiar to the students).

Then the students were shown a dynamic version of the beginning of the story. After seeing the dynamic version, they were asked to figure out how the procedures worked. The flip side of the dynamic story page was analyzed and discussed.

The procedures for the dynamic version of the story are as follows:

First, to draw the forest :

```
to forest
circler 60
pu
right 90
forward 20
setc 2
setsh 23
pd
shade
ht
end
```

This procedure uses the tool procedure, circler, for drawing a circle.

Then the house

```
to house
pu
setpos [-45 25]
setc 5
setsh 20
st
pd
stamp
end
```

Then to bring in the cat

```
to cat
ht
setsh 21
setc 4
seth 180
pu
forward 20
st
repeat 10[forward 3]
end
```

Finally, to put the first sentence of the story together

```
to sen1
ct
rg
ht
print [Once upon a time there was a
   cat who lived by himself in a
   house at the edge of the forest.]
wait 60
forest
house
wait 20
cat
end
```

The wait commands are added to give time between the execution of the different procedures.

The second sentence of the story was animated as follows:

First the window

```
to window
draw.window
shade.window
end

to draw.window
ht
forward 50
repeat 3[right 90 forward 100]
right 90
forward 50
right 90
forward 100
back 50
seth 0
forward 50
back 100
end

to shade.window
pu
setpos [35 20]
setc 2
setsh 23
pd
shade
pu
setpos [80 20]
pd
shade
end
```

Then the action

```
to lookout
setsh 21
setc 4
pu
st
setpos [-130 -50]
seth 90
repeat 20[forward 5 wait 1]
pd
stamp
end
```

Then to put the second sentence together

```
to sen2
ct
rg
print [One day, the cat looked out of
   the window and saw the most beau-
   tiful white rabbit in the yard.]
wait 60
window
lookout
end
```

Then a combination of the first and second sentence animation procedures

```
TO DOALL
sen1
wait 60
sen2
wait 60
end
```

Finally, a superprocedure gets the circle tools and makes it run automatically.

```
to startup
ct
gettools "turtle.tools
doall
end
```

This demonstration made the process of dynamic writing clear to students before they began, and gave them an overview of how procedures can be linked in superprocedures.

The next step was to divide the class into groups of three and hand out a printed version of the poem they were going to use. As noted, we used the poem "Shapes" from *A Light in the Attic* by Shel Silverstein.

A square was sitting quietly
Outside his rectangular shack
When a triangle came down - kerplunk!-
And struck him in the back.
"I must go to the hospital,"
Cried the wounded square,
So a passing rolling circle
Picked him up and took him there.

This poem was selected because students were familiar with the poet, they had already worked with shapes and procedures to draw them, and because there are lots of possibilities for action in the poem.

Each group was to read the poem together, and to map out a simple story plan of how they would make the poem dynamic. When the groups had their plans complete, the jigsaw technique was implemented. Each member of the triad was assigned to a teaching group to review and learn different LogoWriter techniques that the triad would need for this project. That meant that each triad would be made up of three "experts" on different aspects of LogoWriter. The teaching groups were divided as follows:

1. This group learned LogoWriter terms for putting text on the page such as PRINT, TYPE, SHOW, INSERT. They also learned the WAIT command for pacing the presentation of text or graphics.

2. This group learned about how to place the turtle on the page where they wanted it to be. Using turtle move, the students learned SHOW POS and then SETPOS. (Note: There was not a lot of time spent on the fact that students are dealing with Cartesian coordinates here. The use of these numbers was simply shown as a way to get the turtle in position. A nice followup math lesson might deal with the position coordinates by themselves). This group also learned and reviewed SETH as a way to turtle in the direction you want. We used a turtle compass on an index card with a circle and numbers to show heading positions.

3. This group learned how to make shapes of their own on the shapes page.

The demonstration, planning, and creation of experts in three areas took one lab session of about an hour. There were obviously three teachers available to teach the expert groups. If it is not possible to have the three expert groups meet simultaneously, you would have to arrange the schedule differently. Up to this point, however, there were only three computers necessary for the project. If there is only one teacher directing this, then only one computer would be necessary to this point.

The remainder of the project was to have the groups work at the computer to make the poem dynamic. The students had about three lab sessions to work on this, plus whatever time the teacher made available to them in between lab sessions. Some groups did not complete the entire poem by the end, but everyone had part of the plan complete. At the end, the versions were shared.

Since several third grade classes did the same project, it was also possible to share between classes. There was lots of variety in approach to the poem, which provided a good example of creative problem sovling. Many students became involved in LogoWriter to a much greater extent through this project, and they also learned about working together. Some groups adventured into using more than one turtle, and others used sound. Some students even gave up recess in order to work extra time on the project! Although this project takes some time and organization, it is engaging to students and teachers alike, and it provides extensive experience with cooperative learning and problem solving strategies.

References

Cooperative Learning in Social Studies Education: What Does the Research Say? (1985) Washington, DC: National Institute of Education. (ED 264 162)

Maring, Gerald H., et. al. (1985). Five cooperative learning strategies for mainstreamed youngsters. *The Reading Teacher, 39*(3), 310-313.

Silverstein, Shel. (1981). *A Light in the Attic*. New York: Harper and Row.

Slavin, Robert E. (1988) *Student Team Learning: An Overview and Practical Guide. Second Edition.* Washington, DC: National Education Association. (ED 295 910)

Jan J. Bird, Ed.D
Conover Road School
80 Conover Road
Colts Neck, New Jersey 07722
201-946-8590
CIS: 73517,3270

Logo LinX

Transgendered Neology
by Judi Harris

Have you ever wondered why more *men* don't get *ma*nicures, or why *women* go through *men*opause? Might it be more appropriate for *women* to get *woma*nicures and go through *women*opause? Perhaps some playful a*women*dments to our dictionaries may encourage students to experi*women*t with syllabication and new vocabulary words in true Logo fashion.

Syllabic Di*women*sions

Which English syllables seem to specify gender? Your students will probably enjoy making lists of words with seemingly masculine and feminine parts. <groan> From these, you can code a HISYLLABLES procedure:

```
TO HISYLLABLES
OUTPUT [MAN MANS MEN MENS MENT MENTS
   MEND MENDS LAD LADS HE BOY BOYS
   HIM HIS MALE]
END
```

A corresponding HERSYLLABLES procedure should then be defined, which outputs an ordered list of the feminine syllabic complements of HISYLLABLES output. Both procedures can easily be a*women*ded as new syllables and their obverses are discovered.

```
TO HERSYLLABLES
OUTPUT [WOMAN WOMANS WOMEN WOMENS
   WOMENT WOMENTS WOMEND WOMENDS LASS
   LASSES SHE GIRL GIRLS HER HERS
   FEMALE]
END
```

Don't be surprised if your transgender explorers become a bit girlsterous during this activity.

Creating a Neological EnvironWOMENt

To transpose words with male syllables into their feminine counterparts, three Logo tools are needed. The procedure SUBSTITUTE searches user input for matches in HISYLLABLES' syllable list, then outputs corresponding HERSYLLABLES elements.

```
TO SUBSTITUTE :WORD.PART
IFELSE MEMBER? :WORD.PART HISYLLABLES
   [OUTPUT ITEM (ELEWOMENT :WORD.PART
   HISYLLABLES) HERSYLLABLES] [OUTPUT
   :WORD.PART]
END
```

SUBSTITUTE was written with Allison Birch's ELEMENT used as a subprocedure. We will, of course, rename the tool ELEWOMENT for obvious reasons.

```
TO ELEWOMENT :ITEM :OBJECT
IF EQUAL? :ITEM (FIRST :OBJECT)
   [OUTPUT 1] OUTPUT 1 + ELEWOMENT
   :ITEM BUTFIRST :OBJECT
END
```

The third tool, a superprocedure called NEW, accepts a list of syllables that comprise a word, and modifies it to reflect a more feminine di*men*sion.

```
TO NEW :LIST
IF EMPTY? [OUTPUT "]
OUTPUT WORD (SUBSTITUTE FIRST :LIST)
   NEW BUTFIRST :LIST
END
```

User Involve*woment*

To wield these neoteric tools, the user types (for example):

```
PRINT NEW [LA MEN TA BLE]
```

to which the computer responds:

```
LAWOMENTABLE
```

or,

```
PRINT NEW [PHE NO MEN ON]
```

which is transformed into:

```
PHENOWOMENON
```

It is up to you and your students to decide if this is an amusing and instructive supple*men*t to traditional syllabication practice and vocabulary study, or merely (as the computer printed) a la*men*table phenome*non*.

Analagous Arguwoments

It certainly should be *women*tioned that comple*women*tary examples of chauvinistically feminine words have invaded our language. Is it logical that most of the *her*oic patients who suffer from *her*nias between hospital *she*ets are male? Although *she*ep seem aptly named, especially when they travel in *her*ds, must all of their hu*man* attendants be *she*pards? Male *her*mit crabs must also find this di*men*sion of the language quite un*man*ageable (unless, of course, they live in *man*hattan). Yet, with a subtle a*men*dment to the SUBSTITUTE procedure, gender *man*age*men*t becomes a simple accomplish*men*t.

```
TO SUBSTITUTE :WORD.PART
IFELSE MEMBER? :WORD.PART
   HERSYLLABLES [OUTPUT ITEM
   (ELEMENT:WORD.PART HERSYLLABLES)
   HISYLLABLES] [OUTPUT :WORD.PART]
END

PRINT [WOMAN I ZER]

MANIZER
```

Perhaps it is even more desirable to *child*cott this bifurcated *person*ifestation of lingual *their*tory by suggesting that your students liberate lexicographic entries with a THEIRSYLLABLES procedure to add to their Logo transgender toolkit.

```
TO THEIRSYLLABLES
OUTPUT [PERSON PERSONS PEOPLE PEOPLES
   PEOPLET PEOPLETS PEOPLED PEOPLEDS
   CHILD CHILDS S/HE YOUNGSTER YOUNG-
   STERS THEM THEIRS UNSPECIFIED]
END

TO SUBSTITUTE :WORD.PART
IFELSE MEMBER? :WORD.PART HISYLLABLES
   [OUTPUT ITEM (ELEMENT :WORD.PART
   HISYLLABLES) THEIRSYLLABLES]
   [OUTPUT :WORD.PART]
END
```

No matter which no*women*clature you and your students select, it is my hope that this *person* uscript has, at least, added an interesting (if not bo*she*mian) di*men*sion to your *women*u of word study activity options.

AcknowledgeWOMENts

Birch, A. (1986). The Logo project book: Exploring words and lists. Cambridge, MA: Terrapin, Inc.

Searle, R. (1988). Ronald Searle's non-sexist dictionary. Berkeley, CA: Ten Speed Press. [Note: Graphically explicit content makes this book inappropriate for use as a classroom resource.]

Judi Harris taught students in Philadelphia-area elementary through graduate schools to use computers in teaching and learning for six years. She now does similar work at the University of Virginia's Curry School of Education, where she has recently completed her doctoral work in Instructional Technology. She can be reached at

Judi Harris
621F Madison Avenue
Charlottesville, VA 22903
CIS: 75116,1207
BitNet: JudiH@Virginia

A Random Toolkit for LogoWriter

by Charles E. Crume, B.S.

The RANDOM primitive of LogoWriter accepts a single input and reports an integer between zero and that number. For example, the command:

```
RANDOM 6
```

would report a number between zero and five, not a number between one and six as might be expected. Therefore, this specific command would not be appropriate to simulate the rolling of a die.

To circumvent this inconvenience of LogoWriter's RANDOM primitive, the command:

```
1 + RANDOM 6
```

could be used. The RANDOM 6 portion of the command reports a value between zero and five. Then, one is added to that value giving a final result between one and six. If however, a child writes the above command as:

```
RANDOM 6 + 1
```

it will not work properly. This form of the command reports a value between zero and six (not between one and six as would be expected). This is because the operation of addition (+) takes precedence over the primitive RANDOM. Writing the command in the above manner requires parentheses, as shown below:

```
(RANDOM 6) + 1
```

The parentheses are needed so that the RANDOM function is performed before the addition. In either case, both forms of the command are probably more complex and confusing than they need to be.

This article presents three useful procedures along with some sample programs that may make working with random numbers easier.

The first procedure, called RND, reports a positive random integer within a user specified range. The code is shown below:

```
TO RND :LOW :HIGH
OUTPUT (RANDOM :HIGH - :LOW + 1) +
   :LOW
END
```

This procedure requires the inputs LOW and HIGH (where LOW is the lower limit and HIGH is the upper limit). The procedure first computes the difference between the upper and lower limits (subtraction takes precedence over the RANDOM primitive) and then adds one to this value (addition takes precedence over the RANDOM primitive). Next, the procedure computes a random number between zero and this value. Finally, the procedure adds the lower limit to the random number. For example, the command:

```
RND 1 6
```

will always return a number between one and six inclusive. Changing the range is easy. For example, to simulate rolling a pair of die (whose output will always be a number between two and twelve), the command:

```
RND 2 12
```

is used. To obtain a random number in the range of 100 through 999, the command:

```
RND 100 999
```

is used. Compare the example above to

```
100 + RANDOM 900
```

or the form requiring parentheses

```
(RANDOM 900) + 100
```

neither of which have the value 999 (the upper limit) in them.

A sample program that uses the RND procedure plays a number guessing game. The child chooses a range of numbers, the program selects a number at random from that range, and then the child tries to guess the number. The code is shown below:

```
TO GUESS
RG
HT
CT
PRINT []
INSERT [PLEASE ENTER LOWER LIMIT...]
MAKE "LOWER FIRST READLIST
INSERT [PLEASE ENTER UPPER LIMIT...]
MAKE "UPPER FIRST READLIST
MAKE "NUMBER RND :LOWER :UPPER
CT
```

```
MAKE "GUESS.LIST []
CHECK
PRINT []
PRINT []
(PRINT [IT TOOK YOU] 1 + COUNT
   :GUESS.LIST [TRIES TO GUESS THE
   NUMBER.])
END

TO CHECK
INSERT [WHAT NUMBER DO YOU GUESS?]
MAKE "GUESS FIRST READLIST
IFELSE MEMBER? :GUESS :GUESS.LIST
   [PRINT [YOU ALREADY GUESSED THAT
   NUMBER!] CHECK STOP]  [MAKE
   "GUESS.LIST LPUT :GUESS
   :GUESS.LIST]
IF :GUESS < :LOWER [PRINT [YOUR GUESS
   IS BELOW THE LOWER  LIMIT] CHECK
   STOP]
IF :GUESS > :UPPER [PRINT [YOUR GUESS
   IS HIGHER THAN THE  UPPER LIMIT]
   CHECK STOP]
IF :GUESS = :NUMBER [PRINT [YEA, YEA,
   YEA. YOU GOT IT!]  STOP]
CHECK
END
```

Whereas the command RND 2 12 simulates the rolling of a pair of die, it returns only one value — the total of both die. A procedure that returns two numbers (each representing one of the die) is called DICEROLL. This procedure can be useful in analyzing how often each number appears on each die and for knowing when doubles have been rolled. The code is shown below:

```
TO DICEROLL
OUTPUT LIST RND 1 6 RND 1 6
END
```

The output from the command:

```
PRINT DICEROLL
```

can be any of the following:

```
[1 4] [2 2] [6 4]
```

A sample program that rolls the dice electronically is shown below:

```
TO ROLL
RG
HT
CT
LOOP
END

TO LOOP
PRINT []
INSERT [PRESS RETURN TO ROLL THE
   DICE...]
IGNORE READLIST
(PRINT [YOU ROLLED:] DICEROLL)
LOOP
END

TO IGNORE :RETURNKEY
END
```

Sometimes, a list of random numbers is needed. Instead of using the RND procedure in a REPEAT statement or recursive procedure, the procedure below can be used:

```
TO RNDLIST :LOW :HIGH :HOWMANY
IFELSE :HOWMANY = 1  [OUTPUT RND :LOW
   :HIGH]  [OUTPUT SENTENCE RND :LOW
   :HIGH RNDLIST :LOW :HIGH
   :HOWMANY - 1]
END
```

The procedure RNDLIST requires three inputs. The first two are the lower and upper limits as described for the procedure RND. The third input specifies how many random integers you want in the list. For example, the output of the command:

```
RNDLIST 1 6 10
```

can report any of the following:

```
[3 4 2 3 5 4 1 1 3 5]
[1 1 4 3 2 5 6 4 5 2]
[4 3 5 2 1 5 1 5 4 3]
```

Each list contains 10 integers, each between one and six inclusive.

Charles E. Crume, B.S.
Technical Consultant
University of Nevada System Computing Services
University of Nevada-Reno

MathWorlds

edited by
A. J. (Sandy) Dawson

Henri Picciotto is a teacher of elementary and secondary schools in San Francisco. He sent a long article which recounted his teaching experiences over an 18-year period using BASIC, Logo, and now Boxer. In each case, Henri was trying create educational software that fostered discovery based learning. Unfortunately, space does not allow for the inclusion of Henri's entire article.

Henri and his nine-year-old son came to Logo about the same time. Henri writes, "I was thrilled – here was a world which my...son and I could explore together. There was plenty to learn for both of us." Henri began teaching Logo instead of BASIC, and though he ran into some difficulties in doing this, he "...learned Logo myself. Armed with a new understanding of how computers can enhance the learning of mathematics, I decided to figure out how to apply this knowledge to my daily work. While I appreciated the wealth of non-traditional mathematics that could be taught with turtle geometry...I had little opportunity to explore these areas."

In what follows, Henri describes his efforts to find ways to use Logo in teaching the more traditional mathematics topics.

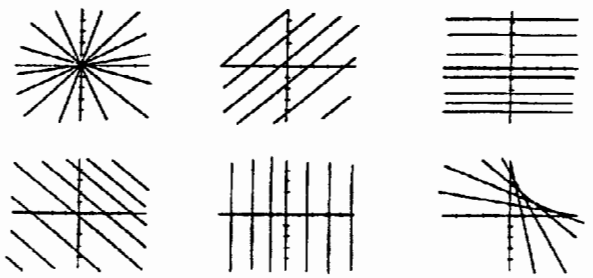
Teacher-Created Educational Software: Logo Tools

by Henri Picciotto

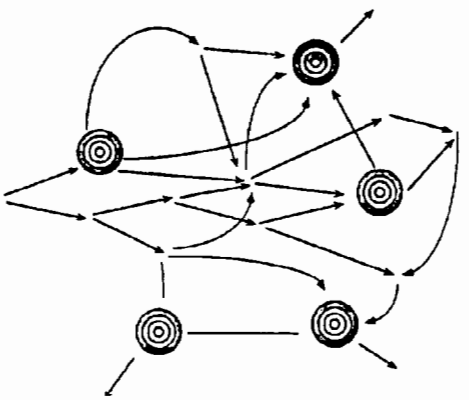
I set out to build an encyclopedic set of programs in that language, to be used in conjunction with the secondary mathematics curriculum (Picciotto, 1989). These include a GEOMETER, which adds the labeling of points to the Logo repertoire, and allows students to make geometric constructions on the screen, measure them, develop conjectures, and so on; and several games, each one a microworld, the exploration of which throws light on a specific curriculum area.

The most widely used program is a function GRAPHER, which has become an essential tool for the learning of algebra, trigonometry, and calculus at my school.

This is an example of an activity we use in Algebra 1: the students are given these arrangements of lines in various configurations. They are asked to create equations that will yield the given graphs. (This can be done before they get any formal exposition to such concepts as slope and intercept, providing an opportunity to discover those concepts, or later on, as a review of the concepts.) The flavor of this software



tool is quite different from the BASIC programs I had previously developed. For one thing, the program does not include any preconceived lessons. The teacher has to plan the lesson (an old-fashioned approach, granted, but one which works!) Moreover, the program does not control the student. The student can select any of the figures as a goal, and pursue them in any order. An input that is incorrect for one figure may be correct for another one. This leads to the possibility of many alternative strategies, and in fact students develop different styles to attack the problems. Beyond this, it becomes possible for the students to set their own goals. (For example: how would one graph a family of lines that intersect at the point (1,1)?) Discussions among students become very focused and productive. The teacher is presented with a plethora of teaching opportunities.

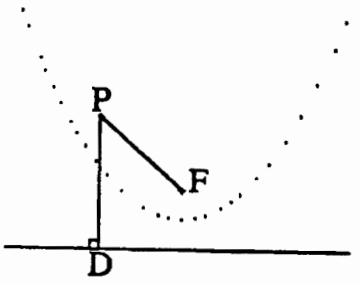


My favorite program in the package is called LOCATER. It allows the student to explore locus problems — with several uses in Geometry, Algebra 2, Trigonometry, plus the possibil-

ity of offbeat investigations. The following figures show how by setting

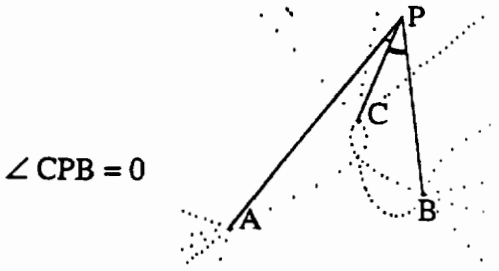
PF/PD = 1,

the program allows students to find and plot points that are equidistant from the line and point F.



The resulting curve is a parabola, a classical topic in high school math, approached here in a pleasing geometric way. But unusual questions can also be asked.

The next figure shows the result of the investigation of the question: "From where in the plane does ΔABC appear to be isosceles?" (In other words, for what points P does CPA = CPB?)



The result, an unexpected mixture of rays and curves, would not have been easy to predict without the help of a computer tool.

These Tools and Games are complex, and well beyond the ability of a novice programmer to write. However, they do not attempt to be professionally slick or fool-proof. For example I did not try to filter input at the keyboard in order to prevent all possible mistakes at the expense of constraining the user. Thus, these programs are less polished, and sometimes slower and less powerful than the best among comparable non-Logo programs, such as the Sunburst products. But they do have five major advantages:

First, the user interface is consistent: all programs function roughly in the same way, so that less classroom time is spent learning to use the programs, and more is spent actually using them.

Second, the tools can be demystified, even to the non-programmer, by using them at various levels. For example, one can graph a curve (say a parabola) by entering its equation (e.g., y = x2 - 3) and then typing GR. But one can break down the process, and plot each individual point (e.g., POINT 2 1). One can quickly calculate the y-coordinate of any point on the curve (F 4 will return 13). By plotting several points "manually," students simulate both the old-fashioned method of plotting by hand, and the automated procedure embodied in the command GR, and bridge the conceptual gap between the two. Similarly, any tool can be broken down into smaller, more understandable chunks.

Third, the code is accessible to any Logo-literate student or teacher, allowing them to inspect or modify it as they see fit, and conveying the idea that programming in Logo, a skill they know is within their reach, can yield powerful, useful tools.

The fourth advantage of Logo tools and games over the slickly packaged competition is that the user has full access to the Logo language at all times – its computational, graphic, and programming capabilities. This means that considerably more power is always at the student's fingertips than one could conceivably include in any one program. (For example, one can design a lesson on area and perimeter of rectangles that uses Logo to draw various rectangles of a given perimeter and output their area for various lengths, with the goal of finding the rectangle of maximum area. Then, using GRAPHER, one can plot area as a function of length of one side, and look for the maximum by inspecting the graph. It would be absurd to try to create a special program for each lesson like this one that one might think up.)

Fifth, students can write their own procedures that call mine as subprocedures. For example, in the GEOMETER, they can write a procedure to inscribe a circle in a triangle, by using my BISECT, LLINT (line-line intersection), DROPTO, and CIRCLE procedures as if they were primitives.

Overall, the introduction of Logo tools and games has been a tremendous boon to the mathematics department at our school. It has worked hand-in-hand with the introduction of

Math Worlds - continued

cooperative learning groups and a discovery-based approach to learning math. Nevertheless, the last three advantages described above have not yet been fully realized, mainly because most of our students do not know Logo or, for that matter, any programming language. As a result, fiddling with the procedure that controls screen colors has been the major way that my students have modified my programs. Of course, I had hoped for a much richer interplay between the user and the tool. To try to achieve this, we now demand that all students take a half-semester introduction to Computer Science as a graduation requirement. This will in fact mean Logo. The result of such a policy, within a few years, will be an entire community that shares some literacy in the computer language in which the software they use in their daily work is written. This may be a first in education, and I anxiously await the results of the experiment.

Reference:
Picciotto, H. (1989). *Logomath: Tools and games*. Malden, MA: Terrapin, Inc.

Henri Picciotto can be contacted at
The Urban School of San Francisco,
1563 Page Street,
San Francisco, CA 94117

A. J. (Sandy) Dawson is a member of the
Faculty of Education at
Simon Fraser University in Vancouver, Canada. He can be reached electronically through Bitnet as
userDaws@SFU.BITNET

Teaching Teachers

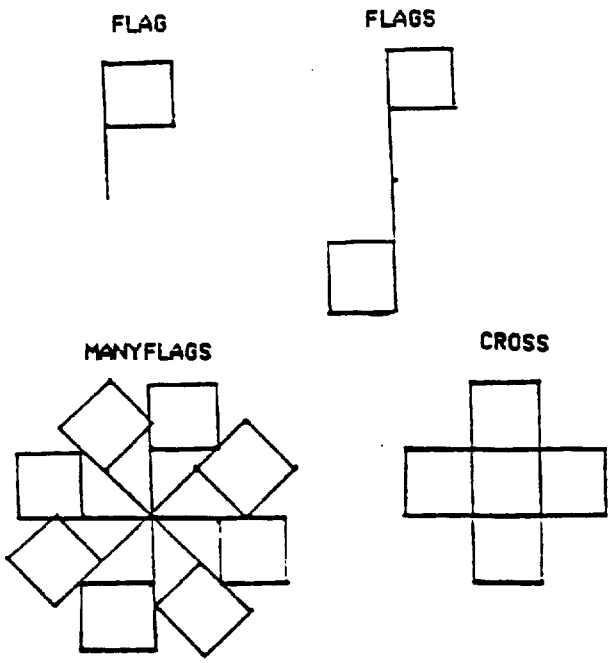
Teachers Beginning to Think in Logo

by Richard Austin

Logo is a popular computer language with elementary teachers, because it is easy to write procedures which "draw" designs on the monitor screen. For this reason, I include a session on Logo in an introductory class about classroom applications of computers. The amount of Logo covered is by necessity only a small bit of the turtle graphics portion of the language. An important point, made several times during the presentation, is that it is possible to have several different procedures that are correct; i.e., the results displayed on the screen satisfy the assignment. If teachers are aware that they approach problems differently, they should be prepared to accept alternate solutions from their students. Following a presentation of about one and a half hours, the students then work in the computer lab writing their own procedures to draw four figures shown on the lab assignment sheet.

The students in this class are mostly teachers with no previous computer experience. The teaching assignments and areas of interest of these students cover the entire range of teaching positions. During the lab sessions, students are encouraged to, and often do, work together in teams of two.

The lab assignment was to write procedures named FLAG, FLAGS, MANYFLAGS, and CROSS, which would produce the corresponding figures shown below:



In evaluating the lab work for the Logo exercise, I enter the procedure name and simply look at the figure that is displayed on the screen. If the figure looks like the one on the assignment sheet, full credit is given. As the instructor I did look at the listing of the procedures out of my own curiosity. Having been given hints during the earlier class presentation, most of the students wrote very similar procedures for the FLAG, FLAGS, and MANYFLAGS. The CROSS procedure however, caused students to think and be a bit more creative than they had been in writing procedures for the first three figures. I found it interesting to examine the several acceptable procedures for making the figure CROSS that I received from these students.

Most students had procedures for SQUARE and FLAG as follows in their workspace. (All of the work was done using Apple Logo.)

```
TO SQUARE
REPEAT 4 [FORWARD 30  RIGHT 90]
END

TO FLAG
FORWARD 30
SQUARE
BACK 30
END
```

For those students who wrote a different procedure for FLAG and used it in their CROSS procedure, I will provide their listing for FLAG.

For all examples, I edited only the distance so that the length of each square would be 30 turtle steps. This was not part of the assignment, but will make the resulting figures the same size and thus possibly easier to visualize the differences in how the procedures were written.

I have identified fifteen procedures that are different. Some are very much like others, but still reflect somewhat different thinking. Some of the procedures were used by several lab groups and some were unique. I was unable to make any analysis or further evaluation of these procedures. My interest was in the wide range of thinking reflected in the way in which student teams approached this problem (for many students, the CROSS procedure was a very real problem for which they could see no immediate approach). These students had only the bare minimum exposure to Logo before beginning this exercise. All of the procedures listed here received full credit as they produced the proper cross pattern on the screen.

Example 1

```
TO CROSS
REPEAT 4
   [SQUARE
   RIGHT 90
   FORWARD 30]
END
```

Example 2

```
TO CROSS
REPEAT 4 [FLAG
   FORWARD 30
   RIGHT 90]
END
```

Example 3

```
TO CROSS
SQUARE
LEFT 90
SQUARE
FORWARD 30
SQUARE
LEFT 180
SQUARE
LEFT 90
FORWARD 30
SQUARE
END
```

Example 4

```
TO CROSS
SQUARE
RIGHT 90
FORWARD 30
SQUARE
RIGHT 90
FORWARD 30
SQUARE
RIGHT 90
FORWARD 30
SQUARE
END
```

Example 5

```
TO CROSS
FORWARD 45
RIGHT 90
FORWARD 30
RIGHT 90
FORWARD 90
RIGHT 90
FORWARD 30
RIGHT 90
FORWARD 60
LEFT 90
FORWARD 30
LEFT 90
FORWARD 90
LEFT 90
FORWARD 30
LEFT 90
FORWARD 30
LEFT 90
FORWARD 15
END
```

Example 6

```
TO CROSS
LEFT 90
FORWARD 45
RIGHT 90
FORWARD 30
RIGHT 90
FORWARD 90
RIGHT 90
FORWARD 30
RIGHT 90
FORWARD 60
RIGHT 90
FORWARD 60
RIGHT 90
FORWARD 30
RIGHT 90
FORWARD 90
RIGHT 90
FORWARD 30
RIGHT 90
FORWARD 30
END
```

Example 7

```
TO CROSS
SQUARE
REPEAT 4
   [FORWARD 30
   SQUARE
   RIGHT 90]
END
```

Math Worlds - continued

Example 8

```
TO CROSS
FORWARD 30
SQUARE
REPEAT 4
[FORWARD 30
   SQUARE
   RIGHT 90]
END
```

Example 9

```
TO CROSS
REPEAT 4 [FLAG
   RIGHT 90]
END

TO FLAG
FORWARD  30
SQUARE
END
```

Example 10

```
TO CROSS
SQUARE
RIGHT 90
FLAG
LEFT 90
SQUARE
FORWARD 30
LEFT 90
SQUARE
END

TO FLAG
FORWARD 60
REPEAT 3
   [RIGHT 90
   FORWARD 30]
LEFT 90
FORWARD 30
END
```

Example 11

```
TO CROSS
PENUP
LEFT 90
FORWARD 15
PENDOWN
RECTANGLE
PENUP
HOME
FORWARD 15
PENDOWN
RECTANGLE
END

TO RECTANGLE
LEFT 90
FORWARD 45
LEFT 90
FORWARD 30
LEFT 90
FORWARD 30
LEFT 90
FORWARD 30
LEFT 90
FORWARD 45
END
```

Example 12

```
TO CROSS
REPEAT 2 [BLOCK
   RIGHT 90]
FORWARD 30
RIGHT 90
FORWARD 30
RIGHT 90
END

TO BLOCK
FORWARD 60
RIGHT 90
FORWARD 30
RIGHT 90
FORWARD 90
RIGHT 90
FORWARD 30
RIGHT 90
FORWARD 60
END
```

Example 13

```
TO CROSS
REPEAT 4 [FOR-
   WARD 30
   RIGHT 90]
LEFT  90
REPEAT 4 [CRY]
END

TO CRY
REPEAT 3
[FORWARD 30
   RIGHT 90]
LEFT 180
END
```

Example 14

```
TO CROSS
SQUARE
FORWARD 30
SQUARE
LEFT 180
FORWARD 30
RIGHT 90
SQUARE
RIGHT 180
FORWARD 30
LEFT 90
SQUARE
RIGHT 180
SQUARE
END
```

Example 15

```
TO CROSS
FLAG
RIGHT 180
FORWARD 30
RIGHT 180
FLAG
RIGHT 90
FORWARD 30
LEFT 90
FLAG
FORWARD 30
LEFT 90
FLAG
END
```

These procedures clearly illustrate that there are indeed several ways to write equivalent procedures in the Logo language. That the teachers solved the same problem by writing the procedures above reflects this variability. These procedures were collected from 10 different classes over a four-year time interval.

Richard Austin is an assistant professor in the Curriculum and Instruction Department at the University of Tennessee, Knoxville, working in the areas of mathematics education and instructional computing.

Richard Austin
301 Claxton Education Building
University of Tennessee
Knoxville, Tennessee 37996-3400

Logo & Company

Creating SETX and SETY in HyperCard

by Glen L. Bull and Gina L. Bull

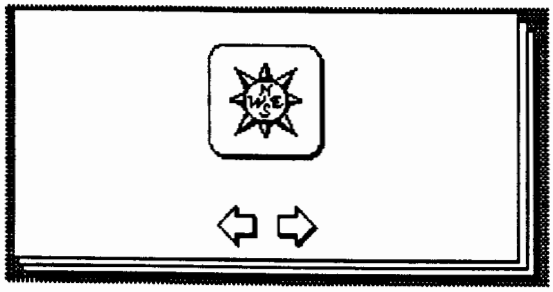
In last month's column we showed you how to create a *HyperCard* button called compass, and how to write *HyperCard* commands which would move the compass around the screen much as Logo commands can be used to move the turtle around the Logo screen. The button named "Compass" looked like this.

If you have not already created this button, and don't yet know how to make one, you will need to refer to the instructions in last month's column. Once you have a button named "Compass" that looks like the one above, you can change its location by typing commands such as the following in the Message Box.

```
set the loc of card button compass to 50,100
```

If the message box is not showing, you can make it appear by holding down the Command key (the one to the left of the spacebar with a picture of a cloverleaf) and typing M. The particular command shown above sets the compass button to a position 50 pixels from the left edge of the screen and 100 pixels below the top of the screen.

In Logo it is not only possible to move the turtle around the screen with turtle graphics commands, it is also possible to determine the position of the turtle on the screen with commands such as XCOR and YCOR. In *HyperCard* you could determine the position of the compass with a similar command. Assume that the position of the compass is in the middle of the screen, as shown below.



You could determine the exact location of the compass button by typing the following in the Message Box.

```
put the loc of card button compass into msg box
```

In this instance *HyperCard* might indicate that the coordinates of the button were 256,171.

```
256,171
```

If you do not tell *HyperCard* where to put the location of the compass button it assumes that you mean the Message Box, so the command could also be written in the following way.

```
put the loc of card button compass
```

We will use the "loc" command in *HyperCard* to write XCOR and YCOR commands very similar to the same commands in Logo. Some versions of Logo do not have XCOR and YCOR commands that give the X and Y coordinates of the Turtle separately. However, almost all versions of Logo have a POS command that gives the X and Y coordinates of the Turtle together, much as the "loc" command in HyperCard gives the combined X and Y coordinates of the compass button. In Logo the XCOR and YCOR commands can be created using the POS command, if they are not already available. We will explain how this is done in Logo first, and then use the same logic to create similar commands in HyperCard.

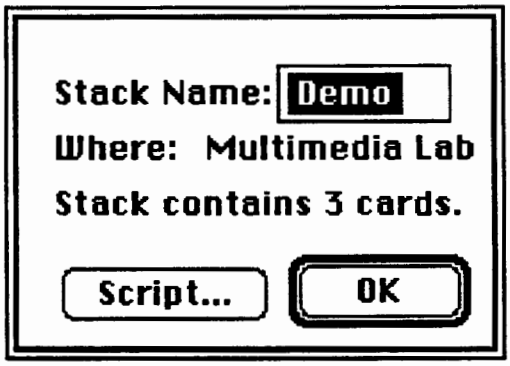
In Logo the first position provided by the POS command is the X coordinate and the second position is the Y coordinate. XCOR and YCOR commands can be created in Logo in the following way (if they are not already available).

```
TO XCOR
OUTPUT FIRST POS
END

TO YCOR
OUTPUT LAST POS
END
```

Logo & Company - continued

In Logo these commands are written in the Logo editor. To get to the equivalent of the Logo editor in *HyperCard*, go to the **Objects** menu on the menu bar at the top of the screen, and select the **Stack Info** option.

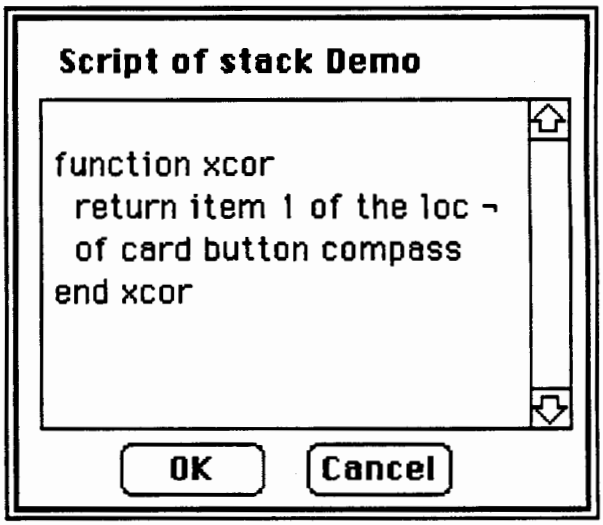
After you select **Stack Info**, a dialog box similar to the following will appear.



Click on the **Script** button of the Stack Info dialog box. The Script editor should appear. If you copied the background of the Home Card when you created your stack, you may find there is already some text in your Stack script resembling that shown in the box below. You should delete this text before you go on to enter your own script. (If you do not want the Stack script of the Home Card to be copied into future stacks, you should be sure the "Copy Current Background" box is not checked when you create new stacks.)

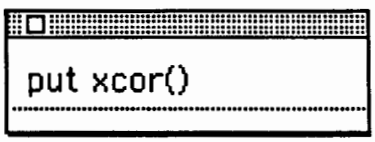
```
on c
  choose browse tool
  doMenu "Card Info..."
end c

on b
  choose browse tool
  doMenu "Bkgnd Info..."
end b
```

Hypertalk is the programming language that accompanies *HyperCard*. A program in Hypertalk is called a script, just as a program in Logo is called a procedure. After you have deleted the previous text in the Stack script (if any), type the following in the Script editor. It will be your first Hypertalk script.



Important note: the symbol "¬" at the end of a line in *HyperCard* indicates that the command is continued on the next line. If there is not room for the command to fit on one line, hold down the "Option" key as you press return, and the "¬" symbol will automatically be generated.

After you have typed the program you can click "OK" in the Script editor dialog box. Now you are ready to use the new XCOR command that you have created in *HyperCard*. Type the following to try it out.



HyperCard should put the X coordinate of the compass button in the Message Box after you press the **Return** key. (Be sure to include the set of parentheses shown, with no space between the parentheses and xcor.) Let's compare the XCOR command we have written in Hypertalk with the XCOR command in Logo. Of course, there is the obvious difference that you must include a set of parentheses after the XCOR command in *HyperCard* in order for it to work.

There are more subtle differences, such as the fact that procedures in Logo tend to be mostly written in upper case, while scripts in Hypertalk tend to be written mostly in lower case. However, those are esthetic differences, and the programs in each language will still run even if you violate conventions regarding case.

Logo Version

```
TO XCOR
OUTPUT FIRST POS
END
```

Hypertalk Version

```
function xcor
  return item 1 of the loc ¬
  of card button compass
end xcor
```

Both programs in this instance are actually *functions*. A function is a program that returns some information to another program. In Logo the OUTPUT command is used to signal that this information is to be provided. The Hypertalk equivalent of OUTPUT is **return**. Although there are some differences between the two functions, it is also clear that there are some correspondences between Logo and Hypertalk, and that the form of the two functions is somewhat similar.

If the Hypertalk XCOR function worked properly, return to the Stack script and add the YCOR function. It is almost identical to the XCOR function:

```
function ycor
  return item 2 of the loc ¬
  of card button compass
end xcor
```

Now that we have equivalents of the XCOR and YCOR functions in Logo, let's create commands that are the equivalent of SETX and SETY. XCOR and YCOR tell where the turtle is (or the compass button in the case of our HyperCard stack) while SETX and SETY will set the position of the turtle or compass to a specific X or Y coordinate. We wrote the Hypertalk equivalent of XCOR and YCOR first because they are needed to create SETX and SETY.

The following procedure (or handler, as procedures are called in Hypertalk) uses YCOR in the SETX command. It says, "Set the location of card button 'Compass' to the value of an X input and the current Y coordinate." Type the following procedure into the Stack script.

```
on setx xpoint
  set the loc of card button ¬
  compass to xpoint,ycor()
end setx
```

Once you have entered the procedure, try it out by typing the following into the Message Box, and pressing **Return**. Did the compass button move to the left side of the screen? If

the SETX procedure worked, you are ready to add a SETY command. Add this procedure to the Stack script.

```
on sety ypoint
  set the loc of card button ¬
  compass to xcor(),ypoint
end sety
```

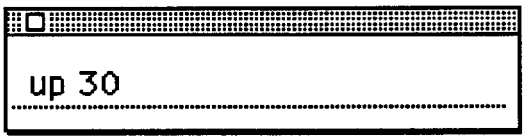
SETX and SETY allow you to set the position of the compass button in HyperCard, just as you can set the position of the Turtle with comparable SETX and SETY commands in Logo. We created SETX and SETY so you could compare the anatomy of Hypertalk scripts with Logo procedures. There are enough similarities that with a little practice you should be able to apply all of the programming expertise that you developed in Logo to Hypertalk as well.

SETX and SETY are absolute commands; they send the turtle to a fixed point on the screen. Most of the time Logo programmers use relative commands such as FORWARD and BACK. These commands move the turtle to another point relative to its current position on the screen. We can create some relative commands in *HyperCard* to complement the SETX and SETY commands that we have just created. The following Hypertalk handler will allow us to move the compass button up by a certain amount.

```
on up distance
  set the loc of card button ¬
  compass to xcor(),ycor() - distance
end up
```

Logo & Company - continued

After you add the UP procedure to the Stack script, type the following in the Message Box:



Did the compass button move up by 30 steps? You may want to add a DOWN procedure to the stack script as well.

```
on down distance
  set the loc of card button ¬
  compass to xcor(),ycor() + distance
end down
```

Now that you have seen two examples that move the compass up and down, can you develop relative commands that move the compass button from side to side in *HyperCard?*

In Logo there is only one place to create new procedures, in a procedure editor. This is roughly the equivalent of the Stack script in *HyperCard*. However, in *HyperCard* there are several places in which procedures can be created. The script of the *HyperCard* stack is one; scripts of *HyperCard* buttons are another. In last month's column we created a button called "Center" and added a command to its script that centers the compass button on the screen.

Let's make a button that will move the compass button up. Go to the **Objects** option on the menu bar at the top of the screen, and select the **New Button** option. Then double-click on the New Button that you created to go to the **Button Info** dialog box. Follow steps similar to the ones that you used to create the "Center" button last month to create an "Up" button:

- Type UP (in place of "New Button") in the Button Name box
- Click the "Autohilite" box to select this feature
- Click the Script button to go to the Script Editor of the UP button

After you are in the Script editor of the **UP** button, type the following script:

```
on mouseUp
      up 30
end mouseUp
```

Then click the OK button of the Script editor. After you return to the *HyperCard* screen, select the **Browse** tool (the one that looks like a hand) from the Tools palette. Place the finger of the **Browse** tool in the middle of the **UP** button, and click the mouse button once. Did the compass button move up by 30 steps?

Congratulations! You have just created a "sticky button" similar to ones in the experimental version of Atari Logo that we described last month. When the Button tool is selected on the tools palette, you can drag the UP button around the screen and place it anywhere you want. When the Browse tool is selected, you can use the finger of the browse hand to press the UP button – which, in turn, will cause the compass to move up the *HyperCard* screen by 30 steps.

Now that you have created an UP button, you may wish to develop others—possibly a DOWN button, a LEFT button, and a RIGHT button. You have created the beginnings of an "Instant Logo" in HyperCard. For now you may want to find a young child who can try out these new commands, but eventually you may want to add more commands to create a *HyperCard* "microworld."

Summary

In last month's column on parallels between Logo and *HyperCard*, we showed you how to do the following:

- Start the *HyperCard* program
- Create a new stack
- Create *HyperCard* buttons
- Enter commands in the Message Box

In this month's column we showed you some additional programming features of *HyperCard*, including the following:

- How to enter procedures in the Stack script
- How to write a Hypertalk procedure
- How to write a Hypertalk function
- How to use inputs in Hypertalk procedures

We hope that this has encouraged you to investigate *HyperCard* on the Macintosh. *HyperCard* has also inspired similar programs on other computers, such as *LinkWay* for the IBM, and *HyperScreen*, *Hyper Studio*, and *Tutor Tech* for the Apple

II computers. We hope to explore some of these programs in future columns.

Last month we recommended the *HyperCard Handbook* by Danny Goodman. The *HyperCard Handbook* is a good beginning introduction and reference manual for Hypercard. It is possible to do a great deal in HyperCard without doing any programming at all. If you would like to write more Hypertalk scripts, *HyperTalk Programming* by Dan Shafer, published by Hayden Books, is a good introduction to programming in Hypertalk.

Glen Bull is a member of the instructional technology faculty in the Curry School of Education at the University of Virginia. Gina Bull is a programmer analyst for the University of Virginia Department of Computer Science. By day she works in a Unix environment; by night, in a Logo environment.

Glen and Gina Bull
Curry School of Education
Ruffner Hall
University of Virginia
Charlottesville, VA 22903

BITNET addresses:
Glen: LB2B@ VIRGINIA. Gina: RLBOP@VIRGINIA.

Logo: Search and Research

Stages of Learning Programming
by Douglas H. Clements

Do students pass through recognizable *stages* as they learn to program? Yes. As with knowing what's hard for students in learning Logo, knowing about stages of learning is invaluable teaching effectively with Logo.

"First steps" in learning Logo commands

A three-step model of children's initial learning of Logo commands has received research support (Fein, Scholnick, Campbell, Schwartz, & Frank, 1988).

Step 1. A good deal of what children learn involves differentiation. At first, the child has a single global concept, "turtle," defined in terms of undifferentiated movement. Children begin to differentiate types of commands—FORWARD/BACK vs. turns. However, they prefer and more easily learn about FORWARD and BACK. This is probably because turn commands only produce a rotation of the turtle, a movement that must be seen as it is happening. In addition, turn commands can be confusing because of many children's lack of mastery of right and left and perspective-taking abilities.

Children then differentiate commands within each type. Forward is favored over backward, probably because children usually move forward themselves and because the turtle's nose favors a forward trajectory. Because backward is rarely used, children are almost forced to differentiate between the two turn commands. Right is favored over left.

Step 2. At first, children use just four command "strings": FORWARD, RIGHT, FORWARD-then-RIGHT, and RIGHT-then-FORWARD. At step 2, children synthesize these fragments into a greater whole. For example, they figure out that any right move can be reversed by an appropriate move to the left. They understand why FORWARD then RIGHT is not the same as RIGHT then FORWARD. This allows them to explore all combinations of the commands.

Children can now produce the designs they wish. They make fewer mistakes in syntax and can debug more efficiently.

Step 3. Children build up more complex relationships. For example, they discover the relations between rotations and distances. Back is equivalent to a 180° turn followed by a forward. So there are as many as four ways to reach the same target and many more ways to draw the same figure.

Thus, three sets of powerful ideas are inherent in the semantics and syntax of primitive Logo commands: ideas about space, ideas about syntax, and ideas about relations between these two. Spatial knowledge is discovered in the first step. Simple relations and syntax emerge in the second. Complex relationships, combinatorial flexibility, and integration appear in the third. *If* we teachers make this knowledge explicit and help children forge links to other knowledge, the benefits of such learning are clear.

This work builds upon previous research on young children's spatial concepts. Using a button box (FORWARD, RIGHT, LEFT) more than a decade ago, Gregg (1978) identified five stages in learning to direct a turtle robot: (I) comprehension that the buttons have some effect on the turtle; (II) differentiation of the FORWARD button from the other two; (III) recognition that the turn buttons have separate functions; (IV) recognition that the turn buttons make the turtle rotate on its axis; and (V) full understanding. Gregg hypothesized that 4- and 5-year-old children initially were unable to map the appropriate buttons onto the appropriate turtle part and so could not select the correct button.

Another researcher focused on the concepts of right and left (Roberts, 1984). At level 1, children can reliably identify their own right and left sides, but cannot use the distinction for specifying directions to the turtle at all. Level 2 children adopt an egocentric frame of reference. When the direction to turn the turtle corresponded to their own left or right side (e.g., 0 or 180 degrees), they choose that side as the direction to turn the turtle; otherwise, they perform at chance level (90 or 270 degrees). At level 3, children make correct judgments for 90 and 270 degrees (performance at this level was uncommon). Level 4 children make correct judgments at all orientations. Roberts suggested that children at higher levels perform imagined mental rotations of themselves. We have all seen children twist their bodies into alignment with the turtle in an attempt to figure out a turn! That is, they seem to "project" themselves into the place of the turtle. (More about this issue, include notes on teaching, is provided in the previous column, "What's hard about beginning with Logo? The research.")

These results led Roberts (1984) to conclude that children not at level 4 would be seriously limited in their ability to write Logo programs due to their inability to understand how the turn commands operate at all possible orientations. This provides additional evidence of the importance of providing children at lower levels with special support programs, such as INSTANT, SINGLEKEY, and TEACH, which ameliorate such difficulties while permitting the construction of complex programs (Clements, 1983-84; Clements & Battista, in press; Clements & Nastasi, in press).

Stages in programming

Other researchers have investigated stages in learning about programming that go beyond the "commands" level. Kull (1986) observed three steps in first graders' programming ability. At the first step, children practiced in immediate mode, took notes, then copied the code from the notebook onto the screen in the editor, often including the errors. They then ran the procedures, noted the bugs, and learned how to edit. They soon wrote down only the moves that worked: "What are you doing?" "I'm writing down the moves that will help it make the J." "What about the moves that don't help it?" "I just don't write those down."

At the second step, children recognized that this process was cumbersome, and began planning several moves ahead, writing them down, then checking them for accuracy in immediate mode, and finally adding them to a procedure. At the third step, children began writing whole procedures at once (around March for the majority). Debugging was easy because they had a strong sense of the sequence of instructions used to control the turtle. Kull noted that the time spent drawing and taking notes for an important developmental step in learning to program effectively. Many teachers attest to the benefit of students keeping Logo notebooks or journals.

Working with older students, Howe (1980) reported three stages of learning Logo programming. In the *product oriented stage*, student attempted to produce effects without concern for the method used. *Style-conscious* students made an effort to program in a correct style (as defined by worksheets the researcher provided). At the highest stage, *creative problem solving*, Logo was used for analytic activities, including the adaptation of other procedures and the use of plans for solving problems. Pea and Kurland (1984) note that it is not clear that one could reliably identify the stage a student was in, and that the second stage could be an artifact of the teaching methodology used.

Instead, these researchers identified four levels of computer programming ability (Pea & Kurland, 1984). At level I is the program user, who can execute already written programs. Code generators at level II know the syntax and semantics of the most common commands of a language. They can read programs and understand what each line accomplishes, debug syntax errors, and write simple programs (usually with little pre-planning). The level III program generator thinks in terms of higher level units and knows sequences of commands that accomplish specific goals ("templates"). At level IV, the software developer writes complex programs that take full advantage of the capabilities of the computer and are intended to be used by others.

Linking Logo with problem solving

Noss (1984) distinguished among three *learning modes* that lead from explorations to problem solving. These modes are not developmental stages, in that children switch readily from one to the other. When introduced to a new idea, children need time to experiment and master the idea—to make sense of it. They often push the idea to extremes. In the second mode, exploring, children make connections between the new idea and established ones by incorporating the new ideas into their procedures. Exploring activities utilize programming tools to extend the power of the language. The third learning mode, solving problems, involves less experimentation and more goal-directed behavior. Noss demonstrated that significant learning takes place in each mode. This framework seems to be a useful way to see the value of the exploratory activity of the first two modes while balancing them with the more *reflective* problem solving of the third.

Computer programming does seem ideal for encouraging problem solving. Linn (1985) cautions, however, that research shows that much learning is domain specific. She describes a *chain of cognitive consequences* that suggests how links between problem solving in programming and problem solving in other disciplines may be forged. The links in the chain are as follows:

Learn the language features. These are the primitives (e.g., FORWARD BACK IF). Too many courses and textbooks spend most of their time teaching at this level (only). Logo, with it's small but extensible set of features, is ideal for quickly mastering fundamental language features and using them for problem solving.

Learn to design programs to solve problems. This has two components. The first is to develop a repertoire of templates, stereotypic patterns of code. Research shows us that experts organize their knowledge of programming into templates. (Templates were discussed in the previous column.) The second component is to develop procedural skills. These skills are used to combine templates or language features to solve problems. They include planning a solution, testing the plan, and reformulating the plan until it succeeds. Again, research has shown that experts spend a great deal of time planning solutions. Novices tend to "jump right in" and begin to write code. These procedural skills thus can and should be taught.

Learn problem-solving skills applicable to other formal systems (e.g., Newtonian mechanics). These include generalized templates (e.g., a general sorting routine), and generalized problem-solving strategies (e.g., the metacomponents

discussed in several previous columns applied to a variety of problems). Students at this level would see the connections between testing a program to be sure it can manage possible division-by-zero cases and testing the solution to a measurement problem in algebra to check that the answer does not require negative quantities.

Research indicated that teachers of successful high-school students explicitly encouraged students to master each link in the chain. They discussed templates and problem-solving strategies (e.g., debugging). They insisted that students make plans and they discussed errors and misunderstandings.

One even has to teach the "style" of programming. Next month's "extra for experts" column will be dedicated to this neglected aspect of programming.

Final Words

These attempts to delineate stages or learning modes serve as useful frameworks and provide us with insights into children's learning. However, they are sensitive to the educational context. For example, Kull's steps might help a primary grade teacher to construct reasonable expectations and to provide appropriate assistance. However, if support programs which encourage immediate use of procedurality are utilized (Clements, 1983-84; Clements & Nastasi, in press), children would not necessarily follow these steps. In addition, this body of work is often based on anecdotal observation and intuition. More work, involving increased specificity and assessment reliability, is needed if developmental stages are to be identified and validated. Teacher-researchers might make a substantial contribution to this effort.

References

Clements, D. H. (1983-84). Supporting young children's Logo programming. *The Computing Teacher*, *11*(5), 24-30.

Clements, D. H., & Battista, M. T. (in press). *Logo-based geometry curriculum: K-6*. Morristown, NJ: Silver Burdett & Ginn.

Clements, D. H., & Nastasi, B. K. (in press). Computers and early childhood education. In T. Kratochwill, S. Elliott, & M. Gettinger (Eds.), *Advances in school psychology: Preschool and early childhood treatment directions*. Hillsdale, NJ: Lawrence Erlbaum.

Fein, G. G., Scholnick, E. K., Campbell, P. F., Schwartz, S. S., & Frank, R. (1988). Computing space: A conceptual and developmental analysis of LOGO. In G. Forman & P. B. Pufall (Ed.), *Constructivism in the computer age* (pp. 105-127). Hillsdale, NJ: Lawrence Erlbaum.

Gregg, L. W. (1978). Spatial concepts, spatial names, and the development of exocentric representations. In R. S. Siegler (Ed.), *Children's thinking: What develops* (pp. 275-299). New York: John Wiley & Sons.

Howe, J. A. M. (1980). Developmental stages in learning to program. In F. Klix & J. Hoffmann (Ed.), *Cognition and memory: Interdisciplinary research of human memory activities*. Amsterdam, NY: North-Holland.

Kull, J. A. (1986). Learning and Logo. In P. F. Campbell & G. G. Fein (Ed.), *Young children and microcomputers* (pp. 103-130). Englewood Cliffs, NJ: Prentice-Hall.

Linn, M. C. (1985). The cognitive consequences of programming instruction in classrooms. *Educational Researcher*, *14*(5), 14-29.

Noss, R. (1984). *Children learning Logo programming. Interim report No. 2 of the Chiltern Logo Project*. Hatfield, England: Advisory Unit for Computer Based Education.

Pea, R. D., & Kurland, D. M. (1984). On the cognitive and educational benefits of teaching children programming: A critical look. *New Ideas in Psychology*, *2*, 137-168.

Roberts, R. J., Jr. (1984). *Young children's spatial frames of reference in simple computer graphics programming*. Unpublished doctoral dissertation, University of Virginia.

Douglas H. Clements is an associate professor at the State University of New York at Buffalo. He is co-author of the *Logo-based Geometry Curriculum*, to be published by Silver, Burdett, & Ginn. His most recent book, *Computers in Elementary Mathematics Education* was published by Prentice–Hall in 1989. His recent research has dealt with the effects of certain Logo environments on children's metacognitive ability, creativity, and geometric conceptualizations.

Douglas H. Clements
State University of New York at Buffalo
Department of Learning and Instruction
593 Baldy Hall
Buffalo, NY 14260.
CIS: 76136,2027 BITNET: INSDHC@UBVMS

Global Logo Comments

Edited by Dennis Harper
University of the Virgin Islands
St. Thomas, USVI 00802

Logo Exchange Continental Editors

This month's column comes to us through a frequent *Logo Exchange* source in Finland, Hannu Korhonen of the University of Jyväskylä. He happened to be in China recently, and despite the turmoil, had the opportunity to visit the University of Beijing and discuss Chinese and Finnish Logo experiences. Here is his exclusive report.

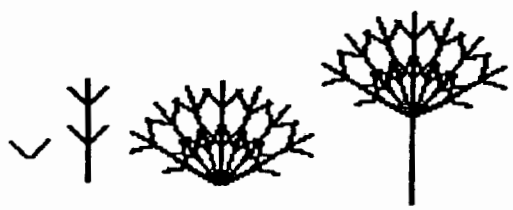
In Beijing, an electronic turtle obeys the command

```
CHONGFU 4 [QIAN 60 YOU 90]
```

as kindly as her American cousin in Eugene, Oregon, obeys the command

```
REPEAT 4 [FORWARD 60 RIGHT 90].
```

A person who understands Logo need not understand very much Chinese in order to understand the meaning of Chinese turtle graphics commands QIAN, HOU, ZUO and YOU. One can conclude the meanings from



and the corresponding program

```
TO YE
 Z 45 Q 10 H 10 Y 90
 Q 10 H 10 Z 45
END
```

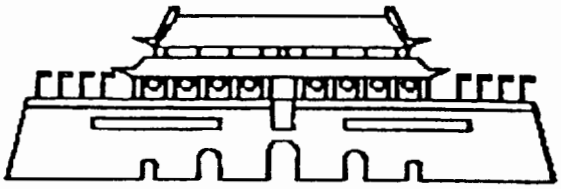
```
TO ZHI
 Q 15 YE Q 15 YE
 Q 10 H 40
END
```

```
TO CONG
 Z 60 CHF 7 [ZHI Y 20]
 Z 80
END
```

```
TO SHU
 CONG H 60 Q 60
END
```

The names of the procedures YE, ZHI, CONG, and SHU mean a leaf, a branch, a bunch and a tree.

A basic Logo textbook in the school of the Beijing University is *Qing-shaonian jisuanji Logo yuyan (Logo computer language for the youth)* by Mr. Zhang Wan Zeng and his colleague Ms. Chen Xu Lin. The book consists of a multitude of examples of turtle graphics. However, there are some more complicated procedures; such as the map of China and the Gate of Heavenly Peace or Tian-an-men.

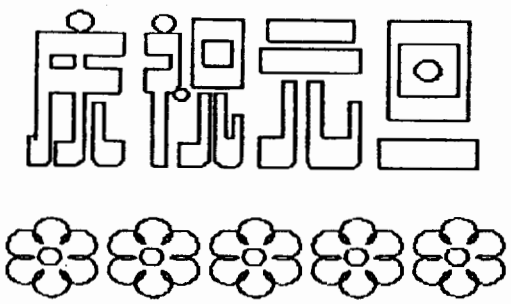


Most of the programs are made using merely the commands for proceeding and turning, subprogram calls, and a little recursion. Some of the procedures may thus be considerably long for elementary exercises, more than 100 commands with parameters. But the results of the programs are amazing and admirable.

Logo has been used in Beijing since 1984. The children work with the computer for a lesson a week starting in the fourth grade. The themes are mostly graphical. The objectives are learning to use a computer and developing algorithmic thinking.

Another, more advanced Logo book used in the school is *Zhongxiaoxue Logo yuyan jiaocheng (A Logo course for secondary schools)*. In this book the topics are progressing quite rapidly via the usual flowers, stars, and spirals to more advanced recursion structures, the variants of a binary tree, and such curves as C- and dragon curves.

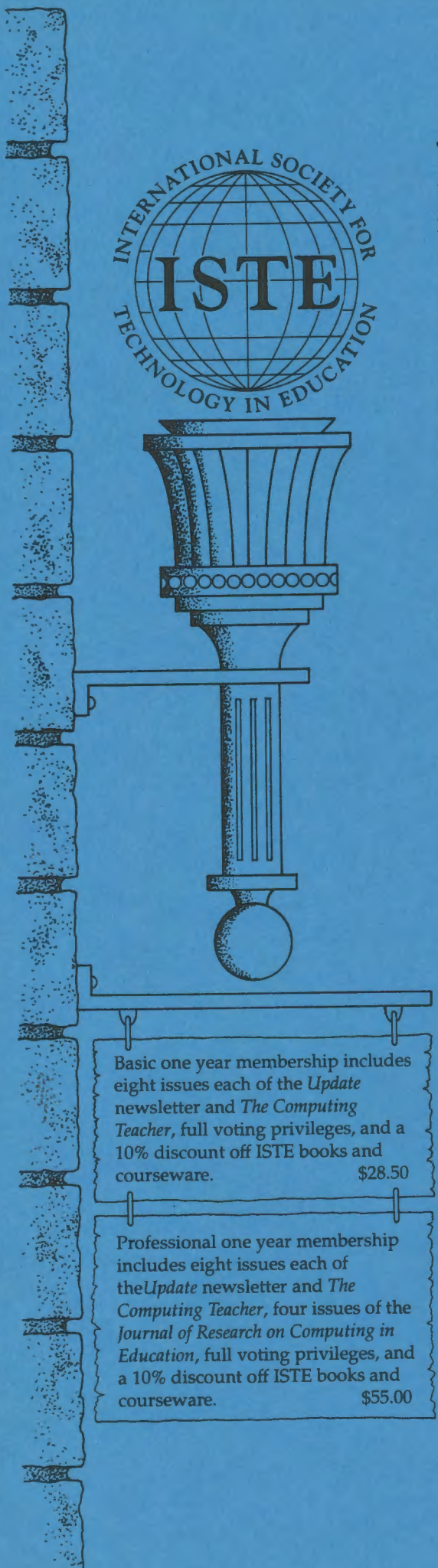
In the most advanced sections there are elegant flower compositions, computer calligraphy (Happy New Year),



and adorable animal shapes.

The choice of themes and the grandeur of the modifications demonstrate enviable insight and diligence. Furthermore, they show how esthetics and programming can shake hands with each other.

The Apple MIT-Logo has been extended in the electronics factory of Beijing University by Chinese characters and commands in pinyin Chinese. Chinese characters have been realized in assembler language code because of the speed considerations. Furthermore, there is a utility program for immediate changing of colors in the same way as in many graphics programs. A third application allows for the possibility of getting digitized video pictures processed with Logo, which the Chinese feel is an essential step toward advancing Logo into a multiusable language.



The ***International Society for Technology in Education*** touches all corners of the world. As the largest international non-profit professional organization serving computer using educators, we are dedicated to the improvement of education through the use and integration of technology.

Drawing from the resources of committed professionals worldwide, ISTE provides information that is always up-to-date, compelling, and relevant to your educational responsibilities.

Periodicals, books and courseware, *Special Interest Groups, Independent Study* courses, professional committees, and the Private Sector Council all strive to help enhance the quality of information you receive.

Rely on ISTE support:

- *The Computing Teacher* draws on active and creative K-12 educators to provide feature articles and carefully selected columns.
- The *Update* newsletter reaches members with information on the activities of ISTE and its affiliates.
- The *Journal of Research on Computing in Education* comes out with articles on original research project descriptions and evaluations, the state of the art, and theoretical essays that define and extend the field of educational computing.
- Books and courseware enhance teaching materials for K-12 and higher education.
- Professional Committees develop and monitor policy statements on software use, ethics, preview centers, and legislative action.
- The Private Sector Council promotes cooperation between educational technology professionals, manufacturers, publishers, and other private sector organizations.

It's a big world, but with the joint efforts of educators like yourself, ISTE brings it closer. Be a part of the international sharing of educational ideas and technology. Join ISTE.

Join today, and discover how ISTE puts you in touch with the world.

Basic one year membership includes eight issues each of the *Update* newsletter and *The Computing Teacher*, full voting privileges, and a 10% discount off ISTE books and courseware. $28.50

Professional one year membership includes eight issues each of the*Update* newsletter and *The Computing Teacher*, four issues of the *Journal of Research on Computing in Education*, full voting privileges, and a 10% discount off ISTE books and courseware. $55.00

ISTE, University of Oregon,
1787 Agate St., Eugene, OR 97403-9905.
ph. 503/346-4414