4

LOGO

A learning environment on your Home Computer.

LAREDO

The History of LOGO

LOGO—a powerful, high-level computer language designed for educational purposes especially as a programming language suitable for young children—is now available on Texas Instruments' TI-99/4A Home Computers. For more than a dozen years, the LOGO Group at the Massachusetts Institute of Technology has been developing the LOGO language and related computer programming activities. Under the leadership of MIT Professor Seymour Papert, LOGO activities have been used with children as young as nursery school age, with MIT undergraduates, and with many students of all ages in between. The philosophy of LOGO's developers has been: "No threshold, no ceiling." A beginner can make the computer do something meaningful and interesting in the very first programming session. Yet at the other extreme, LOGO is suitable for very advanced programming projects.

The philosophy of LOGO has been derived primarily from two sources: The developmental theories of the late Swiss psychologist, Jean Piaget (with whom Seymour Papert worked for several years before coming to MIT), and ideas from a modern scientific field called Artificial Intelligence. From Piaget comes the idea of creating learning environments in which most of what children learn can occur naturally—in the same way children learn to speak their native language, walk or run, and play ball. From Artificial Intelligence comes ideas about ways to use programming languages to aid thinking and problem-solving. Programming a computer in LOGO is seen as the act of teaching the computer a set of new commands, based on what it already knows how to do. Each user is, in effect, creating his or her own computer language, to suit his or her own purposes. Readers interested in learning more about these ideas should read Mindstorms, a recent book by Seymour Papert, in which he develops and extends the vision of the relationship between computers and learning that led to his development of LOGO.

LOGO activities are designed to allow use of the computer in a way that is personally meaningful to the user. Activities developed by the MIT LOGO Group have included using a computer to control the behavior of a robot turtle, to draw pictures and explore geometric environments on a TV screen, to create computer animations, invent interactive computer games, compose, arrange, and play music, and produce "poetry." The best known LOGO activity is using a simulated robot turtle on a TV screen to produce geometric designs and cartoon-like drawings. Hundreds of children have learned computer programming and problem-solving skills and developed mathematical expertise while writing programs for the turtle.

The LOGO language includes commands to make the turtle move and draw pictures. A student drawing with the turtle can make it move around on the TV screen by typing familiar commands such as FORWARD and BACK or RIGHT and LEFT. The information which lets the turtle to control the turtle is already present in their own body knowledge of how to move forward or back and how to turn right and left. Programming becomes an extension of something a learner already knows—rather than something requiring the mastery of an elaborate technical language or a complex coordinate system. The turtle becomes for the learner what Seymour Papert has called "an object to think with." Students using the computer as a programming tool become more aware of both their own body motion and the behavior of the computer.

The version of LOGO developed collaboratively by Texas Instruments and the MIT LOGO Group for the TI-99/4A includes an entirely new graphics environment called a "Sprites World." Sprites are small objects that can move rapidly around the screen, changing shape, color, speed and direction. Large numbers of sprites can appear at the same time to produce exciting animated designs or to be used as elements in programs to create video games. Because of its inherent attraction for so many people and because of the geometric and problem solving ideas embedded in it, the Sprites World promises to be one of the most exciting computer-based learning environments yet invented.

The World of the Turtle

Let's take a closer look at what actually happens when someone learns to program a computer using the LOGO turtle. The turtle responds to simple commands typed at the

keyboard: FORWARD 100, BACK 50, RIGHT 90, LEFT 45, etc. FORWARD 100 moves the turtle forward "100 turtle steps," drawing a line on the TV screen in the process. LEFT 45 makes the turtle rotate 45 degrees to its own left. People learning LOGO find it natural to "identify" with the turtle, imagining themselves going through its motions as it carries out a particular task. At the same time, controlling the turtle becomes a metaphor for controlling the computer itself. Like the turtle, the computer responds to an ordered series of command, and "TO" procedures that are defined series of commands.

The ways in which the actions of the turtle can lead to geometric designs, as well as the method used to define procedures, is illustrated in the following simple examples. The turtle can draw a square by repeating the commands FORWARD 100 RIGHT 90 four times. A procedure can be defined by choosing a name (BOX, for example) and typing in a series of commands in order.

```
TO BOX
FORWARD 50
RIGHT 90
FORWARD 50
RIGHT 90
FORWARD 50
RIGHT 90
FORWARD 50
END
```

To execute BOX enter the following:

TELL TURTLE
BOX

When the new command, BOX, is typed, the turtle immediately draws the shape shown in the figure. The small triangle shown in the figure represents the turtle by showing its position and heading. A similar procedure, TRI, can be defined as follows:

```
TO TRI
FORWARD 50
RIGHT 120
FORWARD 50
RIGHT 120
FORWARD 50
END
```

To execute TRI enter the following:

TELL TURTLE
TRI

A student who has defined procedures such as BOX and TRI is beginning to "teach the computer" his or her own private language. BOX and TRI can now be used in the same way as other LOGO commands. They can be used to create other drawings such as a simple "house" or an abstract geometric "flower."

This approach to geometry and programming provides the basis for a rich universe of activities known as Turtle Geometry, which includes cartoon drawings (simple and complex), geometric designs, mathematical theory building, and computer games. Extensions of Turtle Geometry have proven fruitful when used with advanced high school students or MIT undergraduates. The universe of Turtle Geometry provides a conceptual framework for such aspects of mathematics as the relation between shapes and angles, coordinate systems, positive and negative numbers, the use of variables, symmetry and similarity, and even calculus and differential geometry. The computer programming involved in beginning LOGO activities can include procedures and subprocedures, the naming of procedures and variables, pro-

cedural hierachy, recursion and iteration, the use of conditional logic, and the development of problem-solving strategies.

Within the universe of Turtle Geometry, there is room for different students working individually to create their own sub-universes or microworlds. They can do this with their own limited (but expandable) sets of concepts and related activities and projects. To teach LOGO is really to help learners create, explore, and extend their own microworlds.
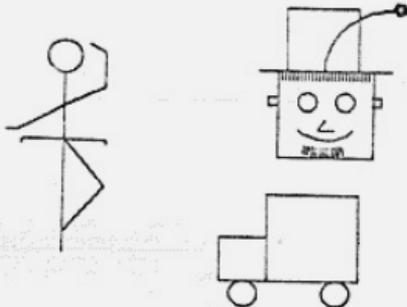
I have used turtle geometry as an example of what can be done with LOGO because it is easy for a reader to visualize the commands and to see how they lead to procedures that produce the results in the pictures—just as it is for young children. Children learning LOGO have actually carried out many other types of projects as well: moving turtles, finding their way around race-tracks or mazes, animated cartoons, interactive computer games such as Nim or Tic-Tac-Toe, programs which generate sentences or poetry (or even play Mad-Libs), and programs to translate English into Morse Code, or vice-versa. As LOGO becomes available to owners of TI-99/4A computers, I hope that these pages can be a forum for describing your LOGO projects. Since there will soon be more LOGO users than ever before, we can expect more and different LOGO projects to emerge. One of the best ways to build the culture of LOGO is for users to share project ideas through the pages of books such as this or magazines such as 99'er Home Computer Magazine.

Although TI LOGO is a recent entry to the LOGO World, a prototype version has already been tested with hundreds of students between the ages of three and nine at the Lamplighter School in Dallas, Texas, and by students in fifteen elementary and junior high schools in New York City. Using the Sprites World of animated graphics activities, these students are busily creating a new universe of LOGO activities to delight and educate a new generation of computer users. In an age in which computers are omnipresent in society, and in which universal computer literacy is a pressing national need, computer-based learning environments like LOGO have become essential to the process of growing up literate in the last decades of the twentieth century.

The Lamplighter LOGO Project

"**A** child is not a vessel to be filled, but a lamp to be lighted." The quote from Alexandrov is on the plaque outside the Lamplighter school. That sign advises any visitor that the school is very unusual.

The curriculum at Lamplighter is individually tailored to meet the needs of each student. Individualization is applied in science, language arts, math, drama, music, art, French, and physical education. The Lamplighter is strongly supported by the parents of its students and by its alumni, with graduates of Lamplighter frequently dropping by to see their former teachers. Such alumni loyalty might not be considered unusual, except that the Lamplighter classes begin with preschool (age 3) and end with the fourth grade level!

The physical arrangement of the school reinforces its approach to learning. Classrooms have only three walls; the fourth side of each class opens onto an airy, bright shared space. Class rooms are clustered around these shared-spaces by grade-level. Inside each classroom there are tables and chairs for writing work and, on one side, a small tiered well which is used for many other activities (e.g., reading, French, music, or story telling). The staff, the facilities, the students, and the parents all contribute to make Lamplighter a very special private school.

Lamplighter has been a leader in the use of new technology for learning. Calculators, *Speak & Spells*, Systems 80 units, and Little Professors are abundant throughout the school. Students regularly use these learning tools and other learning games found in the shared spaces. Teachers make extensive use of slides, films, and video and audio tapes. When Mr. Erik Jonsson (co-founder of Texas Instruments and Lamplighter Board of Directors Chairman and benefactor) first proposed introducing computers into Lamplighter, his idea was well received. Mr. Jonsson had earlier been in contact with Dr. Seymour Papert

of the Division for Study and Research in Education (DSRE) at MIT, and found the LOGO language and philosophy of learning intriguing. Papert's initial explanation of LOGO allowed students to program computers and not vice-versa, enjoyed a favorable reception from the Lamplighter faculty. Later, as Papert elaborated on the LOGO philosophy, it became clear the LOGO was very much in accord with the philosophy and practice of Lamplighter.

In the fall of 1978, Papert and several others from DSRE made a series of preparatory visits to Lamplighter to arrange for the introduction of LOGO to the school. The plan was to begin LOGO training for first the faculty and then the students by using the Digital LSI-11 LOGO (in use at the Brookline, Massachusetts, project) and later, bring TI LOGO into the school as it developed.

Shortly after the first visit by Papert, Lamplighter rented the first of two LSI-11's that were to be used in the initial core of Lamplighter faculty (representing nursery school, second grade, third grade, and fourth grade) became familiar with LOGO. This "Computer Group" then began working with third and fourth grade students. Shortly thereafter, a second LSI-11 was rented, and by the end of the spring term every third and fourth grade student had had at least one hour of LOGO instruction on a computer.

The third and fourth graders considered it a treat to work on the computer—partly because these special computer activities allowed them to miss classes, and partly because they genuinely enjoyed working with LOGO. One student's remark reflects the sentiments of many of these pupils. After he had spent an hour working at figuring line lengths, turn angles, and sections of arcs in order to construct a computer picture of a cat, he thanked me for "getting out of math class."

In the summer of 1979, the Computer Group was expanded, and two workshops were held to refresh the teachers' memories. Subsequently, a 10 day workshop at MIT introduced the teachers to more elaborate LOGO programming and allowed them to participate in discussions on the relationships between learning and LOGO. Then, as the new school year started, the teachers were really surprised to discover how little the fourth graders had forgotten about LOGO. These students generally recalled all of the commands they had learned three months earlier—even though they had had no contact with LOGO in the interim!

Midway through the fall semester of 1979, several early prototypes of TI LOGO were tested at Lamplighter and revised by the MIT LOGO laboratory personnel in consultation with Lamplighter and Texas Instruments. In January 1980, the pace of computing at Lamplighter accelerated as an updated version of TI LOGO was implemented on the TI prototypes. By the end of January, a dozen prototypes were in use at Lamplighter, and a very few students continued to use the LSI-11 LOGO. Most pupils, in fact, switched to the TI prototypes even though that meant re-learning much of LOGO.

In the middle of the spring semester, a few more prototypes arrived and all the machines were upgraded to a later version of TI-based LOGO. Before the school year ended, all of the third and fourth graders had had at least one hour on the new machines. One of the rented LSI-11's was then returned (though few noticed its departure). At that time, several fourth graders were writing elaborate programs which made use of recursion to create "movies" or "rainbows" (changing colors), or elaborate scenes. Some students were so taken with LOGO that their parents happily bought them their own computers (at that time, TI LOGO was not yet commercially available); other students became enthralled with their ability to produce perfectly printed letters and numerals on a keyboard and later received typewriters as presents from their parents.

By September 1980, a total of 50 TI LOGO prototypes were in operation at Lamplighter. The version of LOGO on these units was very close to that which TI is now marketing. Then, late in the fall, the second LSI-11 was returned, but its loss went completely unnoticed because all of the faculty and student interest was already focused upon the TI LOGO prototypes. Since September, the Computer Group has continued to work individually with third grade students. In addition, the rest of the faculty is being trained in LOGO, and it has been introduced into all of the classes as part of the regular school curriculum.

The teams at each grade level decided the best way to introduce LOGO into their classes and worked out various procedures for that introduction. For example, one teacher developed special simplified LOGO programs for the preschool children which required less typing in order to produce interesting effects. And personalizations of LOGO constructs made LOGO easier for first and second graders to understand. Currently, students can be seen at every shared-space LOGO machine during lunch-hour, before school, after school, and whenever other school activities are completed. For the rest of the semester, LOGO will be used in class by the teachers as they feel it is relevant for their lessons and will continue to be available (as are the other learning aids) to students during free periods.

The Lamplighter LOGO project was not intended to be a formal experiment. Since there are no control groups, strong causal claims for LOGO's effects are inappropriate. Several cognitive and psychological assessments, however, were made at the beginning of the project and will be made again at the conclusion of the present school year. And, there already have been some indications of student attitude and behavior change. This is best exemplified by the way in which the pupils express their keen interest in acquiring new LOGO knowledge.

It's always interesting to observe what motivates children to learn. Because LOGO is so extensive, Lamplighter teachers find it impossible to show students all the commands in the initial sessions. As a result, students have taken the discovery of more LOGO commands as a sort of treasure hunt and this new, "unauthorized," LOGO information is disseminated through an "underground network" among the students. During a training session in which teachers were learning to use MAKESHAPE (the LOGO command with which users make their own shapes on a 16 x 16 grid), some students were secretly watching them. Shortly afterward, a hand-copied "underground" LOGO manual with clear and concise directions for the use of MAKESHAPE was found on the floor of a classroom; at the same time, a number of students began using MAKESHAPE.

Students have discovered other information accidentally. One student typed MC instead of MS for MAKESHAPE; this put him in MAKECHARACTER mode. In this mode, LOGO users can modify old characters, or make new characters. The student promptly shouted out his discovery to his classmates, who quickly confirmed his results and spread the news. New information has diffused from grade to grade or class to class or from parent to child in a similar manner.

Sharing among peers is the overwhelming response of Lamplighter students to new LOGO information. Pupils eagerly and proudly explain their accomplishments in LOGO. At first, however, there were a few exceptions. A couple of students were secretive about some LOGO information and effects. One student made the screen's background color black so that no one could read what he typed; another tried to sell LOGO programs to his classmates! After they discovered that other students could find different ways to achieve the same effects and were willing to share, they started sharing as well.

In at least one case, LOGO seems to be responsible for a major behavioral change. Late last year, a fourth grader who had not been performing well academically, and who had been somewhat disruptive in class, started programming in LOGO. As he played on the computer, his typing became very fast (QWERTY keyboards are quite properly regarded by the Lamplighter children as a stupid arrangement with which they reluctantly work), and his program became sophisticated. He was heard to remark, "I can't believe how fast my fingers are typing." He also could not believe how much fun school had become. Not only did he do well with LOGO, but he also became an attentive, productive student.

Figure 1.

```
TO LAZY8
FORWARD 4
RIGHT 10
TEST HEADING 0
IF LAZY8
FORWARD 4
LEFT 10
END
```

TELL TURTLE
LAZY8
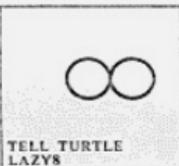
Comment:

TEST checks the heading of the Turtle. If it's not 0 (North), the Turtle continues to draw the LAZY8s.

After finishing the right-hand circle, the heading becomes 0 and the left-hand circle is drawn.

To really understand why the left-hand circle ever gets completed, you have to know something about microprocessors and stack operations. In keeping with the scope of this section, however, a simple anthropomorphic explanation will have to suffice at this time. Other sections will take an in-depth look at the technical aspect of the language.

Think of the job of drawing the LAZY8s as being given to a group of little workmen inside the computer. One workman carries out the first four lines then decides he needs a rest before continuing. Notice that in his initial contract TO LAZY8 he has agreed to eventually carry out the FORWARD 4 and LEFT 10 specifications. The work must go on while he rests, so he subcontracts out the next stage to another little man. This workman also carries out the first four lines, then he too decides to rest. So before he gets to the FORWARD 4 LEFT 10 tasks, he decides to subcontract out the balance of the work on the right-hand circle. This process goes on with enough little workmen (36 in this case) until HEADING = 0. At that time, the last little man carries out his FORWARD 4 and LEFT 10 tasks, and gives the job responsibility back to the next-to-last workman who also carries out his remaining FORWARD 4 and LEFT 10 tasks. This reverse process of finishing the last two tasks and relinquishing responsibility goes on until the original contractor finishes his original job with a single FORWARD 4 and LEFT 10, thus completing the left-hand circle in the LAZY8—Ed.

At present, most of the third and fourth graders—and even some of the first and second graders—are writing LOGO programs. This includes some fairly sophisticated programs which use recursion and the concept of state transparency. A few children even acquired the skill of using *subprocedures*—i.e., breaking a complex program down into its several component parts. This is one of the most important features of procedural languages such as LOGO. Most students had discovered recursive programming, or "cursives" as a few called it. In recursive programs, one of the program lines calls for a new stack to execute the program again. You do this by including the name of the program within the program itself. All the recursive programs written by the students, however, had the recursive step in the last line. [When the recursive step occurs in the last line before END, the procedure is said to have "tail-end recursion." For an example of somewhat more sophisticated usage, see the LAZY8 procedure in Figure 1—Ed.]

A number of programs produced exciting video scenes. In EXPLODE, 32 differently colored balls splay out from the center of the screen before repeating the entire procedure. One third grader saw how he could place a program which printed a message inside EXPLODE, and thus combined recursion and subprocedures. RAINBOW had one or more sprites continuously change colors for an attractive visual effect. There were also programs which had the TV monitor take on a series of sixteen colors, and programs which changed the background of the screen to black and created unusual perceptual illusions by shooting light-colored shapes across the screen. Some even had jets, rockets, or airplanes spouting fires from their engines.

Other children wrote programs which put shapes together to create scenes, such as a home with a car driving down the street in front of the home. Most students had written utilitarian programs like VANISH (Figure 2) which caused the sprites to move off screen, take on the clear color, carry an empty shape, and which caused all the printing to be cleared from the screen.

Figure 2.

```
TO VANISH
TELL ALL
CARRY 0
SETCOLOR 0
SETSPEED 0
SETHEADING 0
END
```
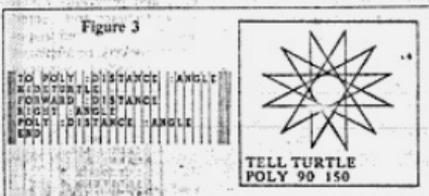
After spring break, several things happened which caused a quantum leap in the computer work of the students. First, the children were shown how to save their programs and shapes on cassette tape. Until then, the students had to write in their computer notebooks anything they wanted to save. That meant that any elaborate shape had to be reproduced on a grid in an arduous manner, and long programs or complex programs required a very long time typing. (Remember these children are elementary pupils with little typing experience before computers!).

Students had not used much of their work as foundations for future work simply because loading the old material took so much of their time. Now, with the recorders, they could use and improve each session's programs just by typing and playing back a cassette. Also, they could design and SAVE complex shapes instead of seeing them lost when the computers were shut off.

The children were also shown the TELL TURTLE mode. This opened up all of the turtle geometry features of LOGO.

(Turtle geometry is such a powerful idea that some Pascal systems have adopted it.) This newly acquired mode, coupled with the previously learned SPRITE MODE, allowed the students to produce many interesting programs and visual effects. As a result of these new developments, many of the students soon exhibited a feeling of mastery over the computers.

In the final eight weeks of school there was an exponential explosion in the complexity of the students' programs and in their ease with the machines. They quickly learned to use variables as inputs, and consequently "discovered" the famous turtle geometry POLYgon program which can generate any regular polygon. (See Figure 3.) Then one student found that changing the angle of the turn on each recursion could produce beautiful patterns—including a striking nested curl in a star pattern. Many students now began putting programs together in subordinate and superordinate structures. Programs contained the unique LOGO controls of TEST, IFT, and IFF, as well as the conditionals IF-THEN-ELSE, plus BOTH and EITHER for conjunctive and disjunctive branching. One of the third graders wrote a CAI (Computer-Assisted Instruction) program to quiz his first grade friends on addition facts using these control commands! He then added visual displays of the addends, and encouraging remarks when a student made a mistake, or a colorful scene as a reward for the correct answer.

Figure 3

```
TO POLY :DISTANCE :ANGLE
KIDSTURTLE
FORWARD :DISTANCE
RIGHT :ANGLE
POLY :DISTANCE :ANGLE
END
```

START TURTLE
POLY 90 150

Using combinations of several user-drawn shapes, students began constructing very elaborate composite pictures. One third grade student also discovered how to change the characters associated with each console key [by reshaping the characters on a grid "tile" with the MAKECHAR primitive—Ed.], and decided to tease the teacher. She replaced the 3 with a 2, and then called a teacher for a demonstration. While instructing the computer to print 3 + 3 (which now looked like a request for the sum of 2 + 2), she remarked to the teacher: "Look how dumb this computer is...it doesn't know 2 + 2."

The activity among the third grade students was exciting to witness. One began programming dramas in which text was printed at the bottom of the screen while the story was enacted in SPRITE and TELL TURTLE modes at the top of the screen. One other third grader was so intrigued by the space shuttle's landing that on the same afternoon of the landing, he began working on a shuttle program. First, he used MAKESHAPE to construct a faithful replica of the shuttle, complete with USA monogram, black-and-white coloring, and auxiliary rocket engines. Then he worked for part of the afternoon and a little of the next morning to write and debug his programs. His final superprocedure

launched the shuttle with flames shooting from the engines, jettisoned the auxiliary tanks, orbited the shuttle among planets in outer space, returned the shuttle to a dry lakebed runway, taxied it to the end of the runway, and stopped it for a perfect landing. His programs are shown here in Figure 4.

Figure 4

```
Note:
BG = BACKGROUND
FD = FORWARD
SC = SETCOLOR
SH = SETHEADING
SS = SETSPEED
```

[Note: Listings of TI LOGO procedures are just that—listings of procedures. There's no way to print out a transcription of the data needed to MAKESHAPE and MAKECHAR as can be done with the HEX Codes in TI BASIC and Extended BASIC. The only way to show the graphics that a program contains is to show it as it is drawn on a series of "tiles" on the grids that appear on screen when the shapes and characters are first designed. This is similar to CHARDEF routine in Programming Aids 1. The listing of the Space Shuttle program was included (without the titles) in this article to demonstrate the simplicity of the language structure.—Ed.]

The gains made by the Lamplighter children with LOGO have indeed been impressive. They confirm Papert's dictum [Mindstorms, Seymour Papert, Basic Books 1980] that children should program computers and not vice-versa. It's obvious that LOGO has indeed furthered Lamplighter's goal of igniting the imaginations and intellects of its children. But more importantly, LOGO has the potential to fire up imaginations everywhere.

Who is LOGO For?

Its not just for
Turtles anymore . . .
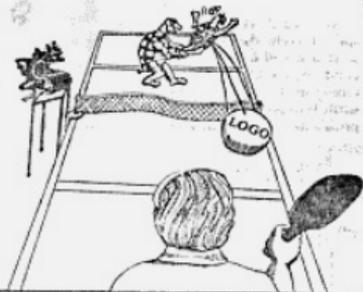
Recently the question of LOGO's relevance for children and its relevance for adults has been stated as an implicit either/or issue. That the issue ever arose means that people (including me) who write about LOGO have not done their jobs as fully as they should. Perhaps the notion that LOGO was just for children developed because of the total attention children invest in LOGO. The position that LOGO is too complex for children may have arisen because published programs seem magic unless one actively explores them (including seeing what happens when the programs are changed). Presenting a program as a *fait accompli* to be copied, run, stored, and used like any other software is contrary to the philosophy of education behind LOGO.

LOGO is for humans. When Papert asked me if I felt comfortable with my LOGO, I said that LOGO is like a hologram—when you grasp just the smallest part of it, you have a small, but complete picture; and later as your understanding grows, you still have a complete picture, albeit larger. From that perspective, people can always learn more from LOGO and do more with LOGO even though they are able to use LOGO after the briefest of introductions. This feature of LOGO is what Papert alludes to in his slogan, "Low threshold, no ceiling."

The LOGO slogan invites empirical verification. In my self-observations and studies of other adults, I have noticed that there are common, identifiable LOGO-developmental stages. Among these are the discovery of heuristics (i.e., powerful ideas), improved understanding of numbers, appreciation of angles and heading, and awareness of states and state independence. Probably the greatest gain people share in working with LOGO is the realization that one can find out on the computer, rather than ignoring the question or looking the answer up somewhere. This is so obvious that it might appear trivial; it is not. All learning theorists agree that active learning is preferable to passive learning. This presents a dilemma for those writing about LOGO: How do you capture the open activity of a LOGO learning enterprise in a closed article?

The purpose of *this* article, however, is to reflect the development of a LOGO game, and in that development show how an apparently complex program is child's play, even for adults. At the same time, I hope that the development will point to variations and will entice you into active exploration. The program was initiated by a student in a course I taught.

The program was supposed to be a "Pong" type game. As you follow its growth, find the point, if there is one, where the program stops being a children's program. The game begins not as a program, but simply as a collection of conditions.

```
TELL 0
CARRY BALL
SETCOLOR 15
SETHEADING 90
SETSPEED 30
HOME
```

These commands set a ball speeding left-to-right across the middle of the screen.

The idea grows into a program as the ball is set to "bouncing" off left and right boundaries. This is accomplished any of several ways:

```
TO BOUNCE1
TELL 0
TEST XCOR > 85
IFT RIGHT 180
TEST XCOR < -85
IFT RIGHT 180
BOUNCE1
END
```

But BOUNCE1 sometimes doesn't work—occasionally the sprite is "caught" at one end or the other. What happens is that the sprite slips past one of the boundaries (e.g., the computer is at line 2 of the program as the sprite moves left through X coordinate equal to -85); by the time the computer reaches line 4, the sprite is well left of X coordinate -85. Then the computer turns the sprite right 180 (a right 180 functions equivalent to a left 180). Before the sprite can move beyond the -85 X coordinate, the computer checks line 4 again, turns the sprite 180 and sends it still further to the left. Of course, when the computer reaches line 4 a third time, the sprite is still left of -85; the poor sprite is stuck beyond the left-hand boundary! This bug could be eliminated with a second type of BOUNCE program:

```
TO BOUNCE2
TELL 0
TEST XCOR > 85
IFT SETHEADING 270
TEST XCOR < -85
IFT SETHEADING 90
BOUNCE2
END
```

that the CHECK program is invoked only if the ball is
beyond XCOR 85. Therefore, part of the scoring and noises
can be controlled after line 3 of BOUNCE2 by rewriting
the CHECK program:

```
TO CHECK
TEST PADDLETOUCH
IFT CALL P15 PLAYER'S SCORE
IF ERRAND [NOISE] NOISE FOR THE
IFT MOUSE ] PLAYER'S POINT
PLAYER'S POINT
IF XCOR > 119
[IFT CALL ] CP1 CP'S
[HIGHLIGHT COMPUTER'S SCORE
IF XCOR > 85
SHOOT BLEP FOR COMPUTER'S
POINT
TYPE (WORD SCORE 15 )PC 32
TYPE PS PC 33
TYPE (TRE COMPUTER'S SCORE 15
PC 80
TYPE CS PC 81
[ TO BE ADDED TO PREVENT
EXTRA SCORING ON EACH SERVE
END
```

```
TO NOISE
REPEAT 5 [BEEP WAIT 5 BOEEP W
]
END
```

It is necessary to set up an initial value for both the com-
puter's score and the player's score as was done with ·Y.
Since this is done just once, it belongs in SETUP. [The in-
itial score is 0 to 0—as in the proverbial "soothsayer's"
prediction or score before it begins. . . .] So SETUP is
revised:

```
TO SETUP
TELL R
CRQDY BALL
SETCOLOR BLUE
HOME
SETHEADING 90
SETSPEED 35
SETCOLOR MAX
SETY 0
SETCOLOR BLACK
SETHEADING 90
SETY 100 S
TELL 2
TELL P
SETCOLOR HALL
CALL 0 :PS
CALL 0 :CPS
END
```

This game, like most LOGO projects, is open-ended. It
could be altered so that a winner is named at a score of 21,
revised for two players, changed to use joysticks or changed
so that the ball has topspin. With each addition, it is
necessary to make sure that the initial conditions are
established only once, that procedures to be repeated are
placed inside a recursive program, and that there are no
Recursion Interface Bugs.

```
TO BOUNCE2
TELL R
TEST MOUSE 85
IFT CHECK
IF XCOR > -85
IFT SETHEADING 70
TEST YCOR > 100
IFT SETHEADING 135
TEST YCOR < -100
IFT SETHEADING 85
END
```

LOGO's POWERFUL SURPRISES!

PART 1: Language Structure and Syntax

LOGO was developed by Seymour Papert and his associates at the MIT Artificial Intelligence Laboratory in order to study the way people might learn in a computer-rich environment. It was designed to be a language so simple to use that a person could manipulate objects or concepts by just thinking about what he or she wanted to accomplish, and not have to worry about programming. Such a language might stimulate a person to explore, to learn, and to grow.

The idea was to provide certain *primitive* commands and operations that could be combined to form more *complex* commands and operations. These more complex ones could then be used exactly like the primitive ones. Thus it would be possible to construct a single command to accomplish anything that could be accomplished using the primitive concepts. Additionally, *recursion*—whereby a command could call and activate itself—was allowed.

LOGO is a relative of LISP, the list processing language used in artificial intelligence. LOGO and LISP share the capability of manipulating numbers, words (character strings without a space), and lists. A *list* is a recursively defined object: it is an ordered set of objects, each of which may be a number, a word, or a previously defined list. In LOGO, a *procedure* is represented by a list; there are commands to access a list that represents any procedure, and to define *new* procedures from lists which might be the result of some manipulation. Furthermore, a procedure may have inputs and may have an output, and is activated by specifying its name (a word) followed by its inputs (which may be numbers, words, or lists). Defined procedures as well as primitive commands and operations all have exactly the *same* syntax. This is why LOGO is so simple to use. Its power comes from its list processing capabilities.

I hope that the description given so far has made it apparent that LOGO is *not* just for children. Although LOGO

can be used in elementary ways, it is much more than FORWARD 20 RIGHT 90. LOGO is a language for all people who want to learn and expand their capacities.

The LOGO Turtle

The first experiments with LOGO were with junior high school students who could appreciate manipulation of words. Then a Turtle was created whose movements could be understood by very young people.

The Turtle was originally a robot that could be commanded to move about the floor. It had a pen which could be either up or down. In an experiment at the University of Pittsburgh Learning Center several years ago, one young person used LOGO to command the floor turtle to draw an alphabet of large letters. He also taught it to act like an airplane, and "fly" between cities on a large map. The plane had the possibility of going out of control, with the turtle going into a spiral and spinning on the floor. The turtle is now usually a small triangle on a terminal screen, but it can still do such things, albeit on a smaller scale.

At the youngest levels, LOGO is being used to teach a feeling for distances and angles. It works through college it is being used to advance a new subject in mathematics called "Turtle Geometry." Some interesting theoretical results have come about. (A wealth of examples and exercises is contained in *Turtle Geometry* by Abelson and diSessa, where procedures are expressed in a language almost exactly the same as LOGO). Recursive designs such as snowflake curves, space filling curves and trees are applications of LOGO's power.

TI LOGO

TI LOGO is marketed as a language for children, and it was a pleasant surprise to discover that TI LOGO has all of the list processing capabilities built into it. All the recursive designs presented in *Turtle Geometry* can be drawn. (The TI Turtle is, however, limited to 192 different

8×8 pixel character positions. Thus, if a figure is very dense, it can't be very large.)

The documentation that comes with TI LOGO doesn't make it easy to discover LOGO's power. Many of the commands needed for manipulating all but the simplest lists are not documented.

At this point, it may be helpful to briefly describe just what is available to a person who sits down to use TI LOGO. The TI Turtle is an object that lives on a coordinate screen with horizontal coordinates from −119 to +120 and vertical coordinates from −46 to +97. The bottom six lines of the screen are used for text. The turtle can be assigned a position, and "knows" where it is. The turtle can be assigned a heading (from 0 to 360 as the points of a compass) and knows its heading. Its heading can be changed by a given angle, and it can be moved a given amount either in the direction of or opposite to the direction of its heading. It can make a dot at any position. The pen can be down, up, or in "reverse" modes, and it can draw in any of 15 colors.

Unique to the TI version of LOGO are sprites—objects familiar to those with TI Extended BASIC. There are 32 sprites (numbered 0 to 31) with each assigned to a 16×16 pixel shape. Users may design and store 26 of these and can direct any collection of sprites to assume simultaneously an attribute such as shape, color, position, heading, speed, or velocity. The commands which control the turtle act similarly on the sprites. Motion is controlled by assigning a speed (in the current direction) or a velocity (horizontal and vertical components). Not only can attributes be assigned, but they can also be obtained as the output of operations because a sprite always knows its own number, shape number, color or number, position (on the full screen), heading, speed, and velocity.

Papert has described Velocity Turtles (which can have velocities) and Acceleration Turtles (whose velocities can be incremented). Sprites can be both. Using sprites we can even simulate Papert's "Dynaturtle"—an acceleration turtle which does not change direction when it is rotated, but changes velocity only by accelerating in the direction it is facing, thus obeying Newton's laws of motion. A dynaturtle therefore behaves like the ship in the popular Asteroids arcade game. The example procedures that follow this article will demonstrate a dynaturtle which can have the force of its "thruster" changed, and which can simulate an environment with friction.

TI LOGO also has 256 tiles (numbered 0 to 255) that can be given arbitrary 8×8 pixel designs. We can assign tiles foreground and background colors and position them anywhere on the 24×32 character screen or on the current print line. Console characters are tiles, the number of each tile being the ASCII code of the character. (Note: The Turtle records its trace using tiles, so simultaneous use of the Turtle and nonprinting characters is limited.)

Numbers, Words, and Lists

A number in TI LOGO is an integer from −32,768 to 32,767. Numbers can be added, subtracted, multiplied and divided (integer quotient), calculations being modulo 32,768. The restriction to integer arithmetic is a definite limitation, but the limitation is not serious for most applications.

A word is a character string without a space. A feature of LOGO, distinguishing it from other programming languages such as BASIC or Pascal is the capability of using a word simultaneously as (1) the name of a command or procedure, (2) a variable, and (3) data. For example, if the word X is to be used as the name of an action, X itself is used. When an object has been assigned to X, the object is denoted :X. The word X as data is denoted "X. Suppose that X has not been defined as an action and has not been assigned a value. LOGO will respond to X with TELL ME HOW TO X, to :X with :X HAS NO VALUE, and to "X with TELL ME WHAT TO DO WITH X.

A word can be assigned any kind of data—i.e., a number, word, or list as a value. This also distinguishes LOGO from BASIC or Pascal where the data type of a variable must be specified in advance. As a bizarre example, note that MAKE "MAKE and MAKE "MAKE [MAKE] assign to MAKE first the word MAKE and then the list whose single member is the word MAKE.

A list is the most powerful data object in TI LOGO and is denoted by a left bracket followed by its members, then a right bracket. Examples of lists are [ ], the null list; [HOW NOW BROWN COW], a list of words; and [REPEAT 4 [FORWARD 20 RIGHT 90]], a list whose members are a word, a number, and another list.

Data Manipulation in LOGO

Commands which are powerful in manipulating data include the following: FIRST(F), LAST, BUTFIRST(BF), BUTLAST(BL), SENTENCE(SE), FPUT, LPUT, NUMBER?, WORD?, THING?, THING, WORD, MAKE, RUN, TEXT, DEFINE. The last three are used to execute a list of commands, to access the list which defines a procedure and to define a procedure represented by a given list. These are powerful commands, but to be able to make use of them it is necessary to be able to construct lists whose members themselves are lists. The following key (undocumented) commands, FPUT and LPUT, are helpful here:

FPUT object list—outputs a list whose first member is object, and whose following members are the members of list.

LPUT object list—outputs a list whose last member is object and whose members all but the last are the members of list.

If object is a word or a number, the results of these commands are the same as SENTENCE object list and SENTENCE list object, respectively. But if object is a list, FPUT object list adds object to the beginning of list while SENTENCE object list adds the members of object to the beginning of list. This is a crucial difference, making possible the construction of arbitrarily complicated lists. The other commands in the above list which are undocumented are as follows:

NUMBER? object—returns TRUE if object is a number, and FALSE otherwise.

WORD? object—returns TRUE if object is a word, and FALSE otherwise.

THING? "name—returns TRUE if name has been assigned a value, and FALSE otherwise.

THING *name—returns the object which has been assigned to name, if name has a value.

WORD word1 word2—returns the word formed by concatenating word1 and word2. (Compare with SENTENCE, below.)

SENTENCE wordorlist1 wordorlist2—(a documented command), returns a list determined by the inputs. If an input is a word, that word is put in the list. If an input is a list, its members are included in the list.

Some of the undocumented commands were found by accident; others by studying the documentation for MIT LOGO. Still others were known to Jim Muller, president of the Young Peoples' LOGO Association (YPLA). We encourage readers to share other discoveries with us.

A Calculating Example

As a simple example, consider the problem of teaching LOGO to act like a calculator. If one enters 2 + 3, the response is TELL ME WHAT TO DO WITH 2 + 3. Here, desired output is 5, which is the result of executing PRINT ` + 3. The problem is solved by using SENTENCE to form the list [PRINT 2 + 3] and then using RUN to execute the list. A solution is the following:

```
TO CALCULATE
MAKE ": READLINE
TEST :D WORD
IFT RUN SENTENCE [PRINT] :T CAL
CULATE
END
```

After you enter CALCULATE, the computer accepts arithmetic expressions and prints out the resulting value until just ENTER is pressed. The recursion then "unwinds," and the procedure stops.

The power of a list processing language such as TI LOGO becomes apparent the more you use it. Yet for learning, all of these advanced capabilities don't have to be utilized. This is what makes the language so versatile—its built-in power that is accessible on demand. And it is this versatility that allows teachers to tailor LOGO for special applications, and reassures all students that with LOGO there is always more to learn.

PART 2: Constructing a DYNATURTLE

The instructions for using the dynaturtle are obtained by typing HELP. The dynaturtle itself is activated by typing DYNATURTLE. The procedure starts out drawing a circle and displaying a white dynaturtle. Touching the E key

SHAPE 10   SHAPE 11   SHAPE 12   SHAPE 13

SHAPE 14   SHAPE 15   SHAPE 16   SHAPE 17

SHAPE 18   SHAPE 19   SHAPE 20   SHAPE 21

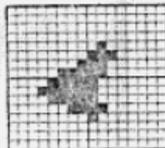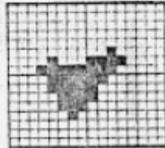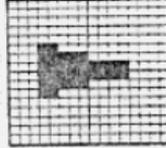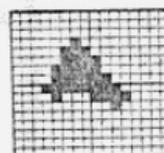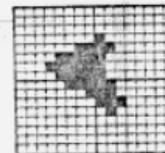causes a "thruster" to impart motion to the dynaturtle with speed 3. Each touch of the E key adds a velocity with magnitude 3 to the dynaturtle. Touching S or D makes the dynaturtle face 30 degrees left or 30 degrees right from its former heading. Velocities add like vectors. If the dynaturtle is not facing in the direction of its motion, the force of the thruster will cause it to head in a direction intermediate between its heading and direction, exactly as if it were a rocket in space obeying Newton's laws.

Touching F will turn friction on. In this state, the dynaturtle will be sluggish and come quickly to a stop after each kick. It will therefore be necessary to increase the force of the thruster. To do this, touch K. You can then enter a number, say 10 or 20, and touch ENTER. The dynaturtle will now be given an increase in velocity with magnitude 10 or 20 with each touch of E. Touching F again will turn friction off. You will find the dynaturtle now very difficult to control. Touch K again and readjust the thrust.

When friction is off, the dynaturtle is seen to act just like the ship in the *Asteroids* arcade game. When friction is on, it behaves as if it were riding on a rough surface—appearing to skid as you direct it around the circle.

Description of Procedures
DYNATURTLE activates the procedures INITIALIZE, SETDYNATURTLE and CONTROL.

INITIALIZE draws a circle and initializes the thruster (sprite 0).

SETDYNATURTLE positions the dynaturtle and gives it its initial shape (shape 10). The secret of the dynaturtle's turning capability is that the twelve shapes (shape 10 through

21) contain designs for the dynaturtle, each rotated 30 degrees from the preceding.

CONTROL is the main loop. Friction is always checked to see if it is on. If it is on, CHECKFRICTION decreases the dynaturtle's speed. If one of the control keys is pressed, the action is taken and control branches to label A. This procedure keeps running until Q is touched.

KICK reads the velocity of sprite 0, which is always kept heading in the direction the dynaturtle is facing. This velocity is then added to the velocity of sprite 1, which carries the shape of the dynaturtle.

ROTRIGHT adds 30 degrees to H, which maintains the heading of the dynaturtle and causes sprite 1 to carry the shape with next highest number, unless that number is larger than 21. If sprite 1 is carrying shape 21, it assumes shape 10. In this way, the dynaturtle appears to be rotating to the right by 30 degrees.

ROTLEFT is similar to ROTRIGHT but gives the effect of rotating the dynaturtle to the left.

SETFRICTION simply makes the value of the word FRICTION? true if it is false, and false if it is true.

SETKICK gets a number from the console and assigns it as the speed for sprite 0. The velocity for sprite 0 (x- and y- coordinates) is used to impart an acceleration to sprite 1. Note the command SS FIRST READLINE. The primitive READLINE outputs a list, and SS requires a number for input. The desired number is the first (only) member of the list entered.

EXT·E·N·D·I·N·G LOGO
Applications for Very Young Children

Seymour Papert designed LOGO to have a low threshold so that even young children could benefit from it. Unfortunately, the present technical requirement that LOGO input and output be *text-bound* limits LOGO to children who can read and type.

It is apparent that for children in a LOGO environment offers greater potential for pre-verbal children than for verbal humans. This is simply because there is so much more for them to learn. But it is equally apparent that the reading/typing prerequisite is an artificial barrier to that same environment. The ultimate solution—a computer which can comprehend *and* generate speech—is not-yet available. (Programs such as TI text-to-speech, however, can do a reasonable job of talking.) Still, there are ways LOGO can be adapted to children who are only able to recognize alphabetic characters or typewriter keyboard symbols.

Even before LOGO was implemented on the DEC LSI/11 or the TI-99/4, people in the MIT LOGO lab worked on simplified LOGO systems. One approach yielded a special LOGO input device which translated symbol cards[1] inserted into slots through a light scanner into ASCII code. Although prototypes of the "slot machine" card reader worked well enough, the idea was never developed commercially. A second approach was followed by Bob Lawler, a graduate student at the time, who wanted his two children to be able to use LOGO. He wrote a number of excellent LOGO programs which allowed very young children to draw with the turtle, to play shoot-out games with the turtle, or to design elaborate turtle pictures. Lawler's programs were written for a large, mainframe computer version of LOGO, but his ideas are compatible with TI LOGO. In fact because of the excellent color graphics of TI LOGO, his ideas may involve children more effectively when set up on the TI-99/4A. The essence of his programs was to simplify access to turtle geometry by simplifying and combining commands. One simplification allows pupils to turn the turtle forward or backward a fixed amount, or to turn the turtle left or right a fixed amount by pressing any of the four "arrow" keys:

```
TO SIMPLEFY
  TELL TURTLE
  SIMPLEF1
END
```

[1] The cards carry labels like ▶ for RIGHT 90, ↑ for FORWARD 10, as well as symbols for recursion and sub-procedure calls.

```
TO SIMPLEF1
  CALL :NC :ANSWER
  IF :ANSWER = "E RIGHT 10
  IF :ANSWER = "D FORWARD 10
  IF :ANSWER = "S LEFT 10
  IF :ANSWER = "X BACK 10
  SIMPLEF1
END
```

With slightly more sophisticated children, or as children work with DRAW, we can add other commands by merely inserting them into SIMPLIFY. For example:

```
IF :ANSWER = "C CLEARSCREEN
IF :ANSWER = "Q STOP
IF :ANSWER = "U PENRAISE
IF :ANSWER = "1 PENDOWN SETCOLOR 1
IF :ANSWER = "4 PENDOWN SETCLOR 4
IF :ANSWER = "U PENUP
```

Then, as students master the fuller set of DRAW commands and learn the idiosyncracies of QWERTY typewriter code, they can be introduced to TELL TURTLE without using DRAW any further.

Coleta Lewis, a teacher at the Lamplighter school in Dallas, adapted sprites for use by nursery school children. (The Lamplighter school pioneered the use of TI LOGO with children; see "The Lamplighter LOGO Project.") Two "games" her children played allowed them either to move a garage around the screen, move a car around the screen, and vary the colors of each separately, or to construct a face and then color in the parts of the face. Programming sprites for very young children is not much more difficult than DRAW. As one example of a sprite game for youngsters, consider a game of blocks. It is fairly easy to create a universe of blocks with simplified sprite commands. First, it is necessary to make up some "blocks" using MAKESHAPE. A circle (shape 4) and a square (shape 5) already exist. A good set of blocks ought to have a triangle.
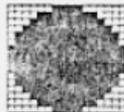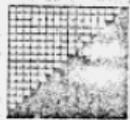

SHAPE 4
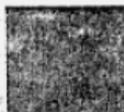
SHAPE 5

It would, however, be better to build triangles with four different orientations because the shapes of sprites cannot be rotated.
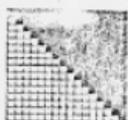
These could be assigned shape numbers 6, 7, 8, and 9. A good block set should also have a rectangle. To show the two orientations, shape numbers 10 and 11 could be designed as shown.

Other shapes could be easily added. Children should be able to color the blocks, move the blocks around, change the shape of any of the blocks, and bring more blocks onto the screen. In addition, when a child begins working with one of the blocks, he or she should be able to identify which one it is. (One way this identification can be aided is to have them briefly flash colors.) The following programs implement these ideas:

SHAPE 6   SHAPE 7   SHAPE 8   SHAPE 9   SHAPE 10   SHAPE 11

I THINK THAT I / SHALL NEVER SEE / A PET WHO'S / AS GOOD AS ME!

The LOGO Poet: USING RECURSION FOR LIST HANDLING

Since TI LOGO's graphics capabilities are so vast and so easy to use, there is a tendency to overlook its other features. List handling is a case in point: By combining some of LOGO's list primitives—such operations as FIRST, BUTFIRST, (or the converse LAST, BUTLAST)—with recursive [see adjoining *A Primer on Recursion and List Primitives*] OUTPUT lines, we can easily write programs to reverse a list, alphabetize a list, or even compose poetry. The several examples that follow will, I hope, demonstrate to you the powerful simplicity and list-manipulation potential of the language.

Verifying the presence or absence of a word in a list is a problem commonly encountered in list processing. The MIT LOGO group refers to this as the "MEMBER?" problem because the program is to answer the question, "Is a specified word in a specified list?" Some aspects of the program are obvious. For example, once the answer is obtained (whether TRUE or FALSE), it should OUTPUT to the user or program which called for the answer. It is also obvious that if the list is empty, the word is not in the list. Given just this much information, it is possible to frame a MEMBER? program:

```
TO MEMBER? :WORD :LIST
...
OUTPUT "FALSE
END
```

apart, following Polya, notes that one way of solving a complex problem is to ignore the complex whole and focus on those parts which can easily be solved. [See *Mindstorms: Children, Computers, and Powerful Ideas* by Seymour Papert—available from the 99'er Bookstore.] In the "MEMBER?" problem, if the first word in the list were the target word, then it would be easy to detect it and solve the problem using the LOGO primitive FIRST*, which returns the first word in a list:

```
TO MEMBER? :WORD :LIST
IF FIRST :LIST = :WORD OUTPUT "TRUE
...
```

Now all that remains is solving for those cases in which the word is in an *interior* position or is *absent* from the list.

Were the word *second* in the list, the problem would be solved by adding a line using the LOGO primitive BUT-FIRST, which returns all but the first word in a list of words:

IF FIRST BUTFIRST :LIST = :WORD OUTPUT "TRUE

since the *second* word in the list is the *first* word in a list which excludes the first word. Similarly, the third word becomes the FIRST of the BUTFIRST of the BUTFIRST of the list, the fourth word is the FIRST of the BUTFIRST of the BUTFIRST of the BUTFIRST of the list. It would be possible to write a separate line for each of those positions as well as the fifth, sixth, seventh or any other potential word position. However, a program that did this would quickly grow ponderous. Fortunately, in LOGO this is unnecessary. Notice that for each position an *additonal* BUTFIRST is all that is needed. The problem therefore requires only a *single* recursion line to complete the program:

```
TO MEMBER? :WORD :LIST
IF :LIST = [] OUTPUT "FALSE
IF FIRST :LIST = :WORD OUTPUT "TRUE
OUTPUT MEMBER? :WORD BUTFIRST :LIST
END
```

Now when we run the program by typing MEMBER? [A QUICK BROWN FOX] "FOX, the first stack checks to see if the list is empty or if the first word in the list matches the target word, FOX. Then it awaits the results of a second stack which runs MEMBER? with the truncated list and the target word. The second stack then awaits the results of a third stack which runs MEMBER? on BROWN FOX and "FOX. That stack then awaits the results of MEMBER? FOX "FOX which returns "TRUE (from the match in the second line). "TRUE is returned to the second stack which outputs "TRUE to the first stack which outputs "TRUE to the program which first called it (or to top level). In the event that there were no matches, each of the stacks would eventually run MEMBER? on an empty list and would output "FALSE.

Another common problem is to count the number of words in a list of words. As before, this problem is solved by outlining the obvious elements of the solution and the simplest case.

TO COUNT :LIST
OUTPUT *some number*
END

The simplest case occurs when the list is empty.

TO COUNT :LIST
IF :LIST = [ ] OUTPUT 0
OUTPUT *some number*
END

When a list has just *one* word in it, the program should recognize that and OUTPUT 1. Since a list with just one word is one word away from an empty list, The LOGO

* FIRST returns the first word in a list of words, or the first letter in a word. LAST returns the last letter in a list of words, or the last letter in a word. BUTFIRST returns all but the first word in a list of words, or all but the first letter in a word. BUTLAST returns all but the last word in a list of words, or all but the last letter in a word.

operation BUT FIRST applied to that list would yield the empty list. If there were two words in a list, then obviously the list is just two words away from an empty list. If a recursive line were ??? into the program which (a) applied BUT-FIRST and (b) added 1 to the count for every applic??? of BUTFIRST, the program would count the words in the list...

```
TO COUNT :LIST : OUTPUT
IF EMPTYP :LIST [ OUTPUT 0 ]
OUTPUT 1 + COUNT BUTFIRST :LIST
END
```

For another example, consider a program which will reverse a list. The simplest case would be a list with no words.

```
TO REVERSE :LIST
IF EMPTYP :LIST [ OUTPUT ]
END
```

The next simplest case would be a list with just one word. For such a list we could have the program OUTPUT the SENTENCE or the word and an empty list.

```
TO REVERSE :LIST
IF EMPTYP :LIST [ OUTPUT ]
OUTPUT SENTENCE FIRST :LIST [ ]
OUTPUT BUTFIRST :LIST
END
```

This solution can be applied to longer lists as well!

For a final example, let's use LOGO to "write" random poetry. As a first effort at LOGO poetry, we'll attempt some "free verse" by instructing a poet to string words together randomly from a list we select. First, we will need a program like SELECT to output a selected item from a list.

```
TO SELECT :N :LIST
IF :N = 1 [ OUTPUT FIRST :LIST ]
OUTPUT SELECT :N - 1 BUTFIRST :LIST
END
```

Then we need a program to generate random numbers for SELECT. Because LOGO's RANDOM primitive provides the integers through nine, if our list is less than ten, we can get a COUNT of it and use that COUNT.

```
TO RAND :LIST
CALL RANDOM :N
TEST :N = 0 :N
IFT [ RAND :LIST ]
IFF [ OUTPUT :N ]
END
```

By first typing

CALL COUNT :LIST "LENGTH

we can then use NUMB for the value of LENGTH. If we then type:

TYPE SELECT (NUMB :LENGTH) [a list of words]

the computer types one of the words in the list. We can write that as a program:

```
TO RAND :LIST
TYPE SELECT [ :NUMB :LENGTH ]
END
```

To turn this into a line of poetry, we should have a random number of such randomly picked words with a random number of spaces between words (E. E. Cummings' style) and then a carriage return:

```
TO SPACE :N
REPEAT RANDOM [ PRINTCHAR 32 ]
END
TO VERSE :LIST
REPEAT RANDOM [ SPACE VERSE :LIST ]
PRINT SELECT :NUMB :LIST
[ :LIST ]
END
```

Note: PRINTCHAR 32 puts the character with ASCII code 32, a space, on the screen.

If we want continuing lines of poetry, we can write a recursive program:

```
TO POEM :LIST
LINE :LIST
LINE :LIST
POEM :LIST
END
```

Now, putting this all together we get:

```
TO POET :LIST
CALL COUNT :LIST "LENGTH
```

Now we can try converting POET into a program which produces either rhyming verse, blank verse, or a finite number of lines of verse. One way to modify POET to produce rhymed verse is to give it two different lists—one of words for the interior words of each line of verse, and another of rhyming words for the last word in each line. Then the program can be changed so that only rhyming words are placed in end positions.

You probably recognize that the problem of generating rhyming verse is one form of the problem of teaching the computer to write text which follows a specified rule (in this case each line must rhyme). The more general application of rules to text is nothing less than grammar. One of the grade school pupils in the Brookline project wrote a textbook rule program like POET which generated random sentences. After she saw the effects of changing parts of speech she exclaimed enthusiastically that she now understood what a noun was.

POET can also be quickly adapted to a sentence generator which young people can play with to make grammar meaningful.

```
TO LINES 'LIST 'RHYMES
  LINES MUST ACCOMMODATE TWO
    LIST
  LINE 'LIST 'RHYMES
  LINES 'LIST 'RHYMES
END

TO POET 'LIST 'RHYMES
  THE SECOND LIST MUST NOW BE
    GIVEN THE PROGRAM
  CALL COUNT 'LIST 'LENGTH
  CALL COUNT 'RHYMES 'LENGTHR
  NECESSARY TO FIND OUT HOW
    MANY RHYMING WORDS THERE
    ARE
  LINES 'LIST 'RHYMES
END

TO LINE 'LIST 'RHYMES
  LINE MUST PUT RHYMES ONLY AT
    THE END OF EACH LINE
  REPEAT RANDOM 'SPACE VERSE 'L
    IST
  PRINT SELECT 1 'NUMS 'LENGTHR
    'RHYMES
  PUTS A RHYME AT THE END
END

TO SPACE
  TYPE 32
END

TO COUNT 'LIST
  IF EMPTY 'LIST OUTPUT 0
  OUTPUT 1 + COUNT BUTFIRST 'LIST
END

TO NUMS 'LENGTH
  CALL RANDOM 'N
  IF 'N > 0 AND 'N < 'LENGTH
  PUT 'N
  IF OUTPUT 'N
  IFF OUTPUT NUMS 'LENGTH
END

TO SELECT 1 'NUMS 'LENGTH 'LIST
  IF = 'N 1 OUTPUT FIRST 'LIST
  OUTPUT SELECT 1 'N + 1 OUTPUT FIR
    'LIST
    'LIST
END

TO VERSE 'LIST
  TYPE SELECT 1 'NUMS 'LENGTH 'LI
    'LIST
END
```

```
TO SENTENCES
  PRINT 'TYPE A LIST OF ARTICLES
    AND THEN PRESS ENTER.'
  CALL READLINE 'ART
  PRINT 'TYPE A LIST OF NOUNS AN
    D THEN PRESS ENTER.'
  CALL READLINE 'NOUNS
  PRINT 'TYPE A LIST OF ADJECTIV
    ES AND THEN PRESS ENTER.'
  CALL READLINE 'ADJ
  PRINT 'TYPE A LIST OF VERBS AN
    D THEN PRESS ENTER. NOW WATCH.'
  CALL READLINE 'VERBS
  GRAMMAR 'ART 'NOUNS 'ADJ 'VERB
    S
END

TO GRAMMAR 'ART 'NOUNS 'ADJ 'V
  ERBS
  TYPE SELECT 1 'NUMS 'COUNT 'AR
    T 'ART
  SPACE
  TYPE SELECT 1 'NUMS 'COUNT 'NO
    UNS 'NOUNS
  SPACE
  TYPE SELECT 1 'NUMS 'COUNT 'AD
    J 'ADJ
  SPACE
  TYPE SELECT 1 'NUMS 'COUNT 'VE
    RBS 'VERBS
  SPACE
  TYPE SELECT 1 'NUMS 'COUNT 'NO
    UNS 0 'NOUNS
  PRINT
  WAIT 30
  GRAMMAR 'ART 'NOUNS 'ADJ 'VERB
    S
END
```

SENTENCES can be made a better grammarian by adding distinctions of number and gender where appropriate; it can be made a more sophisticated language generator if GRAMMAR is altered to allow for conjunctions and subordinate clauses. All of these changes and more can be programmed by students as they learn both the specifics of grammar[1] and the mathematics of LOGO.

---

[1] Papert would probably argue that most students know the grammar which schools attempt to teach, but that the students do not have verbal labels for syntactical rules and parts of speech, and do not see the relevance of the labels once they are told them. A sentence generator program can make grammar "speech syntonic."

Avoiding Turtle Traps

Seymour Papert and his colleagues purposefully decided to structure LOGO to facilitate the writing of good computer programs. The concept of good programming is not necessarily apparent. Of course, a program should accomplish its intended goal, but all programmers recognize that any goal can be achieved by many different types of programs. Beyond simply "working," there are a number of criteria by which programs can be judged. Programs which have multiple applications are generally better than single-purpose programs. Programs which are easier to debug and which can be understood by people other than the authors (or which can be understood by the authors at a future time) are more desirable. Pragmatically, programs which run faster or with fewer bits of memory are better than slower or more memory-intensive programs. Finally, some programs are aesthetically more appealing than others.

It is possible to find examples of program applications in which two or more of the criteria are in conflict. However, it is more often the case that the criteria are in accord. All of the criteria except for aesthetics are straightforward and relatively objective. Still, writing aesthetic programs is so satisfying that aesthetics will be considered first here.

For instance, you can write GO in LOGO, but programs with many different branches from GO commands are particularly inelegant. Why write poor programs when good ones are easier to write? Also inelegant are programs with hundreds of lines of code, especially when the code contains several repetitions of a series of commands. And programs with many inputs are generally less aesthetic than those with fewer inputs. Compare the aesthetics of two programs which count the number of words in a list:

```
TO COUNT :N :LIST
IF BUTFIRST :LIST = [ ] OUTPUT 1
COUNT :N = 1 + BUTFIRST :LIST
```

This program requires typing as input along with COUNT and the list in question requires a starting value of :N - 0.

```
TO COUNT :LIST
IF :LIST = [ ] OUTPUT :N
OUTPUT 1 + COUNT BUTFIRST :LIST
```

This program requires no superfluous input. Elsewhere in this chapter, there is a fairly complex program, DYNATURTLE, which creates a turtle that obeys Newtonian laws of motion. Despite the complexity of the program, DYNATURTLE is relatively elegant: DYNATURTLE has only three lines, which are INITIALIZE, SETDYNATURTLE, and CONTROL. Each of those lines is, in turn, a brief program which serves a unique function. Contrast DYNATURTLE's elegance with a spaghetti-pole BASIC program which would achieve the same effects. Such a program would be long and littered with extensive GO-TO's.

A subtler example of elegant and inelegant programs can be made from the GRAMMAR program. The program was modified from an earlier POET program and was written:

```
TO GRAMMAR :ART :NOUNS :ADJ
SPACE
TYPE SELECT :NOUNS :COUNT :NO
SPACE
TYPE SELECT :NOUNS :COUNT :NO
SPACE
TYPE SELECT :NOUNS :COUNT :VE
SPACE
TYPE :VERB
SPACE
TYPE SELECT :NOUNS :COUNT :VE
[ADJ]
SPACE
TYPE SELECT :NOUNS :COUNT :NO
[NO]
TAS :NOUNS
PRINT [ ]
WAIT 10
GRAMMAR :ART :NOUNS :ADJ :VERB
```

Notice how much of each line is repetitive. A better LOGO program would have taken advantage of that redundancy and used a broader application program:

```
TO WORD'S
TYPE SELECT :   : NUMB COUNT X
END
```

Then GRAMMAR could be written:

```
TO GRAMMAR :ART :NOUNS :ADJ :V?
WORDS :ART
SPACE
WORDS :ADJ.
SPACE
WORDS :NOUNS
SPACE
WORDS :VERBS
SPACE
WORDS :ADJ
SPACE
WORDS :WORDS
PRINT
WAIT 50
GRAMMAR :ART :NOUNS :ADJ :VERBS
END
```

The second GRAMMAR program is more elegant and is shorter. It achieved greater simplicity by taking out of GRAMMAR all of the repeated functions and placing them in WORDS. All of the functions carrying out the program WORDS are directed at placing a single word from a designated set of words. The specification of the set and type of words is left for GRAMMAR, the program surrounding WORD. A common format for many well-written LOGO programs is:

TO DOSOMETHINGSPECIFICALLY :SPECIALINPUT
GENERALPURPOSEPROGRAM :GENERALINPUT
END

TO GENERALPURPOSEPROGRAM
:GENERALINPUT
LOGO commands :GENERALINPUT
END

On occasion it is necessary to string together several general-purpose programs inside a specific-purpose program. In that case, the general program often requires that there be some set-up steps and some "fix-up" steps before and after the general program. Such programs have the form:

TO GENERALPURPOSE
SETUP
GENERALFUNCTIONS
FIXUP
END

Mathematicians may indeed recognize a similarity between the concept of elegance and aesthetics in programming and the expression of algebraic functions. There are many ways to express algebraic functions, but it is often more useful and always more elegant to express such functions in a form which collects common factors and simplifies terms even where such simplification may require a set-up or a quick fix-up manipulation along with the factoring.

There are two other major aspects to consider in order to write better LOGO programs. One is writing programs which don't run out of memory; the other is writing them to run as fast as possible. It is important to understand the major feat accomplished by the MIT LOGO Lab in putting LOGO on the 99/4. LOGO is a very high level computer language which requires large amounts of memory. The architecture of microcomputers limits the speed with which large amounts of memory can be addressed. The TI LOGO which emerged from the joint efforts of TI and MIT represents an effort to compress code to the minimum memory requirement without compromising its applications. There are two tricks which they built into TI LOGO to make LOGO feasible on a micro. If you use these tricks you can gain even greater satisfaction from your computer. The first feature is an automatic garbage-collector. A garbage collector is a part of the operating system which takes used memory and makes that memory available for further uses. Of course, the garbage collector should not destroy and overhaul all of memory's work. The way that the automatic garbage-collector in LOGO recognizes when a unit of memory has served its purpose is by checking the instructions written in the memory. Below are examples of programs which permit or exclude the collector:

```
TO POLYGON :SIDES :ANGLE
FORWARD :SIDE
LEFT :ANGLE
POLYGON :SIDE :ANGLE
END
```

In this program, the garbage collector notes that each time POLYGON is entered (referred to as the *level* of POLYGON), there are no further commands or instructions after the line POLYGON :SIDES :ANGLES (called the recursive call line). Thus the piece of memory that was used to store POLYGON at that level is collected for reuse. If all memory gets used up in TI LOGO, the message "OUT OF SPACE" appears, but POLYGON will never generate that message because it will never run out of memory.

```
TO POLYGON :SIDE :ANGLE :LENGTH
FORWARD :LENGTH
LEFT :ANGLE
END
```

This program will never run out of memory in TI LOGO because the program terminates.

This program could use up all available memory before it reaches its stop conditions because the garbage collector cannot refurbish the memory used to execute this POLYGON at any level. The program leaves work to be done (namely PENUP) once control is passed back to the level of POLYGON.

Unfortunately, the garbage collector is not empowered with the authority to decide if any instructions following the recursion call are worth keeping, and so the following POLYGON program could run out of memory:

```
TO POLYGON :SIDES :ANGLE
FORWARD :SIDE
LEFT :ANGLE
POLYGON :SIDE :ANGLE
END
```

The only difference between the first POLYGON program and the one here is the empty line following the recur-

sion call and before END. The garbage collector sees that there is a line of commands and cannot tell that the line is useless, so it is barred from refurbishing the memory! Empty lines use up memory and can block garbage collection (depending on their location), so empty lines should be eliminated from your programs.

Finally, the operating system can work faster when fewer sprites are being used, i.e., programs which use no sprites run faster than programs which use sprites. The more sprites in use (generally), the slower the system operates. The reason for the slight degradation in response time is obvious—the system has to check to see which, if any, sprites must be displayed or moved. The system checks on its sprites by looking up the highest number of sprite called upon. For example, TELL 31 or TELL SPRITE 31 would cause the system to check on every sprite from 31 on through to sprite 0. Such a check is necessary (from the user's perspective) only if all 32 sprites are being used. If only one sprite is needed, then the user should type TELL 0 or TELL SPRITE 0 and the system would skip the checkup on sprites 1 to 31, thus saving a small amount of time.

Student Reactions to a Four Week LOGO Class By Gene Branum

Students pick up these principles quickly. For instance, Gene Branum, a student in a four-week LOGO course, reflects on this experience:

"The expectations of the students varied—we wanted to know more about computers, we wanted a different Jan-term experience, or maybe just a free Jan-term. Whatever the motivation, all came away affected in some way by our experience. All experienced both the frustration of failure and the flush of triumph as the computer finally 'did what it was supposed to.'

"The format for our experience was a four-week mini-term (Jan-term) at Austin College. Our class met; five days a week for two hours, and we were required to spend at least one hour of work on our own as well. This requirement was easily met; as one student put it, 'it was not unusual to spend four hours at a time' on the computer. Needless to say, the experience was very intense, and there was a great deal of self-teaching. This was felt to be one of the greatest strengths of the course.

"Professor Hank Gorman did a fine job of teaching the basics early in the course. As he told us his expectations, we scoffed. After two weeks, he told us, we would be drawing cartoons and making up games. Even though his leadership was great, the majority were insecure about 'the machine.' Our confidence, however, grew with experience and familiarization.

"The two greatest aspects of the course for all of us were (1) the team experience and (2) application in general problem solving skills. The true strength of LOGO is that students, working together, can teach each other massive amounts of material. The realization that everyone had problems put us all on the same level. Sharing ideas and solutions became important for everyone because no one could work totally independently. Many social experiences allow students to interact, but LOGO is one of the few that forces students to think together.

"Without exception, all of the students involved in the course commented that, after LOGO, they knew better how to approach a complex problem. Dr. Gorman spent several class periods on problem solving skills: decomposition, recursion, naming, multiple descriptions, and the 'little men.' These skills not only aided our search for solutions to LOGO problems, but also for problems that require a thinking solution. The overriding principle of LOGO is that the simple builds to the complex, which is its major strength as a system for any age-group.

"While it was widely agreed upon that none of us 'mastered' LOGO, each of us developed confidence in our abilities to control the computer and make it do what we requested. The LOGO experience allowed everyone to use logical approaches to problem solving and gain valuable hands-on experience in a discipline that continues to increase in importance."

The following students, whose students wrote during this course, show an emerging appreciation for elegance, speed, and simplicity in programming. Except for correction of typographical errors, their work hasn't been edited in an attempt to find still more elegant ways of achieving their programs' goals. Note, however, that they all grasp the essentials of esthetic programming.

Space Pylon Racer

Once set up, the player guides his saucer through pylons. Two shapes must be made first (check graph paper). The keys control the saucer. E moves it upward, X moves it downward, D moves it forward, S moves it backward, F speeds it up, A slows it down. If the ship hits a pylon, the beep sounds.

Use arrow keys to change direction.
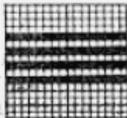**Use F for fast speed.**
Use A for slow speed.

MAKESHAPE 20
Saucer

MAKESHAPE 21
Pylon

```
TO SP1
TELL 0 [C1 CARRY 20
100 RC1 1 2 3 4 5 6 1
TELL 1 30C1 BLACK
TELL 1 1 1XY - 50 60
TELL 2 1 1XY - 80 60
TELL 3 30C1 80 80
TELL 4 1 1XY 30 60
TELL 5 1 1XY 100 80
TELL 6 1 1XY 100 60
END

TO SP3
CONTROL
CHECK
SP9
END

TO CONTROL
IF RC1 HAVE THEN TEST 3 = 4
CALL SP1
IF - AC T TELL 0 1XY 0 STOP
IF - AC 2 TELL 1 1XY 100 STOP
IF - AC 3 TELL 2 1XY 70 STOP
IF - AC 4 TELL 3 30C 0 STOP
IF - AC 5 TELL 4 1XY 60 STOP
IF - AC 6 TELL 5 1XY 0 STOP
END
```
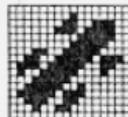

Spinout

This program was designed as a cartoon to depict two Indianapolis-style racing cars racing, crashing, burning, and being towed away. The central program, SPINOUT, contains 7 subprograms. These short programs make the central program neat and concise.
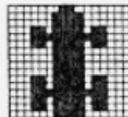
MAKESHAPE 6   **MAKESHAPE 7**

```
TO SPINOUT
WAVE
MOVE
WAIT 550
SWERVE
WAIT 30
SPIN
WAIT 5
BURN
TOW
END

TO SETUP
TELL 0
CLEAR
SC 6
SC 8 100C6 0
NOMOVE
TELL 1 CARRY 8
SC 1
SS 1 8 90
NOMOVE
TELL 0 1 SS 15
END
```


MAKESHAPE 8

```
TO WAVE
TELL 1 CARRY 1
SC 1
SS 1 1
SC 1
TELL 0 1 CARRY 10
SC 10
TELL 0 85 - 80 ST 8
SC 10
SETUP
TELL 1 REPEAT 4 1ST 5 WAIT 10
ST 10 WAIT 15 ST 15 WAIT 15 WAIT 15
KEEP
WAIT 30
NOKEEP
END

TO MOVE
TELL 0
SS 92
SC 1 1 SS 90
SC 1
END

TO SWERVE
TELL 0
REPEAT 35 SS - 2 WAIT 5 SS 38 WAIT
WAIT 5
END

TO SPIN
CALL 10 1 1 TEAM
TELL 1 TEAM 10
SC 2
REPEAT 5 CARRY 6 WAIT 5 CARRY 8
WAIT 5 CARRY 4 WAIT 5 CARRY 2
WAIT 5 CARRY 8 WAIT 5 CARRY 2
WAIT 5
END

TO BURN
REPEAT 10 1C8 6 WAIT 5 C8 11 W
SC 8 0 WAIT 5
SC 1
TELL 1 TEAM 10
TELL 10 TEAM
SC 1 SS 0
C8 7
END

TO TOW
TELL 1
SS 1 90
ST 1 20
WAIT 100
SC 378
SS 1 WAIT 205
CALL 10 15 11
TELL 1
SS 1 90
SC 11 15C 0 SS 0
END
```


MAKESHAPE 9

MAKESHAPE 10

MAKESHAPE 11

Munchie

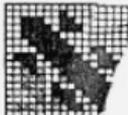Munchie illustrates how one can program a sprite to move to certain locations where an object may be found. After testing coordinates within the procedure, it eats that object and continues on until it eats all objects. You move Munchie by using arrow keys, and set speed by using keys 0, 5, and 1. You should stop Munchie when passing over the object to be eaten.
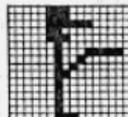
```
TO MUNCHIE
SETUP
KEEP
MOVE
END
```

```
TO SETUP
TELL 0 CARRY BLANK
TELL 1 BLUE KEY
TELL 2 CARRY 73
TELL 3 GREEN SET 30 30
TELL 4 CARRY ROCKET
...
END
```

MAKESHAPE 10   MAKESHAPE 11

MAKESHAPE 6

```
TO WIPE
GS
CALL 1 ALL
REU
SET 110 95
END

TO SETUP
WIPE
TELL 0 CARRY RED
SET RED
LEY 4 5 P
TELL 1 CARRY U
SET BLUE
TELL 2 CARRY 73
SET 78 33
TELL 3 CARRY U
SET BLACK
TELL 9 P B
END
```

```
TO MOVE
BEEP WAIT 5
CARRY 73 BEEP WAIT 5 CARRY 19
ROSLEP
IFF CALL IN
IFT CALL SC
...
IFT CARRY 19 --> CARRY 11 303
END
CHECK
END
```


MAKESHAPE 7

```
TO SLICE
WAIT 90
TELL 5 CS
CARRY U
SET 70
IS 93
TELL 4
XS 93
END
STILE 11 JUMP 1
END
```

```
TO EAT1
REPEAT 25  BEEP WAIT 2 ROSLEP
WAIT
TELL 0  SS 0
TELL 0  SS 0
MOVE
END

TO EAT2
REPEAT 25  BEEP WAIT 2 ROSLEP
TELL 0  SS 0
TELL 0  SS 0
MOVE
END
```

```
TO RISE
TELL 5
CARRY U
SET 30
XS 93
WAIT 60
XC U 50
SET 93
XC U 20
END
```


MAKESHAPE 8

```
TO CHECK
TEST IN XCOA --> SS XCOA
...
IFT TEST 10TH YCOR --> YCOR
...
IFT TELL 0 SS 0 EAT1
IFT CARRY 19 BEEP WAIT 5 CARRY
TELL ROSLEP
TEST 10TH XCOR --> SS XCOR
...
IFT CARRY 19 BEEP WAIT 5 CARRY
TELL ROSLEP
...
IFT TELL 0 SS 0 EAT1
```


MAKESHAPE 9
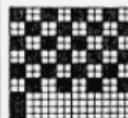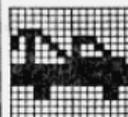
```
TO JUMP
CARRY 20
SS 14
YC U 30
WAIT 10
END
```
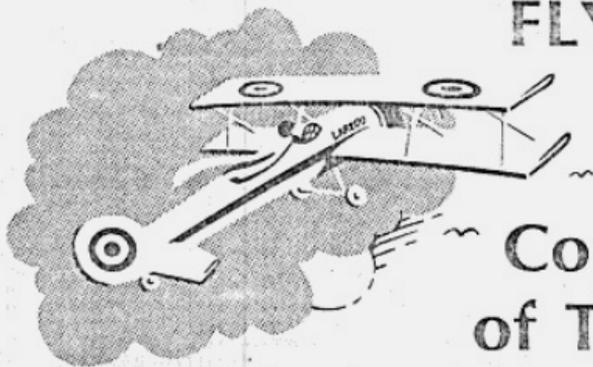

MAKESHAPE 10

MAKESHAPE 11

MAKESHAPE 12

FLY AWAY

with the

JOY

Commands
of TI LOGO

You push the stick forward, and the aircraft begins to roll. You then gradually pick up speed and start moving down the runway. After reaching takeoff speed, you move the stick again, and suddenly you're airborne. Now you have control of the skies—to fly high or low, do loops and other maneuvers, and then land. But be careful with your speed! You don't want to stall and crash.

This isn't flight training or a simulator. It is a TI LOGO procedure that gives you the opportunity to fly by keyboard or joystick. It uses either the arrow keys or the JOY 1 and JOY 2 commands. The JOY commands return one of nine values depending on the position of the joystick, thus opening a wide range of possibilities for interactive games and other activities.

At first, it might appear that the nine values have little relationship to each other. You'll note that the three and the seven omitted. However, the pattern of the values is quite interesting.

```
2     6    10
 \    |    /
  5 --5-- 9
 /    |    \
0     4    8
```

Moving from left to right in each row, you'll note that each digit is four more than the previous digit, i.e., 2 + 4 = 6, 6 + 4 = 10, etc. Moving from bottom to top in each column, observe that each digit is one more than the previous one.

These patterns begin to suggest why three and seven were omitted from the values assigned. However, to make the logic behind these patterns even more graphic, let's convert them to binary numbers.

```
0010   0110   1010
 |      |      |
0001   0101   1001
 |      |      |
0000   0100   1000
```
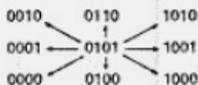
Now look at the first two digits in each column. You'll note that they are the same, representing from left to right, 0, 1, and 2. Also, if you look at the last two digits in each row, you'll note that they are also the same. Moving from bottom to top, they also represent 0, 1, and 2. So what we really have here is a distinctive coordinate system with real meaning, rather than what might first be perceived as a random placement of values.

```
(0,2)   (1,2)   (2,2)

(0,1)   (1,1)   (2,1)

(0,0)   (1,0)   (2,0)
```

Let X and Y be used to name these coordinates. The coordinates for the joysticks can be assigned with the command.

MAKE "Y (JOY1) / 4
MAKE "X (JOY1) - 4 * :Y

Now let's put these JOY commands to work in FLYAWAY, a procedure developed by Roger Kirchner, a fellow YPLA member. This is a procedure for one or two players that tests each player's ability to take off and safely land an airplane on the runway shown on the screen. Either the direction keys on the keyboard or the joysticks can control the plane.

The joystick commands are incorporated in the procedure, STICK S. Push the stick forward, and the aircraft increases its speed.

IF :X = 6 THEN FASTER

Pull the stick back and the aircraft slows down:

IF :X = 4 THEN SLOWER

To minimize the chance for error, direction commands are accessed by merely moving the joystick to the left or

right. It does not matter whether you hit position 0, 1, 2; the aircraft will turn left.

```
IF :X > 4 THEN TURNLEFT
IF :X < 6 THEN TURNRIGHT
```

Of course, it would be possible to add additional maneuvers using each of the nine joystick positions. This would require a much more sensitive touch to the joystick, but that could also add to the challenge of the flight.

With each turn of the aircraft, a new shape is called to show that new position. These range from #10 through #18. The first shape, #10, is similiar to the Plane shape in TI LOGO. The next shape shows the aircraft at a 45° angle. The other shapes depict the plane in a 90° angle, 135° angle, at 180°, 225°, 270° and 315°. Shape #18 depicts the crash.
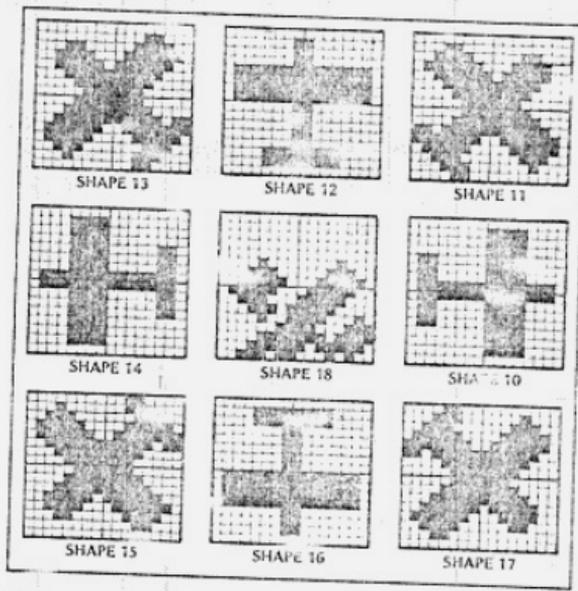
The CONTROLJOY and the CHECK P procedures control the aircraft in the air. CHECK P monitors the speed and "altitude" o, the aircraft to test for the CRASH parameters.

FLYAWAY is an excellent graphics program that begins to tap the power of the LOGO language. It goes quite a bit further than merely drawing pictures with the computer. This is an important point to realize. But unfortunately, not all educators do. In a November issue of *Infoworld*, one educator stated that although he can understand young peo-

ple enjoying maybe 50 to 100 hours of drawing pictures with LOGO, he would imagine they would then tire of larger [?] and move on to the other things.

Do chess players ever tire of chess? Do chess masters ever [?] feel that they have mastered the game? Probably not. The moves of chess can be easily learned by primary grade youngsters, but entire lifetimes are spent learning the game. Certainly the graphics capabilities and the speed of LOGO are spectacular. Indeed, they tend to overshadow the other attributes of the language. Where that happens it is most unfortunate because LOGO offers a young person so much more than just graphics—much more than BASIC and some other high level languages.

For example, look closely at BASIC: it uses a fin [?] number of commands that must be strung together [?] statements that tend to hide the operation of the program [?] Were the operation of BASIC programs easily discern [?] TRACE would be unnecessary. LOGO, on the other han [?] is a virtually unlimited language. If the command doesn [?] exist, use your imagination and create it! This is the [?] marvellous challenge of TI LOGO—using your imagina [?] tion and creativity to discover the *real* potential of the com [?] puter. Way back in the dark ages before microcomputers [?] Albert Einstein expressed a truth that is especially relevan [?] to today's computer learning environments: "Imagination [?] is more powerful than knowledge." Undoubtedly, Einstein [?] would have approved of TI LOGO.

SHAPE 13

SHAPE 12

SHAPE 11

SHAPE 14

SHAPE 18

SHAPE 10

SHAPE 15

SHAPE 16

SHAPE 17

```
TO HELP
CS
PRINT [FLYAWAY]
PRINT
PRINT [BLUE PILOT USES KEYS E,
 S, X, D ON [JOYSTICK 1]
PRINT
PRINT [RED PILOT USES KEYS J,
 K, M, N OR [JOYSTICK 2]
PRINT
PRINT [CONTROLS]
PRINT
PRINT
PRINT [FASTER  E  OR STICK FO
RWARD
PRINT [SLOWER  X  OR STICK BA
CK
PRINT [TURN LEFT  S  OR STICK
 LEFT
PRINT [TURN RIGHT  D  OR STIC
K RIGHT
PRINT
PRINT [SEE WHO CAN TAKE OFF AN
D LAND SAFELY FIRST
PRINT
PRINT [BON VOYAGE ]
END

TO VANISH
TELL [ALL HOME SX 0 SS 0 CARRY
 0]
END

TO CONTROL.JOY
STICK 2 CHECK 1
STICK 2 CHECK 2
IF RC1 TREK STOP
CONTROL.JOY
END

TO SET.PLANE
TO  M  TREK MAKE "YS  YS  E
SET MAKE "YS  YS  P
TELL "X C  IF P = P - 7  SET
  XS  YS SS 0 SX 90 CARRY 10
END
```

```
TO STICK.S
MAKE "S JOY 1
TELL S
IF S = 4 THEN TURNLEFT
IF S = 6 THEN TURNRIGHT
IF S = 5 THEN FASTER
IF S = 0 THEN SLOWER
END

TO CHECK P
TELL P
IF SPEED > 15 THEN TREK STOP
IF YCOR > 25 THEN TREK STOP
TEST (EITHER YCOR > UPOFF YCOR
 < DOWNOFF)
IFT CRASH
IF SPEED = 0 THEN WELCOME
TREK STOP
WAIT 120
PLANE WHO
END

TO TURNRIGHT
RT 45
IF SHAPE = 10 THEN CARRY 17 EL
SE CARRY SHAPE - 1
END

TO TURNLEFT
LT 45
IF SHAPE = 17 THEN CARRY 10 EL
SE CARRY SHAPE + 1
END

TO SLOWER
IF SPEED = 0 THEN SS SPEED - 5
END

TO FASTER
IF SPEED = 100 THEN SS SPEED +
 5
END
```

```
TO PLUS.W
IF W = 91 THEN OUTPUT  W + 1
ELSE OUTPUT 91
END

TO MINUS.W
IF W = 0 THEN OUTPUT  W - 1
ELSE OUTPUT 57
END

TO WELCOME
PRINT [  NICE LANDING. ] PILO
T WHO
END

TO CRASH
PRINT [NOT SO GOOD. ] PILOT
WHO
MAKE "Y 10
MAKE [DROP YCOR - 50 ]
...
MAKE "I 55 + ...  DROP + [ ] ]
IF Y = 0 THEN MAKE "I 11 - Y
CARRY 18
SS 0
END

TO PILOT.W
IF W = 0 THEN OUTPUT  PILOT
ELSE OUTPUT [PILOT]
END

TO RUNWAY
TELL TILE 86 SC 12 YS 2
CLEAN W 10 SC GREEN
MAKE "TILES [164 164 164 164 5
& 164 164 164 164 ]
MAKE [COL YS ]
MAKE "TILES [ ]  TREK STOP
MAKE "COL W
...
COL [ ] YS [ ]  COL  NOW MAK
MAKE "TILES 87 "TILES
MAKE "ROW W [ ]
GO "A
END
```

Problem Solving
WITH LOGO

I t is pleasureable to work with a language like LOGO because it gives us something to "think with," and it encourages us to think in what Papert has called "mind-sized bites." The solution of a problem can be identified with the definition of a procedure. If the problem is simple, we can specify the procedure directly. Otherwise, we try to specify it in terms of a small number of simpler procedures.

Often, this method leads to a complete solution of a problem. But sometimes, a problem is so complex that the method leads to an indefinite number of problems. A solution seems hopeless.

But suppose that new problems have the same form as previously encountered problems, and are simpler. The problem will be solved at least "theoretically," if the rules lead to a solution in a finite number of steps. Such a solution is said to be *recursive*.

One of the beauties of a language such as LOGO is that recursive procedure definitions are allowed. And writing a LOGO procedure not only gives a *theoretical* solution, but a practical one which can be carried out by executing the procedure. Of course, for the latter, one needs access to a TI-99/4A with TI LOGO (or some other implementation of LOGO).

In thinking through the solution of a problem, one often works "both ends." The big picture leads to smaller pictures. But also details occur which can be incorporated into procedures, which then make the solution of larger problems easier.

Translating the Pig Latin

As a concrete example of these ideas, consider the momentous task of translating an English word into Pig Latin. According to my children, the rule is to add "HAY" at the end of a word beginning with a vowel, otherwise to take the consonant sound from the front, add "AY" to it, and put it at the end. Thus "AND" translates to "AN-DHAY", and "BREAK" translates to "EAKBRAY".

These rules lead immediately to a LOGO procedure for accomplishing the task:

ICENAY OVEMAY!

```
TO TRANWORD :W
TEST MEMBER FIRST :W :A E I O
IFT OUTPUT WORD :W "HAY
IFF OUTPUT TRANCWORD :W
END
```

This procedure reduces our problem to the solution of three simpler problems, which we might need to reduce further. The procedures we need are:

    MEMBER *object list*
    TRANWORD *word*
    TRANCWORD *word*

MEMBER returns TRUE if *object* is in *list* and returns FALSE otherwise. TRANWORD translates *word* if it begins with a vowel. TRANCWORD translates *word* if it begins with a consonant. We can hope that MEMBER is a utility built into LOGO. It isn't, but this is no problem. Nearly anything that isn't a primitive can be built in.

At any stage in the solution process we can decide to work on big problems or focus on little ones. The solution of a problem isn't a linear process, even if solutions are usually presented as if the process were orderly and straightforward. The LOGO procedures document and organize progress. Let's focus on the problem of deciding membership. If *object* is in a list, it is either the first item of the list, or else

```
TO PRINT :X
CR PRINT (FOR PIGLATIN PRACTICE)
PRINT (TYPE "PIGLATIN :X
END

TO PIGLATIN
CR
PRINT (I WILL HELP
PRINT (YOU LEARN PIGLATIN )
PRINT
A
PRINT
PRINT (TYPE A SENTENCE )
MAKE "LINE READLIST
IF :LINE = [] THEN PRINT.LINE
IF :LINE = [] THEN TO PRINT
PRINT YOU.LINE
GO :A
END
```

```
TO TRANWORD :W
IF :K > 5 THEN MAKE :W
:VOWELS [A
K > 5 THEN MAKE :W
TEST MEMBER FIRST :W :VOWELS
IFT OUTPUT WORD :W :VOWELS
IFF OUTPUT TRANCWORD :W
OUTFIRST :W FIRST :W
END

TO TRANWORD
TEST MEMBER FIRST :W :A E I O
IFT OUTPUT TRANWORD :W
IFF OUTPUT TRANWORD :W
END
```

```
TO PRINTPIG :LINE
TEST :LINE = W
IFT PRINTSTOP
IFT PRINTSTOP
IFF OUTPUT WORD TRANWORD F :LINE PRIG
TCARD $3 PRINTPIG BF :LINE
END

TO PRINTWORD :W :HAY
END

TO MEMBER :X :SET
TEST :SET = []
IFT SET = FIRST :SET THEN OUTPUT "TAL
IFF OUTPUT TRUE
IFT OUTPUT FALSE
IFF OUTPUT MEMBER :X BF :SET
END
```

it is the first of a truncated list, or it is not in the list. The definition is, naturally, recursive:

```
TO MEMBER :X :SET
IF :SET = [ ] THEN OUTPUT "FAL
IF :X = FIRST :SET
 IF OUTPUT "TRUE
 IF OUTPUT MEMBER :X BF :SET
END
```

With this definition, MEMBER FIRST :W [A E I O U] will return TRUE if :W begins with a vowel, and FALSE if it doesn't.

The definition of TRANVWORD is so simple we can write the procedure anytime. Let's do it now:

```
TO TRANVWORD
OUTPUT WORD :W "HAY
END
```

The (undocumented) primitive WORD takes two words as input and outputs the word formed by joining them.

The definition of TRANCWORD takes more thinking. We want it to be recursive. We want to move letters from the beginning to the end until the first letter is a vowel, and then add "AY". We are led to:
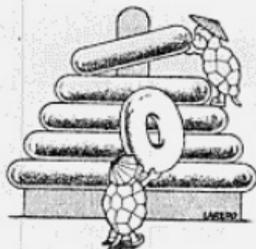
```
TO TRANCWORD
TEST MEMBER FIRST :W [A E I O U]
IF T OUTPUT WORD :W "AY
IF F OUTPUT TRANCWORD ( WORD BF :W
TFIRST :W )
END
```

If we try (that is, think through, or execute in LOGO) TRANCWORD "BREAK, we find it will return EAKBRAY, as desired. And TRANCWORD "YOU returns OUYAY. But TRANCWORD "YOU runs out of space because the recursion cannot end. Evidentally Y must be added to the list of vowels. But then TRANCWORD "YOU would return YOUHAY and not OUYAY.

Can you fix this bug? We want Y to count as a vowel only if it isn't the first letter. One solution is to use two inputs to TRANCWORD, one of which is a flag. This solution, as well as the generalization to translating a sentence, can be seen by reading the PIGLATIN procedure and the procedures it calls.

TOWER OF HANOI

Now we turn to a less frivolous example. The Tower of Hanoi is a puzzle familiar to many. It consists of three pegstands. One contains a "tower" of circular rings. The object is to move the tower from one peg to another, moving one ring at a time, and never putting a larger ring on top of a smaller one. There is rumored to be a Buddhist priest working on a puzzle with 64 rings; when he finishes, the world will end. If he makes one move per second, how much should we worry?

We can use LOGO to worry about this problem. We need a procedure, say NUMMOVES, which takes for input the number of rings and outputs the number of moves. Suppose we think of the task this way: Move the top $n - 1$ rings to an auxiliary peg, then move the largest ring, then move the smaller $n - 1$ rings onto the largest.

The way of viewing the problem leads to the following recursive definition for NUMMOVES:

```
TO NUMMOVES :N
TEST :N = 1
IF T OUTPUT 1
IF F OUTPUT ( 1 + 2 * NUMMOVES
 :N - 1 )
END
```

Trying this procedure, we find that NUMMOVES 2 = 3, and also that NUMMOVES 3 = 7. The reader might try to find a formula for NUMMOVES n, and also the value of NUMMOVES 64.

Of more interest is a procedure for actually solving the puzzles, and beyond that, for implementing the solution graphically. By the above reasoning, what we need is a procedure SOLVE with four inputs:

SOLVE *n peg1 peg2 peg3*

which would move the top *n* rings from *peg1* to *peg2* using *peg3*. Using the rules we obtain:

```
TO SOLVE :N :P1 :P2 :P3
TEST :N = 1
IF T GETRING :P1 SETRING :P2
IF F SOLVE :N - 1 :P1 :P3 :P2
SOLVE :N :P1 :P2 :P3
SOLVE :N - 1 :P3 :P2 :P1
END
```

To have LOGO print out the moves in order, we need to implement two procedures called GETRING and SETRING.

In the meantime, let's implement GETRING and SETRING simply so we can test our solution:

```
TO GETRING :P
TYPE [ PICK UP ] PRINTCHAR 32
TYPE :P PRINTCHAR 32
END
```

```
TO SETRING
TYPE [SET ON ] PRINCHAR :X2
PRINT P
END
```

Now, if we enter SOLVE 2 "A "B "C, the output will be:

PICK UP A SET ON C
PICK UP A SET ON B
PICK UP C SET ON B

The number of moves for three rings is 3, as expected. What will be the seven moves for SOLVE 3 "A "B "C? Try it!

We've looked at a LOGO procedure for solving the Tower of Hanoi as an abstraction. This procedure, SOLVE, prints out—as a list—the sequence of moves necessary for the solution. But given the graphics power of LOGO, we should be able to design a program—a series of procedures—which will represent the actual movement of rings from one peg to another graphically. And, in fact we can use LOGO's MAKECHAR command to define the required graphics, called *tiles*, and we can move these newly-defined tiles about, using LOGO procedures. So let's begin at the beginning.

Let A, B, and C be the three pegs. When we know which rings are on which pegs, we then know the particular state of the puzzle. In our LOGO implementation, the variables A, B, and C will be the names for lists which tell us which rings are on each peg. Our puzzle will have 8 rings. Let us number them 1 through 8 in order of increasing size. The beginning position, with all rings on peg A, is represented by :A = [1 2 3 4 5 6 7 8], :B = [ ], and :C = [ ]. Moving the top ring from A onto B results in the state :A = [2 3 4 5 6 7 8], :B = [1], :C = [ ]. In essence, a move consists of removing a number from the beginning of one list and adding it to the beginning of another list. At the same time, of course, the graphic representation ring must be erased and redisplayed in the correct position.

Let us first construct a procedure HANOI, which will allow us to play with the puzzle and then, when we want, solve it automatically.

```
TO HANOI
INITIALIZE
SETUP
PLAY
SOLVE 8 "A "B "C
END
```

INITIALIZE should set colors and define constants. SETUP should display the puzzle with all the rings on peg A. PLAY should allow us to pick rings up and put them down by simply pressing the names of the corresponding pegs. Play might continue until 'Q' is pressed. The puzzle should then be redisplayed and solved automatically, beginning with the rings on peg B. The procedure SOLVE was developed in the previous section. Procedures SETUP, PLAY, and SOLVE will depend on workhorse procedures GETRING and SETRING. The requirements for INITIALIZE will become apparent as we make choices about representation.

Assume that INITIALIZE assigns the value 8 to N and :TOP is the number of the ring to be displayed. Then SETUP can be:

```
TO SETUP
STAND :A
START :B
START :C
MAKE "TOP N
MAKE :A
MAKE :B
MAKE :C
REPEAT N [SETRING :A MAKE [TOP]
P :TOP - 1]
END
```

Using utilities MEMBER?, EMPTY, and ALARM, we can write PLAY in such a way as to validate all inputs. We want to accept either 'Q' or to stop PLAY the letters A, B, and C only. (VALID will be initialized to [A B C].) We also want to prevent an attempt to remove a ring from an empty peg. If an error is made, we will cause an alarm to be sounded. (See the listing for definitions of the utilities.)

```
TO PLAY
MAKE "X RC
IF NOT MEMBER? :X VALID THEN
IF NOT MEMBER? :X :VALID THEN
IF EMPTY? THING :X THEN ALARM
MAKE :X
GETRING :X
:X
MAKE : RC
IF NOT MEMBER? :X VALID THEN
ALARM GO :Q
SETRING :X
END
```

In this procedure, note that the value of X, :X, is the name of a peg, either A, B, or C. One might expect that the value of :X would be denoted :X, but this denotes the value of ':X'. The primitive THING must be used. THING :X is the list named by :X.

In order to design GETRING and SETRING, we need to be specific about how to represent the graphics. We could use the turtle, but we choose tiles because this allows the most colorful display. The LOGO screen is divided into 32 columns numbered 0 to 31 from left to right, and 24 rows numbered 0 to 23 from top to bottom. We can place the rings on the display by locating them relative to their

pegstands. Let ABASE, BBASE, and CBASE name the coordinates for the centers of the pegstands. Reasonable choices are: :ABASE = [7 21], :BBASE = [25 21], and :CBASE = [16 11]. Suppose a ring is the top one on a given peg. Its center has as its column coordinate the same column coordinate as the peg, and its row coordinate is equal to the row coordinate of the base minus as many rings as are on the peg. If we use TOP, COL, and ROW to contain the number of the top ring and its column and row coordinates respectively, we are led to:

```
TO SETRING :P
MAKE :P SE :TOP THING :P
MAKE :RCOORD THING WORD :P BA
SE 1
MAKE :COL FIRST :RCOORD
MAKE :X COUNT THING :P
MAKE :ROW LAST :RCOORD
END

TO GETRING :P
MAKE :RCOORD THING WORD :P BA
SE 1
MAKE :TOP FIRST THING :P
MAKE :COL FIRST :RCOORD
MAKE :X COUNT THING :P
MAKE :ROW LAST :RCOORD
END

TO ERASERING
MAKE :P SE :TOP THING :P
END
```

In using these procedures, :P is a letter (A, B, or C). Thus WORD :P "BASE will return the word ABASE, BBASE or CBASE. Note how BF (BUTFIRST) and SE (SENTENCE) are used to change the value of :P (which will equal A, B, or C). By passing the name of the peg, we can change its value. This would not be the case if we passed the value of the peg to the procedure. (Computer scientists call this passing parameters "by reference" rather than "by value.")

We are left with the problem of actually displaying the pegs and displaying and removing the rings. The work will be done by STAND, DISPLAYING, and ERASERING. We need to choose the tiles and colors.

The bases will use tile 96 and be black. The pegs will use tiles 104 and 105, and be white. Tile 104 is square, and tile 105 is rounded at the top. Recall that the number of rings is :N, and the division in LOGO is integer division.

```
TO STAND :P
MAKE :RCOORD THING WORD :P BA
MAKE :COL FIRST :RCOORD
MAKE :ROW LAST :RCOORD
REPEAT 3 [
PUT 96 COL + 1 ROW 1 MAKE ...
REPEAT :N [
PUT 104 COL ROW ... ]
PUT 105 COL ROW
END
```

Tiles and colors for the rings will be chosen as follows: The shapes for the tiles are designed so that ring k appears to be $k + 2$ tiles wide, but it is actually $3 + 2*(k/2)$ tiles wide. The accompanying figure shows the number and shape of all the required tiles, which we will have to make using MAKECHAR.

Ring	Tiles	Color	Tiles wide
1	112,113,114	Red	3
2	120,121,122	Orange	5
3	128,129,130	Yellow	5
4	136,137,138	Lime	7
5	144,145,146	Olive	7
6	152,153,154	Sky	9
7	160,161,162	Blue	9
8	168,169,170	Purple	11

A ring appears when the right number of tiles of the right shape and color are displayed. A ring is erased by displaying blanks and the peg tile. For effect, the rings will be displayed from the center out and erased from the outside in.

```
TO ERASERING
MAKE :J 1
REPEAT 1 + :TOP / 2 [
PUT 104 COL + :J ROW 1 2 COL - :J
PUT 104 COL - :J ROW 1
MAKE :J :J + 1 ]
PUT 104 COL ROW
END

TO DISPLAYING
PUT 184 + :TOP * 8 COL ROW
MAKE :MID TOP * 8 + :TOP + 8
MAKE :LFT 184 + :TOP * 8
MAKE :J 1
REPEAT :TOP / 2 [
PUT :MID COL + :J ROW 1 COL
PUT :LFT COL - :J ROW 1
MAKE :J :J + 1 ]
PUT :LFT COL - :TOP / 2 ROW 1
END
```

We are almost ready to play with the puzzle. INITIALIZE (see listing) defines colors for the tiles, and assigns values to N, VALID, ABASE, BBASE and CBASE.

Ring	No.	Color	No.		No.	Color	Part	No. Req'd		
1	112	Red	114		96	Black	Base	9 x 3		105 White
3	128	Yellow	130		104	White	Peg	16		Peg Top
5	144	Olive	146		113	Red	Ring 1	5		3 Req'd
7	160	Blue	162		121	Orange	Ring 2	5		
					129	Yellow	Ring 3	5		
Ring	**No.**	**Color**	**No.**		137	Lime	Ring 4	5		
2	120	Orange	122		145	Olive	Ring 5	5		
4	136	Lime	138		153	Sky	Ring 6	7		
6	152	Sky	154		161	Blue	Ring 7	7		
8	168	Purple	170		169	Purple	Ring 8	9		

Before anything will happen, though, the tiles must be defined using MAKECHAR. (See figures.) Then, ENJOY! Recall that to manipulate the rings, you just need to press the letter of the peg from which you want to take, or to which you want to add a ring. Use the procedure HELP if you forget.

After you have had some fun with the puzzle, you might want to try a four peg variation. To implement the four peg version, do the following:

Change INITIALIZE to include:

```
MAKE "VALID [ A B C D ]
MAKE "ABASE [ 8 10 ]
MAKE "BBASE [ 24 10 ]
MAKE "CBASE [ 8 23 ]
MAKE "DBASE [ 24 23 ]
```

In SETUP, add:

```
MAKE "D [ ]
STAND "D
```

The puzzle should then contain four pegs: A, B, C, and D. It can be manipulated just like the three peg puzzle. The automatic solution will still use just three pegs. But as a worthy challenge, you might try to write a better version of SOLVE which takes advantage of the fact that there are two auxiliary pegs instead of just one. The puzzle should take fewer moves to solve. How many less than $2n - 1$ moves are required if there are n rings and four pegs? I would be interested in any of your results. Then can five pegs be fit on the screen. . . ?

But if you are looking for a lesser challenge, or just want to experiment with a simpler puzzle, note that the number of rings is set in INITIALIZE and can be changed. Try this: Enter INITIALIZE, and then MAKE "N 5 (or some other integer). If you now enter SETUP, a puzzle with 5 rings will be displayed. Enter PLAY, and you can manipulate this puzzle until you press Q. Now enter SETUP again, and then SOLVE 4 "A "C "B. This will cause four rings to be moved automatically to peg C. Then enter PLAY and you can complete the puzzle by yourself. With LOGO, the procedures are your own to do with or modify as you please. Use your imagination, make up other puzzles, or just go ahead and play with this section's puzzle as is.