TANDY®

Cat. No. 26-2717

SUPER LOGO

Creative Programming and Graphics for the Color Computer

TERMS AND CONDITIONS OF SALE AND LICENSE OF TANDY COMPUTER EQUIPMENT AND SOFTWARE PURCHASED FROM RADIO SHACK COMPANY-OWNED COMPUTER CENTERS, RETAIL STORES AND RADIO SHACK FRANCHISEES OR DEALERS AT THEIR AUTHORIZED LOCATIONS

LIMITED WARRANTY

I. CUSTOMER OBLIGATIONS

A. CUSTOMER assumes full responsibility that this computer hardware purchased (the "Equipment"), and any copies of software included with the Equipment or licensed separately (the "Software") meets the specifications, capacity, capabilities, versatility, and other requirements of CUSTOMER.

B. CUSTOMER assumes full responsibility for the condition and effectiveness of the operating environment in which the Equipment and Software are to function, and for its installation.

II. LIMITED WARRANTIES AND CONDITIONS OF SALE

A. For a period of ninety (90) calendar days from the date of the Radio Shack sales document received upon purchase of the Equipment. RADIO SHACK warrants to the original CUSTOMER that the Equipment and the medium upon which the Software is stored is free from manufacturing defects. **This warranty is only applicable to purchases of Tandy Equipment by the original customer from Radio Shack company-owned computer centers, retail stores, and Radio Shack franchisees and dealers at their authorized locations**. The warranty is void if the Equipment's case or cabinet has been opened, or if the Equipment or Software has been subjected to improper or abnormal use. If a manufacturing defect is discovered during the stated warranty period, the defective Equipment must be returned to a Radio Shack Computer Center, a Radio Shack retail store, a participating Radio Shack franchisee or a participating Radio Shack dealer for repair, along with a copy of the sales document or lease agreement. The original CUSTOMER'S sole and exclusive remedy in the event of a defect is limited to the correction of the defect by repair, replacement, or refund of the purchase price, at RADIO SHACK'S election and sole expense. RADIO SHACK has no obligation to replace or repair expendable items.

B. RADIO SHACK makes no warranty as to the design, capability, capacity, or suitability for use of the Software, except as provided in this paragraph. Software is licensed on an "AS IS" basis, without warranty. The original CUSTOMER'S exclusive remedy, in the event of a Software manufacturing defect, is its repair or replacement within thirty (30) calendar days of the date of the Radio Shack sales document received upon license of the Software. The defective Software shall be returned to a Radio Shack Computer Center, a Radio Shack retail store, a participating Radio Shack franchisee or Radio Shack dealer along with the sales document.

C. Except as provided herein no employee, agent, franchisee, dealer or other person is authorized to give any warranties of any nature on behalf of RADIO SHACK.

D. **EXCEPT AS PROVIDED HEREIN, RADIO SHACK MAKES NO EXPRESS WARRANTIES, AND ANY IMPLIED WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE IS LIMITED IN ITS DURATION TO THE DURATION OF THE WRITTEN LIMITED WARRANTIES SET FORTH HEREIN.**

E. Some states do not allow limitations on how long an implied warranty lasts, so the above limitation(s) may not apply to CUSTOMER.

III. LIMITATION OF LIABILITY

A. **EXCEPT AS PROVIDED HEREIN, RADIO SHACK SHALL HAVE NO LIABILITY OR RESPONSIBILITY TO CUSTOMER OR ANY OTHER PERSON OR ENTITY WITH RESPECT TO ANY LIABILITY, LOSS OR DAMAGE CAUSED OR ALLEGED TO BE CAUSED DIRECTLY OR INDIRECTLY BY "EQUIPMENT" OR "SOFTWARE" SOLD, LEASED, LICENSED OR FURNISHED BY RADIO SHACK, INCLUDING, BUT NOT LIMITED TO, ANY INTERRUPTION OF SERVICE, LOSS OF BUSINESS OR ANTICIPATORY PROFITS OR CONSEQUENTIAL DAMAGES RESULTING FROM THE USE OR OPERATION OF THE "EQUIPMENT" OR "SOFTWARE." IN NO EVENT SHALL RADIO SHACK BE LIABLE FOR LOSS OF PROFITS, OR ANY INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY BREACH OF THIS WARRANTY OR IN ANY MANNER ARISING OUT OF OR CONNECTED WITH THE SALE, LEASE, LICENSE, USE OR ANTICIPATED USE OF THE "EQUIPMENT" OR "SOFTWARE."**
NOTWITHSTANDING THE ABOVE LIMITATIONS AND WARRANTIES, RADIO SHACK'S LIABILITY HEREUNDER FOR DAMAGES INCURRED BY CUSTOMER OR OTHERS SHALL NOT EXCEED THE AMOUNT PAID BY CUSTOMER FOR THE PARTICULAR "EQUIPMENT" OR "SOFTWARE" INVOLVED.

B. RADIO SHACK shall not be liable for any damages caused by delay in delivering or furnishing Equipment and/or Software.

C. No action arising out of any claimed breach of this Warranty or transactions under this Warranty may be brought more than two (2) years after the cause of action has accrued or more than four (4) years after the date of the Radio Shack sales document for the Equipment or Software, whichever first occurs.

D. Some states do not allow the limitation or exclusion of incidental or consequential damages, so the above limitation(s) or exclusion(s) may not apply to CUSTOMER.

IV. SOFTWARE LICENSE

RADIO SHACK grants to CUSTOMER a non-exclusive, paid-up license to use the TANDY Software on **one** computer, subject to the following provisions:

A. Except as otherwise provided in this Software License, applicable copyright laws shall apply to the Software.

B. Title to the medium on which the Software is recorded (cassette and/or diskette) or stored (ROM) is transferred to CUSTOMER, but not title to the Software.

C. CUSTOMER may use Software on one host computer and access that Software through one or more terminals if the Software permits this function.

D. CUSTOMER shall not use, make, manufacture, or reproduce copies of Software except for use on **one** computer and as is specifically provided in this Software License. Customer is expressly prohibited from disassembling the Software.

E. CUSTOMER is permitted to make additional copies of the Software **only** for backup or archival purposes or if additional copies are required in the operation of **one** computer with the Software, but only to the extent the Software allows a backup copy to be made. However, for TRSDOS Software, CUSTOMER is permitted to make a limited number of additional copies for CUSTOMER'S own use.

F. CUSTOMER may resell or distribute unmodified copies of the Software provided CUSTOMER has purchased one copy of the Software for each one sold or distributed. The provisions of this Software License shall also be applicable to third parties receiving copies of the Software from CUSTOMER.

G. All copyright notices shall be retained on all copies of the Software.

V. APPLICABILITY OF WARRANTY

A. The terms and conditions of this Warranty are applicable as between RADIO SHACK and CUSTOMER to either a sale of the Equipment and/or Software License to CUSTOMER or to a transaction whereby Radio Shack sells or conveys such Equipment to a third party for lease to CUSTOMER.

B. The limitations of liability and Warranty provisions herein shall inure to the benefit of RADIO SHACK, the author, owner and or licensor of the Software and any manufacturer of the Equipment sold by Radio Shack.

VI. STATE LAW RIGHTS

The warranties granted herein give the **original** CUSTOMER specific legal rights, and the **original** CUSTOMER may have other rights which vary from state to state.

12/84

Radio Shack Super LOGO

By

Larry Kheriaty

and George Gerhold

First Edition

Super LOGO program:
©1984 Micropi
All Rights Reserved.
Licensed to Tandy Corporation.

Super LOGO manual:
©1984 Micropi
All Rights Reserved.
Licensed to Tandy Corporation.

Reproduction or use, without express written permission from Micropi and Tandy Corporation, of any portion of this manual is prohibited. While reasonable efforts have been taken in the preparation of this manual to assure its accuracy, Micropi and Tandy Corporation assume no liability resulting from any errors or omissions in this manual, or from the use of the information obtained herein.

Please refer to the Software License in the front of this manual for limitations on the use and reproduction of this Software package.

FOREWORD

For more than a decade, the authors have been involved in the use of computers in education, and particularly with Computer-Assisted Instruction. Our experience made us aware of problems in getting students started on the right track in programming, and of LOGO's potential to help solve those problems. We decided that it would be worthwhile to develop a version of LOGO which ran on low-cost hardware and allowed relatively long sets of procedures. Color LOGO met those objectives and allowed the addition of some features, notably multiple turtles. Now more memory is available at low cost, and we have the benefit of feedback from many users. Super LOGO makes use of both those developments; larger ROM packs made addition of list processing and decimal arithmetic practical, and the feedback showed us many small improvements which make the package more useful.

Our debt to the original designers of the LOGO language is acknowledged in Chapter 1. We would also like to thank our children Aaron, Jenell, and Kirstin, whose responses to early versions of Color LOGO convinced us that we were on the right track.

George Gerhold

Larry Kheriaty

Table of Contents

Introduction 1

1. A Bit About Color LOGO 3

2. Getting Started 5

3. Repeat 11

4. Modes and Editing 15

5. Procedures 19

6. Subprocedures 23

7. Variables 29

8. Colors 33

9. Other Turtle Commands 37

10. Saving, Loading, and Printing Procedures 41

11. Recursion 45

12. DOODLE Mode—Procedures Without Typing 55

13. One Key Doodling 59

14. Use of DOODLE Mode and OK Set 63

15. Additional Editing Features 75

16. Multiple Turtles 77

17. New Shapes for Turtles 87

18. Turtle Games 97

19. Word and List Operations 107

20. Communication Between Procedures 111

21. Interactive Procedures 117

22. Playing with Words and Sentences 123

23. Generating and Sorting Lists 129

24. Card Games 135

25. Word Games 145

26. Dice Games 149

27. Grab Bag 153

Appendix: Language Summary 161

Index 185

INTRODUCTION

Radio Shack Super LOGO is an educational computer language. The language can be used to draw pictures on the computer's video display, using a shape on the screen called a "turtle," and it can be used to manipulate lists of words.

The graphics portion of Super LOGO is designed to let children learn by exploring. Children plan an action, then enter simple commands that move the turtle forward or back, or turn it in any direction. Here are a few of the special features of Super LOGO:

- Line-oriented editing allows you to write and save sequences of turtle moves (called "procedures").
- A "doodle mode" lets children who are too young to read or type use the program.
- A "SLOW" command lets you control how fast the turtle moves.
- Screen colors can be changed.
- Animation is possible with Super LOGO.
- Variables and arithmetic expressions can be used in the sets of turtle moves that you write and save.
- Multiple turtles can work in concert on graphics and on list-processing tasks.

Super LOGO is a language for beginners. For this reason, the Super LOGO manual has been written to guide you through use of the language, step by step, with many examples and illustrations. Here is a summary of the organization of the manual:

1. Chapters 1 through 11 introduce turtle graphics, the LOGO syntax, and use of the editor. Readers who are already familiar with LOGO may wish to skim these chapters or to bypass them in favor of the summary in Appendix I.
2. Chapters 12 through 18 cover features unique to Super LOGO.
   a. Chapters 12 through 14 provide hints for using Super LOGO with very young children.
   b. Chapter 15 provides more information about using the Super LOGO editor.
   c. Chapters 16 through 18 introduce the use of multiple turtles and new turtle shapes.
3. Chapters 19 through 26 cover list processing, including multi-tasking applications of list processing.
4. Chapter 27 contains sample sets of more complex turtle moves that you may wish to explore.

The Radio Shack Super LOGO program is available in three versions. The Disk version (26-2716) requires a 32K disk-based Tandy Color Computer with Color BASIC. The Network 2 version (26-2738) is designed for use with a Tandy Network 2 Controller, a 32K disk-based Tandy Color Computer as host system, and from one to sixteen 32K disk- or cassette-based Color Computers as student stations. The ROM version (26-2717) requires a 16K ROM-based Tandy Color Computer with Color BASIC. Procedures that you write can be saved on diskette or on cassette tape using the Disk version, on cassette using the ROM version, or on a diskette at the host computer using the Network 2 version.

1. A BIT ABOUT Super LOGO

Super LOGO is a computer language for children. Like all the best things for children, Super LOGO can provide endless fascination and challenge for adults as well. At first glance, Super LOGO may seem to be simply a language for drawing pictures because the result of running the procedures in the early sections of the manual is almost always a picture. However, Super LOGO is far more than an easy way to draw pictures. Super LOGO is a tool for learning about some of the most powerful concepts in mathematics, physical sciences, computer science, problem solving, and language syntax—but in a way so appealing and simple that "even a kid can do it."

Notice that we said that Super LOGO is a language for learning; we very intentionally did not say that it is a language for teaching. The role of the learner is all important. Super LOGO puts the student in the role of explorer, one who sets goals (problems to solve) and tries to find a way to those goals. The role of the teacher is guide, one who stays in the background as much as possible, one who does not set the goals for the learner, and one who assists only when asked. Effective use of Super LOGO has much of the flavor of play: "It's not whether you win or lose, but how you play the game." The goal the student reaches is not as important as the process of seeking the goal.

Super LOGO is based on a set of ideas for use of the computer. These ideas were first developed under the name "LOGO." Many people have contributed to the LOGO project — too many to list — but we must mention the names Wallace Feurzeig, Harold Abelson, Andrea diSessa, and, with special emphasis, Seymour Papert. Most of the development and testing of LOGO was done at MIT. There were two vital steps in bringing the LOGO approach to the attention of the educational community. One was the publication of two books: *Mindstorms* by Papert and *Turtle Geometry* by Abelson and diSessa. Any serious user of LOGO will want to read those books. The other was the implementation of the LOGO language on microcomputers, a step which decisively moved LOGO from the laboratory into the classroom. If you are already familiar with LOGO, you will find much of Super LOGO to be familiar too. Wherever possible we have kept the same syntax as LOGO, and the logical structures of the two languages are essentially the same. Most of the programs in books on LOGO will run in Super LOGO without change.

Super LOGO is not just LOGO under another name for another computer; there are some differences between the two. LOGO handles words and letters via a set of operations called list processing. Many versions of LOGO allow the advanced user to handle nested lists, but Super LOGO restricts the user to simple lists. Many versions of LOGO use floating point numbers in arithmetic; Super LOGO uses decimal arithmetic. Both of these restrictions are imposed to allow the following very significant additions. Super LOGO provides multiple turtles whereas most versions of LOGO provide only a single turtle. Super LOGO thus can be used to introduce important concepts like multi-programming and messages between independent procedures, but still with great simplicity. Consequences of multiple turtles include provision for simple animation and the potential for user-created games. All these are possible because, in contrast to LOGO, the memory requirements of Super LOGO are modest. Super LOGO also provides a mode for doodling, designed for children who are too young to type keywords reliably.

If you are just starting on computers, all that sounds rather complex. That's because we're just talking about it instead of doing it. Let's do it.

2. GETTING STARTED

The steps below tell you how to load Super LOGO into your Color Computer system. You will want to have some way to store your favorite creations for future display, so you will want to have a cassette recorder attached to your computer system. Consult the chapters on installation and operation in your copy of the Color Computer Operation Manual that came with your computer system, for instructions as to proper cable connections for the cassette player. DO NOT TURN ON THE POWER YET!

Loading Super LOGO Using a Color Computer Cassette System

1. With the computer's power *off,* plug the Super LOGO cartridge into the slot on the right side of your Tandy Color Computer. Check that the label is up and that the cartridge is seated firmly.

2. Turn power *on*. (The computer power switch is on the back left corner of the computer.) The screen will display the prompt:

SUPER LOGO COPYRIGHT 1984
LARRY KHERIATY & GEORGE GERHOLD
LICENSED TO TANDY CORP.
ALL RIGHTS RESERVED.

LOGO

Now turn to "**Using the Super LOGO Program**" on page 6.

Using the Super LOGO Program

You should now be in BREAK mode, which is indicated by the prompt which ends

LOGO:

at the left of the screen. When you are in any other mode, you can return to BREAK mode at any time by any one of three actions:

1. If you press the **BREAK** key, whatever you are doing will be interrupted and the computer will return you to BREAK mode. If a procedure (a program that you have written in LOGO) is actually running, you must press the **BREAK** key twice: once to interrupt the procedure, and a second time to get into BREAK mode.

2. The RESET button (located on the right rear side of the computer) will always return you to BREAK mode, but you will lose all programs in memory.

3. A complete restart (as described on page 5) will place you in BREAK mode.

BREAK mode will be covered in detail in Chapters 3 and 10. For now, let's move into RUN mode by pressing **R**. There is the turtle, sitting in the center of the screen facing straight up. Admittedly, this turtle does not bear a strong resemblance to the ordinary pond-type turtle; but, like an ordinary turtle, it can crawl forwards and backwards and it can turn right and left. Unlike ordinary turtles, computerized turtles can drag their tails to leave tracks (in colors) or raise their tails and not leave tracks. Turtles can even be made invisible.

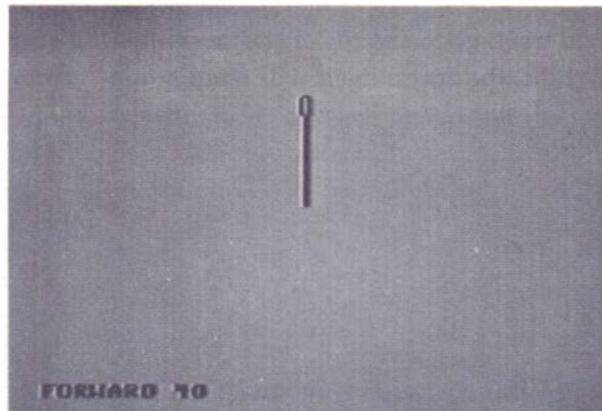
The name "turtle" was given originally to a tiny mechanical robot which could be made to crawl around the floor under computer control. The name probably had much more to do with the speed of the robot than with the shape of the robot. The track left by the turtle was called a turtle graphic. The term "turtle graphics" is now used to indicate a way of drawing where lines are described by a direction and a length (the alternative is to describe a line by giving the coordinates of the two end points of the line, a method called—strangely—vector graphics). The item which moves is called the turtle, even when it is just a shape on the screen. The graphics portion of Super LOGO is a language for controlling turtles.

We have a turtle in the center of the screen, itching for action. Let's tell the turtle to move forward. Simply type

FORWARD 40

Then press **ENTER**.

The number after "**FORWARD**" tells the turtle how far forward to move. After you enter **FORWARD 40** the screen will show a turtle track.



NOTE: If you forget to leave a space between "**FORWARD**" and "**40**", you'll see the message "**I DON'T KNOW HOW TO FORWARD 40**." You'll get a similar message if you make any other typing error. Just press ENTER to get another chance to enter "**FORWARD 40**." You can use the left-arrow key to correct typing errors before you press ENTER . Simply backspace to the beginning of the error, and retype the turtle instruction.

It won't be long before you get tired of typing **FORWARD** all the time, so there is an abbreviation which has the same effect. Enter the following (that is, type it and press ENTER):

FD 10

Try to get a feel for the screen size and resolution. Try

FD 1

It's almost too little to see. Then try with a larger number, like

FD 100

The turtle moved, but it didn't leave a complete track. When the turtle goes off the top of the screen, it reenters at the bottom, a process which is called "wrapping around." However, at this stage we have the bottom four lines of the screen reserved for text, so the portion of turtle track which passes through those bottom four lines does not appear.

Now let's find out how far it is from the center of the screen to the top. To get a fresh start and a clear screen, enter the word (not the single key)

CLEAR

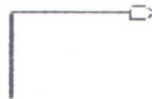
Then try to make the turtle track go to the top of the screen with a single **FD** command. When you have it exactly right, the turtle itself will wrap around (disappearing into the bottom four lines of the screen—disappearing because these lines are reserved for text), but the line will be drawn to the top of the screen. No doubt it will take you several tries of **CLEAR**, **FD** to hit the top exactly, using the smallest possible number.

By now, you're probably tired of drawing vertical lines. It's time to turn the turtle. Clear the screen (by typing **CLEAR**, then pressing [ENTER]), and enter these commands

```
FORWARD 40
RIGHT 90
```

to make the change more obvious, enter

```
FORWARD 50
```



```
RIGHT 90
FORWARD 50
```

The turtle understands degrees.

If you are using Super LOGO with small children, we have a suggestion. There is now quite a bit of information gathered about the effective use of LOGO with small children. LOGO is a language for experimentation, not a language to learn by imitation of items from a textbook. Resist any temptation to explain degrees to the child who does not already know about them. The child will learn about degrees easily from experimenting with LOGO.

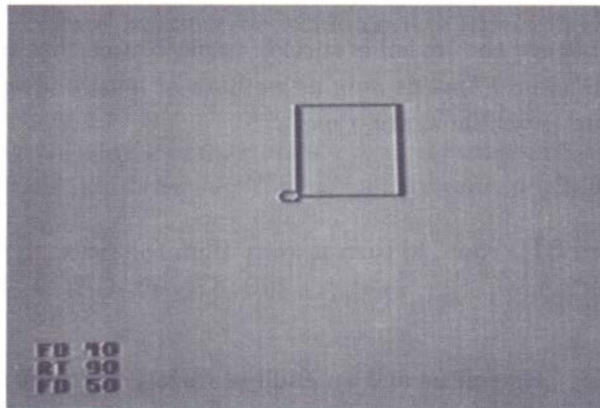
Again, we soon get tired of typing **RIGHT** so we abbreviate **RT**. Try

```
RT 90
```

(Think "right turn" for **RT**.) Now the turtle points down. We're half way to drawing a rectangle, so let's finish it. Enter

```
FD 40 RT 90
```

and see if you can finish it.



Let's look at one very important fact about turtle behavior. Clear the screen and enter

RT 45 RT 45

This produces the same heading as **RT 90**. When the turtle is told to turn, it turns that far from whatever its current heading is. We are telling the turtle how to change its heading; we are *not* telling the turtle to head toward some point. In the same way, when we tell the turtle to go forward we are telling the turtle how to change its position; we are not telling the turtle to go to some point on the screen. Thus the position and heading of the turtle after one of these commands will depend on where the turtle started.

So far, we have learned three primitive turtle commands. (Papert would say, three words in "turtle talk.") They are **CLEAR**, **FORWARD**, and **RIGHT**. With these three, we can draw any figure which will fit on the screen and which could be drawn on paper without lifting the pencil from the paper. You might try drawing a triangle (three-sided figure) and a pentagon (five-sided figure) for practice. If you're like us, you don't remember the angles for pentagons, so experiment.

We could go a long, long way with just **RIGHT** and **FORWARD**, but **LEFT** and **BACK** are useful too. Clear the screen and try

LEFT 90

(We could have used the abbreviation **LT** for left turn.) Now let's make the turtle move backwards. Try

BACK 40

(or, in abbreviated form, **BK 40**). Notice that the turtle is somewhat transparent. You can see the track through the turtle. If you'd rather not see the turtle at all, you can hide it. Enter

HIDETURTLE

(Here the abbreviation **HT** is much shorter.) The turtle is still there, but it is invisible. Type

LT 3Ø BK 3Ø

and then press **ENTER**, to see the invisible turtle's track. Notice that we can type more than a single turtle command on a line as long as we have at least one space between the commands. To make the turtle visible again, type

SHOWTURTLE

(you guessed it, abbreviated **ST**). Then, to turn it away from the track, type

LT 12Ø

At this point, the only thing between us and an endless variety of stunning graphics is an immense amount of typing. In the next two chapters, we'll learn some things which will save us from this immense amount of typing.

3. REPEAT

There are many times when we want to repeat a series of turtle commands several times. For example, if we wanted to draw a square we would need to repeat the sequence

```
FD 60 RT 90
```

four times. Fortunately, the turtle understands a control statement which saves us from typing the same thing four times in succession. The control statement is **REPEAT**. With a clear screen, try

```
REPEAT 4 (FD 60 RT 90)
```

```
REPEAT 4 (FD 60 RT 90)
```

Notice that after the statement **REPEAT** we must tell the turtle how many times to repeat, and we must tell the turtle what to repeat. The "what" is enclosed in parentheses. Here the number of times to repeat is 4, and the "what" to repeat is **FD 60 RT 90.**

The figure the turtle has just drawn may be more of a rectangle than a square. On some TV sets, the size of a turtle step in the vertical direction is a little different from the size of a turtle step in the horizontal direction. If you plan to use the TV mainly for LOGO, you may want to adjust the set for this difference. By adjusting the Vertical Size Control you should be able to change the rectangle into a square. However, you must understand that this will affect the proportions of everything you display on the TV, so don't make the change without considering other uses of the set.

Now that we have a way to repeat a short list of commands as often as we wish, we can draw a series of regular polygons. For example, a triangle

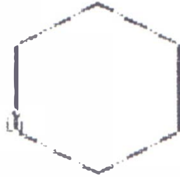
```
REPEAT 3 (FD 60 RT 120)
```

a pentagon

```
REPEAT 5 (FD 60 RT 72)
```

and a hexagon

```
REPEAT 6 (FD 40 RT 60)
```



```
REPEAT 6 (FD 40 RT 60)
```

Remember that you can use **CLEAR** as a command to start with a fresh screen whenever you want.

One of the most useful figures to draw is a circle. The idea is that one draws a circle by moving forward a bit and turning a bit, many times. For a complete circle to be drawn, the total of all the repeated turns must be at least 360 degrees. Two different sized circles could be drawn by the following:

```
REPEAT 360 (FD 1 RT 1)
REPEAT 180 (FD 1 RT 2)
```

Many other combinations are possible.

The figures drawn above are really 360- and 180-sided polygons instead of circles. They look like circles because of the finite resolution of the screen display. In fact, even cutting the number of sides down to 36 still gives a pretty good circle.

```
REPEAT 36 (FD 10 RT 10)
```

There are two advantages to using a smaller number of sides; the circle is drawn faster, and we can adjust the size of the circle in smaller steps. Try

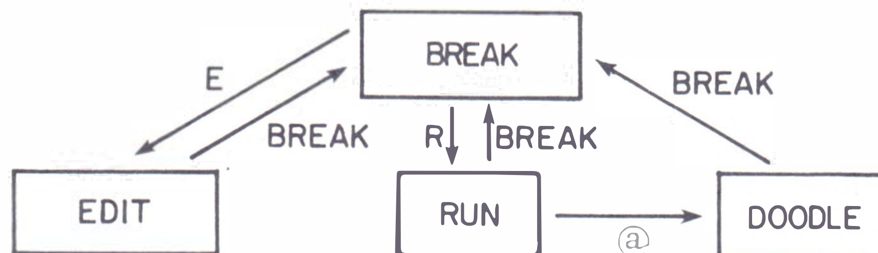
```
REPEAT 36 (FD 11 RT 10)
REPEAT 36 (FD 10 RT 10)
REPEAT 36 (FD 9 RT 10)
REPEAT 40 (FD 9 RT 9)
```

Several geometry lessons could be built around the examples given above. There is usually little problem in getting the number of sides correct (the number of times to repeat). The challenge is to get the angle of the turn correct. Let the user experiment with different angles until they get the correct angle. After they have gotten a few correct, see if they can generalize and predict the angle — say, for an 8-sided polygon (an octagon). It would be difficult to overstate the value of these kinds of activities. They are much closer to the activities of mathematicians and scientists than what is usually taught in science and mathematics courses.

4. MODES AND EDITING

New users of computers often find the idea of modes awkward. Mode is the term used to describe the separation of the various things a computer language can do into groups. There are a number of good reasons for having various modes. One is that there are not enough different keys on the keyboard to control all the different things that need to be done. The same keys can be used for different tasks in different modes without confusion (at least on the computer's part).

The following diagram is a map of the modes in Super LOGO.



The keys which trigger the jumps between modes are indicated on the arrows. You've already been in BREAK mode; that's the mode that you are in when you start. You've already been in RUN mode; you got there from BREAK mode by pressing [R]. Now we want to move into EDIT mode. The map shows us that we need to leave RUN mode (by pressing the [BREAK] key) and then get into EDIT mode (by pressing the [E] key).

EDIT mode provides a line-oriented editor. EDIT mode is used to create and alter programs written in Super LOGO, but for the rest of this chapter we will forget LOGO and concentrate on the mechanics of using EDIT mode. We'll do something familiar — write a note to Grandma.

When you get into EDIT mode, a short horizontal line appears at the start of the bottom line of the screen. This line is called the cursor. The cursor indicates where any typed letters, numbers, etc., will appear. Start the note by typing

```
DEAR GRANDMA,
```

Press [ENTER], and the cursor moves to the start of the next line. Type the next line as

```
I'M STARTING TO USE AN EDITOR.
```

Again press [ENTER] to complete the line. Notice that this editor produces only upper-case letters; Super LOGO uses only upper-case letters.

We could continue to enter as many lines as we wanted in the same fashion. Let's assume that this is to be a very short note and that we now want to quit editing. Press [BREAK]. Upon reflection we decide to alter the note, so we return to EDIT mode (press [E]). The first line of our note appears with the cursor at the start of the line.

We decide to change the word **STARTING** in the second line of the note to the word **BEGINNING**. To do this we first must display the second line and position the cursor under the **S** in **STARTING**. We move the cursor by use of the arrow keys. Up-arrow and down-arrow move the cursor to a different line, and left-arrow and right-arrow move the cursor within a line.

Changing lines always resets the cursor to the start of the line. Arrow commands which make no sense are ignored. Thus if we press right arrow when the cursor is under the comma following **GRANDMA**, nothing happens because there are no more characters on the line.

Let's go through this step by step. To see the second line of the note, press the up-arrow key once. Then press the right-arrow key several times to position the cursor under the **S** in **STARTING**. Then type

BEGIN

Notice that the overtyping simply replaces the letters. Now we have another kind of change to make because **BEGINNING** has one more letter than **STARTING**. We want space for another **N** before the **ING**. To create a space we hold down the [SHIFT] key and press the right-arrow key. Now we can type the extra **N** in the created space. Remember: to insert, press [SHIFT] right-arrow to create the space, then type in what you want.

Next let's change the line from

I'M BEGINNING TO USE AN EDITOR.

to

I'M LEARNING TO USE AN EDITOR.

Again position the cursor at the start of **BEGINNING** and overtype the characters you want to change. Here the problem is that an extra **N** remains. To delete a character (or space) hold down the [SHIFT] key and press the left-arrow key. Try it, and remember: press [SHIFT] left-arrow to delete.

Poor Grandma isn't going to know who the note is from unless we add a line at the end. Use the up-arrow key to move the cursor as far down as you can. It should be at the start of a blank line following the text. We want to skip a line before signing the note, so press [ENTER] once. Notice that when you press [ENTER], a line is added at the end. But if the cursor is within the text, pressing [ENTER] has the same effect as the up-arrow. Now space over and sign your name.

While we are at it, we should skip a line after **DEAR GRANDMA**. That is, we want to change

DEAR GRANDMA,
I'M LEARNING TO USE AN EDITOR.

LOVE, ANN

to:

DEAR GRANDMA,

I'M LEARNING TO USE AN EDITOR.

LOVE, ANN

Position the cursor at the beginning of the line "**I'M . . .**". Then hold down the [SHIFT] key and press the down-arrow key. Move the cursor down to check that you got what you wanted. Remember: to insert a new line, position the cursor at the start of the following line; then press [SHIFT] down-arrow.

We want to make one more change. We want to change the closing to

LOVE,
ANN

We want to break one line into two. Position the cursor where you want the break to occur; then press [SHIFT] down-arrow to break the line. You'll have to insert some spaces to move the name over as shown above.

This, we think, is the final form of the note, so we exit EDIT mode (press [BREAK]). To make a last check, we get back into EDIT mode (press [E]). To get the whole note on the screen without repeated pressing of up-arrow or [ENTER] we press [SHIFT] up-arrow twice. This will show us everything in memory. If we want to interrupt this process, just press any key to stop the scan. To restart the scan, press [SHIFT] up-arrow twice again. To jump back to the start of the text, press the [CLEAR] key.

That covers the basics of using the editor. You should practice a bit with it so that when we return to Super LOGO you can concentrate on the language and not have to worry about the mechanics of the editor. Chapter 15 covers the more advanced features of the editor.

To conclude this chapter, we give a summary of the editing features.

To:	Press:
get into EDIT mode	[BREAK], [E]
display the next line of text	[↑] or [ENTER] ([↑] has no effect at last line)
add a line at end of text	[ENTER] , then type the line
move text down one line	[↓] (no effect at top line)
move cursor right	[→] (no effect at line end)
move cursor left	[←] (no effect at line start)

replace character	position cursor, overtype
insert character	position cursor, SHIFT → (no effect if line full), then type character
delete character (or delete blank line)	position cursor, SHIFT ←
insert line	position cursor at start of following line, SHIFT ↓
break line	position cursor at break point, SHIFT ↓
return to top line	CLEAR
scroll or scan through text	SHIFT ↑ twice
stop scroll or scan	any key
delete from cursor to end of line	SHIFT CLEAR
search for a word	SHIFT ↑ followed by the word, followed by ENTER

5. PROCEDURES

You have now mastered five primitive turtle commands (**CLEAR**, **FORWARD**, **BACK**, **RIGHT**, and **LEFT**). Next we want to combine these commands into a unit which we call a procedure. The first step is to tell the computer not to obey each command as it is typed, but to store the commands. This is what happens in EDIT mode. Press [BREAK], then hold [SHIFT] down and press [CLEAR] (to clear the memory of old programs). Then get into EDIT mode (press [E]).

The screen should be blank with the cursor in the lower left corner. If the screen is not blank, return to BREAK mode (by pressing the [BREAK] key), hold the [SHIFT] key down, and press the [CLEAR] key firmly. Return to EDIT mode by pressing [E].

You are now using a line-oriented editor. We will practice using the editor as we create and edit procedures. Our first exercise will be to write a procedure for drawing a rectangle. First we must give the procedure a name. We'll call this first one "**RECTANGLE**." The first line of the procedure contains the name, and we let the computer know that we're naming a procedure by starting the first line with the keyword "**TO**." To name this first procedure **RECTANGLE**, enter

TO RECTANGLE

Procedure names must fit on a single line, must contain no spaces, and must not be the same as any of the keywords or abbreviations (for example, **REPEAT**, **FORWARD** or **FD**). The keyword **TO** must begin in column 1.

If you made a typing error when you were using RUN mode, you got the error message "**I DON'T KNOW HOW TO**" followed by your mistyped command. Because a procedure name can be almost anything, the computer assumes that any characters which don't form a correct keyword must form a procedure name. If the characters are really a typing error, then the name is not found in the list of procedures and the error message is sent.

Next type in the turtle commands for drawing the rectangle. That is, type

```
FD 5Ø RT 9Ø FD 3Ø RT 9Ø FD 5Ø
RT 9Ø FD 3Ø
```

Many commands can be typed on a single line as long as they are separated by one or more spaces. To finish the procedure, type

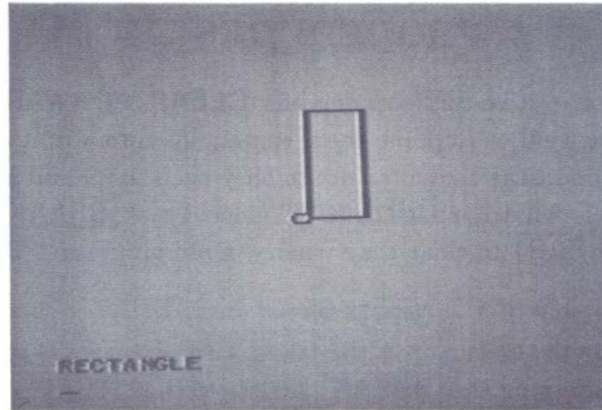
END

on a new line and press [ENTER].

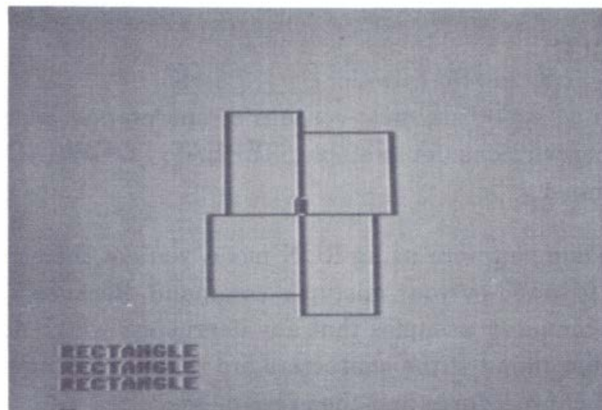
To try out **RECTANGLE**, you must leave EDIT mode (by pressing the [BREAK] key) and then get into the RUN mode (by pressing the [R] key). To actually run the procedure, type

RECTANGLE

and press [ENTER].



That's so neat that we should try it again and again. Type and enter the procedure name at least three more times. Now the screen should show



By placing the procedure **RECTANGLE** in the computer's memory, we have taught the turtle to understand a new word. The turtle now understands **RECTANGLE** in the same way that it understands **LEFT**, **RIGHT**, **FORWARD**, and **BACK**.

Before moving on to other procedures, we want to review use of the editor. Press [BREAK] to return to BREAK mode; then press [E] to reenter EDIT mode. The screen should now show the first line of the procedure **RECTANGLE**. Let's change the name to **BOX**. Use the right-arrow key [→] to position the cursor under the **R** in **RECTANGLE**. Then type **BOX**. Remember, overtyping replaces characters. We need to delete the remaining letters, which we do by holding down the [SHIFT] key and pressing the left-arrow key [←]. We can see the rest of the lines in the procedure by pressing either [ENTER] or the up-arrow key [↑] several times.

It is good programming practice to clarify the structure of a procedure by indentation. Here we want the procedure **BOX** to look like this

```
TO BOX
  FD 50 RT 90 FD 30 RT 90 FD 50
  RT 90 FD 30
END
```

To make these changes we must insert a couple of spaces at the beginnings of the second and third lines. Move the second line to the bottom of the screen by using the up- and down-arrow keys. The cursor will move to the start of the line whenever you change lines. To insert spaces, hold down the SHIFT key and press the right-arrow key. If this does not insert spaces, it means that the line is already full. Insert spaces at the start of line 3 as well.

The structure of the procedure would be even clearer if it were typed as follows.

```
TO BOX
  FD 50 RT 90
  FD 30 RT 90
  FD 50 RT 90
  FD 30
END
```

These changes require us to break single lines into multiple lines. To break a line, position the cursor where you want to break the line, hold the SHIFT key down and press the down-arrow key.

What if we want to add lines to a procedure; for example if we want to add a diagonal line through the box? We'd have to tell the turtle to turn and go forward. You'd better run **BOX** to get an estimate of the angle and distance (remember press BREAK, then press R, then enter **BOX**). The turtle needs to be turned more than 90 degrees to point along the diagonal. Make a guess and return to EDIT mode (BREAK, E). Now place the cursor under the **E** in **END**; hold down SHIFT and press the down-arrow key. This inserts a blank line (try the up-arrow key to check that **END** has just been bumped down one line). You can now insert your **RT** and **FD** commands in this new blank line. It will no doubt take you several tries to get the angle and length exactly right; that will give you good practice in bouncing back and forth between RUN and EDIT modes. (No fair using your knowledge of trigonometry; with turtles you are supposed to experiment.)

In this chapter we have covered two main topics. We have learned how to enter and change multiple command procedures, and we have learned how to teach the turtle to understand more complex commands via procedures.

6. SUBPROCEDURES

Once we have taught the turtle a new word by writing a procedure, we can use that new word in other procedures. Return to EDIT mode and remove the commands for drawing the diagonal (we used **RT 122 FD 59**) from **BOX**. Now move to a new line (press ENTER). In fact a blank line between procedures will help keep things easy to read, so press ENTER again. We're going to write another procedure to draw the pattern of four boxes. We'll call it **FOUR**, so type

```
TO FOUR
  BOX
  BOX
  BOX
  BOX
END
```

Notice that we've used **BOX** as a turtle command in the same way that we used **FORWARD** and **RIGHT** within **BOX**. Run **FOUR** to see that it works. The result is the same as that shown on page 20.

To run the procedure **FOUR**, the computer must have available the subprocedure **BOX**. Both procedures must be in the program space when **FOUR** is run, but their order within that space is of no importance. We could have written **FOUR** first and then written **BOX** with exactly the same result.

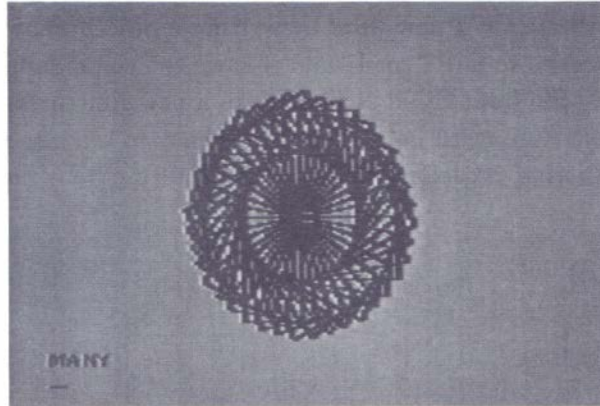
The procedure **FOUR** can be shortened by use of the **REPEAT** control statement. The altered form of **FOUR** is

```
TO FOUR
  REPEAT 4 (BOX)
END
```

The space after the number **4** is optional. The parentheses can include a whole list of turtle commands and subprocedure names. The list in parentheses can extend over many lines, but the parentheses are essential.

Now that we have taught the turtle what **FOUR** means, we can move to a higher level procedure. Try

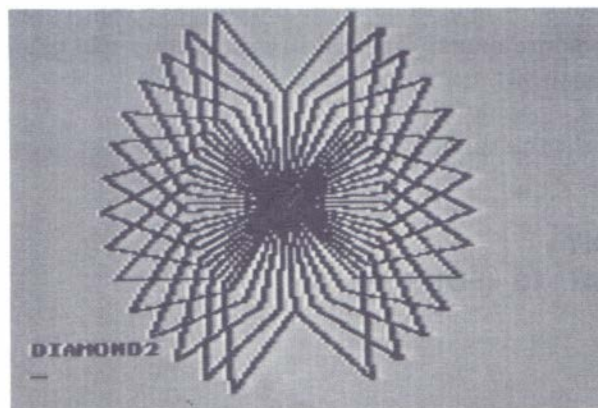
```
TO MANY
  REPEAT 10 (FOUR RT 9)
END
```



You are probably tired of following the manual and are consumed with curiosity. What will happen if I change the number on the **REPEAT** in **MANY**; what will happen if I change the angle in **MANY**; what will happen if I restore the commands to draw the diagonal in **BOX**? Don't hesitate to find out by trying; that's the whole point of Super LOGO. Try triangles, pentagons, hexagons, threes and fives instead of just boxes and fours.

Here is another sample.

```
TO DIAMOND
  FD 5Ø LT 45 FD 5Ø LT 135
  FD 5Ø LT 45 FD 5Ø
END

TO DIAMOND2
  REPEAT 29 (DIAMOND RT 4Ø)
END
```



When you run **DIAMOND2**, the pattern on the screen is missing the bottom portion. That is because the computer is using a split screen; the top of the screen is reserved for graphics, and the bottom four lines are reserved for text. The turtle draws behind the text portion of the screen; it does not wrap around until it reaches the very bottom of the whole screen. However, when the computer is using a split screen, we cannot see what the turtle draws behind the text area.

We can see what the turtle draws on the full screen by entering one of the statements

DRAW or **FULLSCREEN** (abbreviated **FS**)

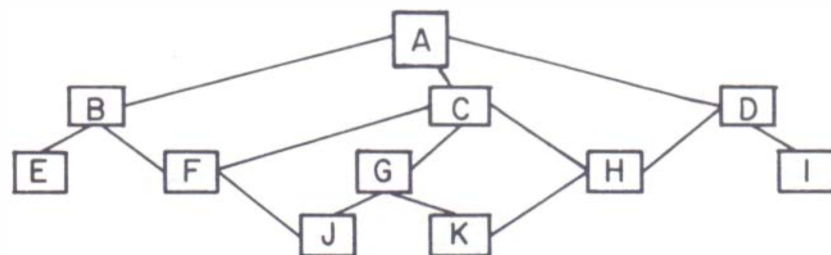
before entering the procedure name. Of course, this can all be on one line; for example

DRAW DIAMOND2

Be careful of one thing when using the full screen. If you type more than three lines of commands in RUN mode, the text lines at the bottom of the screen will scroll up. In full screen, these lines could contain turtle tracks or even turtles. The scrolling up will mess up your pattern and at times might make it appear that there are two turtles. Therefore, avoid giving more than three lines of commands when using the full screen.

The easiest way to return to split screen is to press **BREAK** and then return to RUN mode.

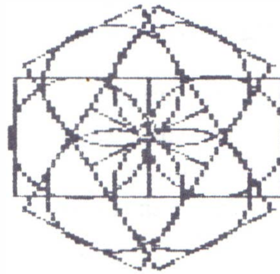
Super LOGO is a structured language. A complex program written in Super LOGO could have the following structure.



Each letter within a box represents a procedure; each line of type on the page includes the subprocedures of a particular level; the lines indicate which subprocedures are used by each procedure. There are four levels of procedures within this program. The master procedure **A** (level Ø) might use the subprocedures of level 1 in the order **B**, **C**, **D**, **C**. Subprocedure **B** might use the subprocedures of level 2 in the order **E**, **F**, **E**; subprocedure **C** might use the subprocedures of level 2 in the order **G**, **F**, **H** etc. Notice that subprocedures can be used many times and many places within the overall program.

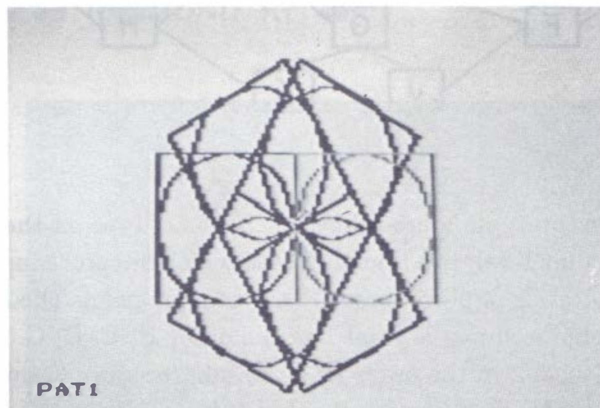
So far, in our examples, we have been working from the bottom up, defining a first procedure, then writing a second procedure that uses the first procedure as a subprocedure, etc. That is typical of programming manuals where the emphasis is on the mechanics of a language instead of on problem solving. It is interesting to adopt that approach with Super LOGO at times because the results are often unpredictable. However, as we become more serious we often will have a problem we wish to solve. Then we should work from the top level down. Now we illustrate that process.

The sample problem is to create the following pattern.



PATTERN

Obviously, the figure is so symmetrical that it must contain a repeated pattern. Counting shows that something is repeated six times. The crucial step is to recognize that the element that is repeated six times is a square with a circle inside.



Therefore our main procedure could be

```
TO PATTERN
  REPEAT 6 (SQUARE—CIRCLE RT 6Ø)
END
```

The six-fold symmetry tells us to repeat 6 times with turns of 6Ø (because 6 * 6Ø = 36Ø). As yet we have no idea how to draw a square with a circle inside. Notice that we use the hyphen as part of the procedure name to avoid using a space; a space between the two words would indicate two subprocedures. We can use a hyphen this way any place it cannot be confused with a minus sign.

Now we move to the next lower level.

```
TO SQUARE-CIRCLE
  CIRCLE
  SQUARE
END
```

Again we break the task into simpler tasks. This time the breakdown is obvious; you draw a square around a circle by drawing a circle and then a square.

Now we drop down to level 2. The obvious procedure for drawing a circle is

```
TO CIRCLE
  REPEAT 36Ø (FD 1 RT 1)
END
```

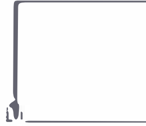
This gives a circle, but one that is rather large. If you try it, you'll see that we'll never get six of those on the screen. To make the circle smaller we try

```
TO CIRCLE
  REPEAT 18Ø (FD 1 RT 2)
END
```

That's more like it. Next we need to fit a square around this circle. One disadvantage of this way of drawing circles is that we do not know the size (that is, the radius) of the resulting circle. We can find it by experiment. Simply run the **CIRCLE** procedure, then turn the turtle right 9Ø degrees and move the turtle forward until it crosses the circle. A first guess of 5Ø seems about right. When we try **FD 5Ø**, we find that we need a bit more, and that the diameter of the circle is about 56 units.

It appears that we should then enter a procedure to draw a square with sides of 56 units.

```
TO SQUARE
  REPEAT 4 (FD 56 RT 9Ø)
END
```



SQUARE

This will draw a square, but it will leave us with a problem. **SQUARE** starts with the turtle at a corner of the square. The corner is an awkward place to start drawing a square which is around a circle. This example shows that when procedures are to be used together, some attention must be devoted to making them fit or connect. We choose to make the two procedures connect by starting and ending the square at the center of a side, where the circle and the square touch.

```
TO SQUARE
  REPEAT 4 (FD 28 RT 90 FD 28)
END
```

Now verify that this set of procedures is a solution to the original problem by running **PATTERN**.

Let's analyze what we've just done. The road map for attacking the problem was to break the problem into a set of subproblems, and in turn to break each subproblem into even simpler subproblems until the subproblems can be solved by a single **REPEAT** statement. Specifically, we broke the original problem into the problem of drawing **SQUARE–CIRCLE** six times; we broke **SQUARE–CIRCLE** into the problems of drawing a square and drawing a circle. These last two problems were easily solved with a single **REPEAT** statement. In general, we follow this sequence in attacking a problem, although we do not insist that the lowest level procedure consist of a single **REPEAT** statement.

One of the reasons for using Super LOGO with children is that it is an excellent way to teach children a most powerful and useful general problem-solving approach. That approach is what we have just illustrated. Basically, it involves working from the overall view down to the details by breaking each problem into pieces. Moreover, there is no limit to the number of problems like the one given above that can be generated to give children practice in problem solving. By reviewing the children's solutions for style and clarity and by comparing their solutions with other solutions of the same problem, you can teach them that problems may have several equally good solutions but also that not all solutions are equally clear and understandable. In a teaching situation, give the students feedback on the style of their procedures as well as on the correctness of their procedures.

7. VARIABLES

"Variable" is the name used to describe unique storage locations where numbers, letters, words, or sentences can be kept. In this chapter we'll use variables only for keeping numbers. We specify the contents of a variable by typing a colon (:) followed by any number of letters and/or numbers. Variables can be used anywhere numbers can be used. By using variables in place of numbers, we can make our procedures useful in a wider variety of applications. For example, we can make our **SQUARE** procedure draw squares of many sizes.

```
TO SQUARE :SIDE
  REPEAT 4 (FD :SIDE RT 90)
END
```

If you came here directly from the last chapter, then there is another version of **SQUARE** in memory. To clear out the memory, press [SHIFT] [CLEAR] while you are in BREAK mode. Then get into the EDIT mode and enter the new version of **SQUARE**. Now, to run **SQUARE**, get into RUN mode and enter

```
SQUARE 40
```

Because we have listed the variable **:SIDE** on the **TO** statement, we must give a value when we call (or use) the procedure **SQUARE**. Now try a variety of other numbers, for instance

```
SQUARE 60
SQUARE 20
```

Notice that the computer takes the number which follows the procedure name **SQUARE**, sees that that number is the value of the variable **:SIDE**, and uses that number every place the variable **:SIDE** appears within the procedure.

What happens if we forget the number? Try

```
SQUARE
```

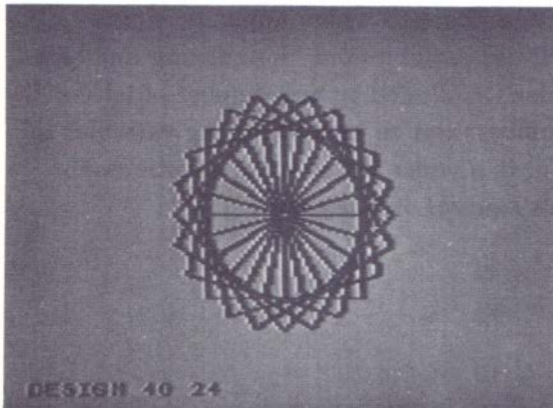
If we don't provide a number, then the computer provides a zero. The brief flicker is due to the turtle turning in place while drawing a square with zero length sides.

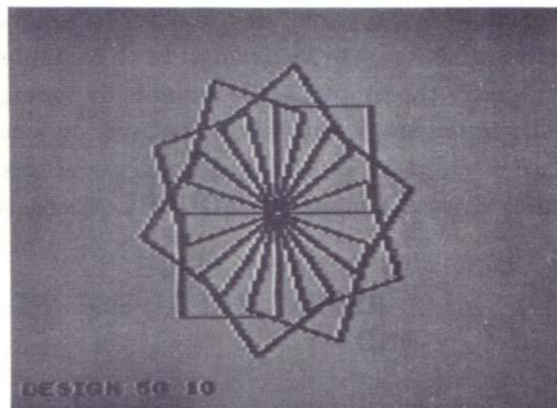
Variables can be used in other positions as well. Here's another example.

```
TO DESIGN :LENGTH :TIMES
  REPEAT :TIMES (SQUARE :LENGTH
                 RT 360/:TIMES)
END
```

Enter this and try running with a few different values of **:TIMES** and **:LENGTH**. For example

```
DESIGN 40 24
DRAW DESIGN 50 10
```





The computer keeps track of the variables by the order. Because the order in the **TO** statement for **DESIGN** is **:LENGTH :TIMES**, the command **DESIGN 4Ø 24** causes the value **4Ø** to be assigned to **:LENGTH** and the value **24** to be assigned to **:TIMES**.

Notice also that the name of the variable in the call of **SQUARE** (**SQUARE :LENGTH**) need not be the same as the name in the definition of **SQUARE** (**TO SQUARE :SIDE**). By the time the command **SQUARE** is reached within **DESIGN**, the variable name **:LENGTH** has a value (for example, **4Ø**). The value, not the variable name, is passed to **SQUARE** and then assigned by **SQUARE** to the variable **:SIDE**.

Variables listed on the **TO** statement are local to the procedure. Again we illustrate using the previous programs. Enter

```
TO DESIGN :LENGTH :N
  REPEAT :N (SQUARE :LENGTH
                    RT 36Ø/:N)
END

TO SQUARE :N
  REPEAT 4 (FD :N RT 9Ø)
END
```

Here the variable **:N** is used for two different quantities, one in the main procedure **DESIGN** and another in the subprocedure **SQUARE**. This causes no problems or confusion because the variables for the two procedures are kept completely separate in the memory. The variable **:N** in the main procedure refers to a different memory location than the variable **:N** in the subprocedure.

If we want a variable to be local to a procedure, we mention it in the **TO** statement which begins the procedure. We also can create global variables, variables which use a common memory location in all procedures in which they appear. Global variables are created when you use them in a procedure without including them in the **TO** statement. This provides one way to share information among procedures.

DESIGN contains our first example of arithmetic expressions, here **360/:N**. For a while we'll use only the standard four arithmetic operations: addition (+), subtraction (—), multiplication (*), and division (/). No parentheses are needed unless the order of operations is non-standard. Thus, in Super LOGO,

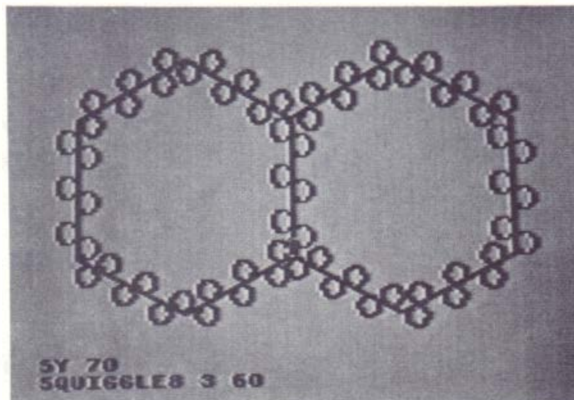
```
2 * 3 + 4 = 10
2 * (3 + 4) = 14
```

The following procedures give additional examples of the use of variables and arithmetic expressions.

```
TO SQUIGGLE
  FD 7
  REPEAT 8
    (FD 4 RT 45)
  FD 7
  REPEAT 8
    (FD 4 LT 45)
  FD 7
END

TO SQUIGGLE8 :SIDE :ANGLE
  REPEAT 360/:ANGLE
    (REPEAT :SIDE (SQUIGGLE)
      RT :ANGLE)
  REPEAT 360/:ANGLE
    (REPEAT :SIDE (SQUIGGLE)
      LT :ANGLE)
END
```

Notice the use of the nested **REPEAT** twice in **SQUIGGLE8**. If **:ANGLE** were **180** and **:SIDE** were **4**, then each of the pairs of nested **REPEAT**s will repeat 2*4 or 8 times. Try

```
SQUIGGLE8 1 20
SQUIGGLE8 3 60
SQUIGGLE8 4 90
```



8. COLORS

Turtle tracks can be colored, and they can change color. Your Tandy Color Computer offers two color "sets" (or "settings") in the high resolution screen on which turtles live. Up to now, you have been running in color set Ø. You can shift color sets by the **COLORSET** command.
Get into the RUN mode and enter

```
COLORSET 1
```

to change color set. Then enter

```
COLORSET Ø
```

to change back.

Within each color set there are four colors, numbered Ø, 1, 2, and 3. The normal drawing color is color Ø and the normal background color is color 3. Change the background color by entering

```
BACKGROUND 1
```

or abbreviate

```
BG 1
```

Change the pen (or drawing) color by entering

```
PENCOLOR 2
```

or abbreviate

```
PC 2
```

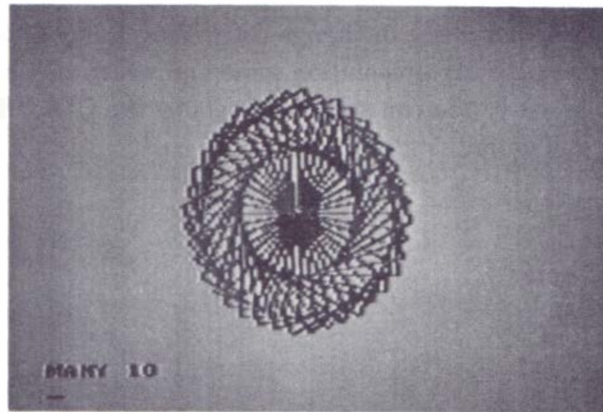
You can erase a portion of a drawing by making the pen color the same color as the background color and drawing over the unwanted part of the drawing.

Let's add color to some of our earlier procedures. One interesting choice is **FOUR**. Retype **BOX** (see p. 21), then enter the procedure **FOUR** as:

```
TO FOUR
  REPEAT 2 (PC 1 BOX
            PC 2 BOX)
END
```

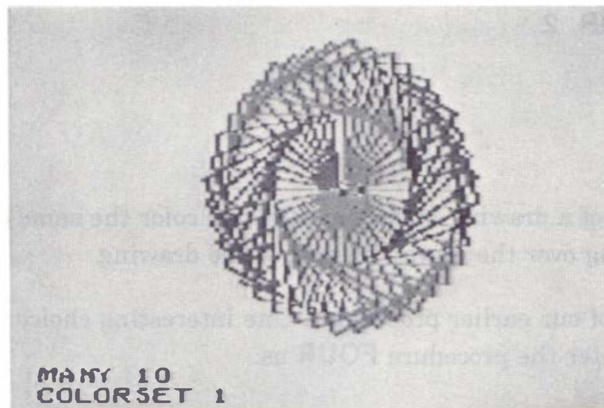
and to allow easy experimentation, make **MANY** into

```
TO MANY :N
  REPEAT :N (FOUR RT 9Ø/:N)
END
```



We would like to be able to name the colors you will get with specific pen colors and color sets, but colors vary from TV to TV; they vary with the color settings on the TV, and they may even switch when you restart your computer. Try running **MANY** (from page 33) with a value of **10**. Then adjust the color and tint controls on your TV set to your satisfaction. On many TV's, color set 1 will give more interesting colors, so be sure to try that too. You can change color sets without redrawing the figure by typing

```
COLORSET 1
```

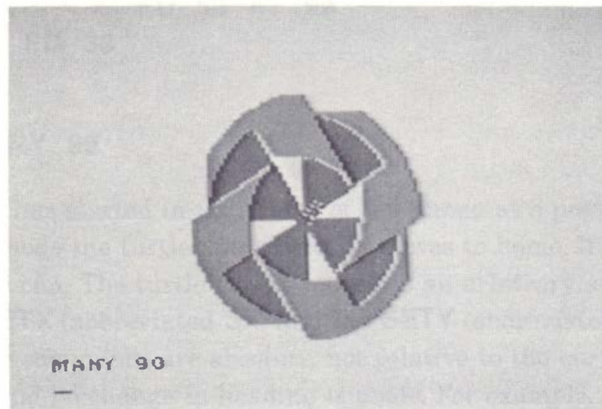


An interesting variation can be created by the following changes.

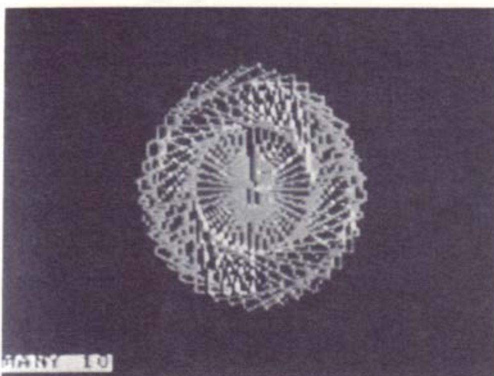
```
TO BOX
  PC 1 FD 50 RT 90
  PC 2 FD 30 RT 90
       FD 50 RT 90
  PC 1 FD 30
END
```

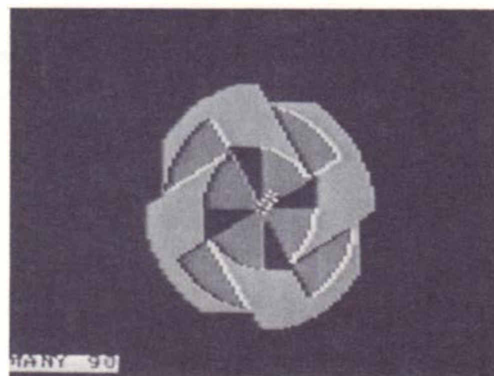
```
TO FOUR
  REPEAT 4 (BOX)
END

TO MANY :N
  REPEAT 2 * :N/3 (FOUR RT 90/:N)
END
```

Try it with **:N = 90**. If it is too slow, or if you feel sorry for any turtle that has to run around at that speed for so long, hide the turtle (**HT**) before calling **MANY**.



You might prefer the colors you get with a dark background. Try setting the background to 0, and rerun the preceding two examples **MANY 10** and **MANY 90**.





9. OTHER TURTLE COMMANDS

There are a few additional turtle commands which we have not yet used. We can raise and lower the turtle's tail, so we have the choice of leaving a track or not leaving a track. The commands are just what you'd guess: **PENUP** (abbreviated **PU**) and **PENDOWN** (abbreviated **PD**).

Let's illustrate by removing the lines of one color from the previous figure. Change **BOX** to

```
TO BOX
  PU FD 5Ø RT 9Ø
  PD PC 2  FD 3Ø RT 9Ø
           FD 5Ø RT 9Ø
  PU FD 3Ø
END
```

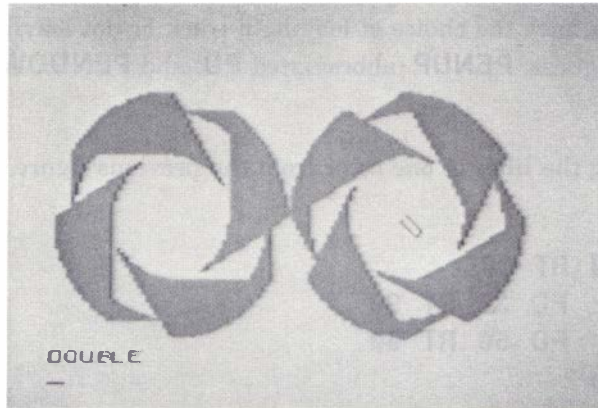
and again run **MANY 9Ø**.

Every figure so far has started in the center of the screen at a position called home. When we get into the RUN mode the turtle automatically moves to home. If we want to start the turtle somewhere else, we can. The turtle can be moved to an arbitrary and absolute screen position by means of the **SETX** (abbreviated **SX**) and the **SETY** (abbreviated **SY**) commands. The results of these two commands are absolute, not relative to the current position of the turtle. No line is drawn, and no change in heading is made. For example,

```
TO DOUBLE
  SETX 6Ø
  MANY 9Ø
  SETX 18Ø
  MANY 9Ø
END
```

The heading of the turtle also can be set to an absolute value independent of its current heading. The command is **SETHEADING** (abbreviated **SH** or **SETH**). The heading can be anything between Ø and 359 degrees. Zero degrees is straight up. Try

```
TO DOUBLE
  HT
  SX 6Ø
  MANY 9Ø
  SX 18Ø SH Ø
  MANY 9Ø
END
```



The remaining turtle instruction is **HOME**. **HOME** returns the turtle to the home position (the center of the screen) with a heading of Ø degrees (straight up). The turtle is also made visible.

Procedures which draw circles and parts of circles (arcs) are very useful in other projects. There are some drawbacks to the **CIRCLE** procedure given in Chapter 5, page 27. It's hard to predict the size of the circle from the size of the step, it's hard to find the optimum number of steps for the best circle, and it's hard to figure where the circle is centered. The following procedure is a useful alternative.

```
TO ARC :X :Y :RADIUS :DEGREE
  PU SX :X SY :Y
  REPEAT :DEGREE (FD :RADIUS
    DOT BK :RADIUS RT 1)
END
```

We've used a new turtle command, **DOT**. As you would guess, **DOT** tells the turtle to make a dot on the screen.

For a circle, try

```
ARC 15Ø 1ØØ 2Ø 36Ø
ARC 12Ø 12Ø 4Ø 36Ø
```

For an arc, try

```
ARC 6Ø 6Ø 1ØØ 1ØØ
```

For a partial circle, try

```
ARC 16Ø 9Ø 6Ø 24Ø
```

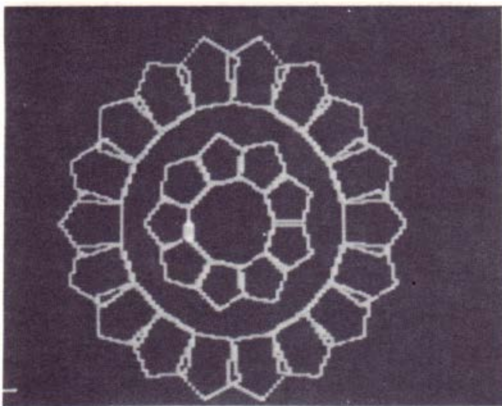
ARC is slower than **CIRCLE**, but in some situations it is more convenient. With some values of **:RADIUS** you might be able to get the same accuracy with fewer steps (for example, **REPEAT 180** and **RT 2**), but then the number of degrees will have to be divided to get the right number on the **REPEAT**.

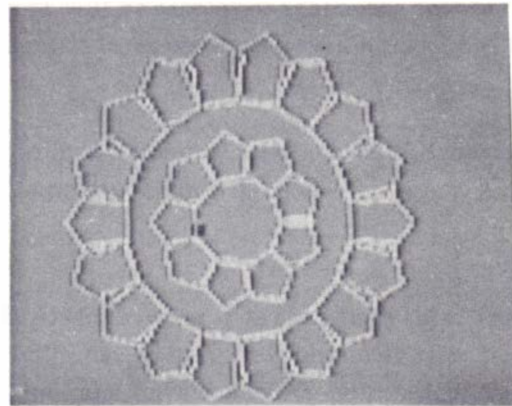
The following program again makes use of the **SX** and **SY**, here to get the right relative spacing of independent parts. The correct numbers for the two are found by trial and error. Try the procedures with a variety of pen colors and backgrounds. Remember to remove the split screen by entering

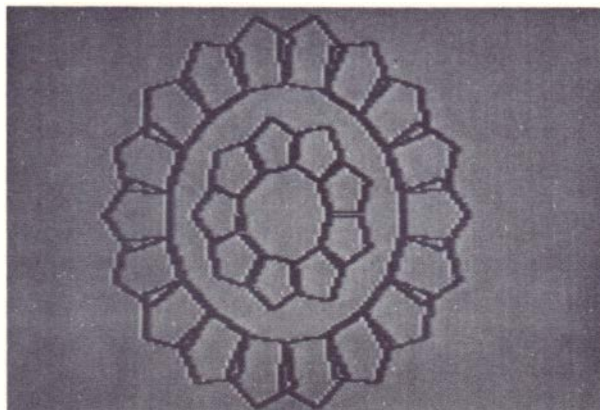
```
DRAW KIRSTIN
```

in RUN mode.

```
TO KIRSTIN
  CLEAR SX 60 SY 80
  REPEAT 18 (PENT 20 RT 20)
  SX 95 SY 82
  REPEAT 9 (PENT 15 RT 40)
END

TO PENT :SIDE
  REPEAT 5 (FD :SIDE LT 72)
  FD :SIDE
END
```







10. SAVING, LOADING, AND PRINTING YOUR SUPER LOGO PROCEDURES

Although Super LOGO procedures can do an amazing amount with very little code, we still don't want to have to retype procedures every time we start up. To avoid having to do this, you can store procedures on cassette tape. This chapter describes how to do so.

To move procedures to and from cassette, we must be in BREAK mode. By now, you probably have a number of procedures in memory that you would like to keep. Get into EDIT mode and delete any that you do not want to save. The delete-line operation will speed the deletion of unwanted procedures. [SHIFT] [CLEAR] (that is, holding down the [SHIFT] key and pressing the [CLEAR] key) deletes from the current cursor position to the end of the line. To delete a whole line, position the cursor at the start of the line and press [SHIFT] [CLEAR]. Next get into BREAK mode (use the [BREAK] key), and press [S]. At this point, the prompt will be

LOGO: SAVE:_

You now have to tell the computer where to save the procedures that are in memory. Of course, the tape recorder must be plugged in as described in the Operation Manual for the Tandy Color Computer. Make sure that the volume control is set close to 5. Rewind the tape (REWIND, STOP). Next press the RECORD and PLAY buttons so that they both stay pressed down. If you are not using leaderless tape, pull out the MIC plug for about 5 seconds. (This will make sure that you begin recording on blank tape.) Now you are ready to record the procedures. Simply respond [T] [ENTER] to the BREAK mode prompt:

LOGO: SAVE:_

When the recording is done, the BREAK mode prompt will be displayed again. If a number and a question mark appear after the [T], then the procedures were not saved properly, so try again.

Loading programs from cassette is also simple. Again it is necessary to be in BREAK mode. In response to the BREAK mode prompt, press [L]. The prompt then will read

LOGO: LOAD:_

The response here is exactly the same as for **SAVE:**. Use the letter **T** to load from tape. Then press [ENTER] to start the process. Of course, you will have to have the volume on the cassette recorder set to about 5, have the tape rewound, and have the PLAY button depressed before pressing [ENTER].

At times you will want to carry out some more elaborate transfers between the computer memory and tape. For example, you might want to add some procedures already on tape to the procedures in memory. This would be the case if you had created a module of procedures for drawing circles and polygons which you planned to use in many projects. A module of procedures can be added to whatever is already in memory by use of the **MERGE** operation. Simply respond to the **LOGO:** prompt with [M]. At this point the prompt will be

LOGO: MERGE:_

To start the process, enter **T**.

We just learned how to combine sets of procedures; how do we save only a portion of what is in memory? The computer recognizes special start and end markers. If these markers are present in a set of procedures, then a **SAVE** operation saves only the lines between the two markers. Therefore, to save only a portion of the procedures in memory you must first enter EDIT mode to insert the markers. The marker for the start is >>, and the marker for the end is <<. Once these two are in place, do a regular **SAVE** ([BREAK], [S], and [T],). If you want to run the procedures still in memory, you first should return to EDIT mode to remove the markers.

Note that with the **MERGE** and the partial **SAVE** operations you can build new modules which contain any combination of procedures selected from other modules, without retyping any of the procedures.

If you have a printer for your Color Computer, you can print all the procedures in memory. Again it is all or nothing, except that you can interrupt the printing by pressing [BREAK] without damaging or losing the programs in memory. To print, connect the printer as described in the Owner's Manual; load the paper and turn on the printer.

You may have to reset the baud rate to get the printer to work properly. The baud rate is the rate at which the computer sends characters to the printer; the computer must transmit at the rate the printer expects. If the baud rate is wrong, the printer will print but it will be gibberish. To reset the baud rate, get into RUN mode and enter

BAUD *number*

"*number*" should be replaced by one of the numbers in the right-hand column:

for baud rate:	use number:
300	180
600	87
1200	41
2400	18

Consult the manual for your printer to find what rate it wants. Once the rate is set, it remains unchanged until reset by another **BAUD** command or until the computer is turned off. When the computer is turned on, the baud rate is automatically set to 600.

Next, from BREAK mode, enter

P for single space

or **Q** for double space

and the contents of memory will be printed. If for some reason you want to eliminate the line feed at the end of any line (thus using a larger portion of the paper width) enter EDIT mode and insert an @ character at the end of every line for which you want to eliminate the carriage return and line feed. (To place an @ character in a line in EDIT mode, you'll need to press the [@] key twice.)

There remains the question of saving results, the pictures and/or text on the screen. One way is to take pictures (this is the only way to get color if you don't have a Radio Shack Color Printer). To avoid false patterns due to interactions of the camera shutter with the video display we recommend a shutter speed of ½ second. Use a tripod and a cable release for the camera. The lens setting is somewhat dependent on the brightness setting of the TV, and of course on the film speed. A good starting point is to set medium brightness on the TV and use a lens opening of about f8 with film speed of 100 ASA. The reds are likely to come out rather brownish, and commercial developers are likely to overexpose prints with large dark backgrounds. However, the illustrations in this manual are typical of what can be done without much trouble. You will minimize distortion if you use a telephoto lens.

The **PRINTSCREEN** command tells the computer to make a paper copy of whatever is on the screen. The command can be abbreviated as PS. The command must be followed by a number (or expression) with a value between 1 and 4. These numbers tell the computer what printer is in use. Pick from the following table:

1 - RS DMP 110 single width; Line Printer 7
2 - RS DMP 110 double width
3 - RS Color Printer
colors: 0 - magenta
1 - yellow
2 - violet
3 - white
4 - RS Color Printer
colors: 0 - blue
1 - red
2 - green
3 - white

As with all uses of printers, the baud rate must be set correctly. See the instructions above.

The **PRINTSCREEN** command makes a copy of the screen dot by dot; it does not print letters as units. It can take several minutes to print out the whole screen. However, you can interrupt the process by holding down the **BREAK** key. One caution: characters printed with the Color Printer are not printed clearly; they have colored ghosts.

In text or list processing operations you may want the results printed at the same time they appear on the screen. The **ECHO** command causes all characters displayed on the screen by **PRINT**, **TEXT**, and **REQUEST** commands to also be printed. If no printer is connected, then the characters appear only on the screen. **ECHO** can be turned off with the **NOECHO** command.

11. RECURSION

In the Super LOGO language, any procedure can call any procedure. When the procedure calls itself, we have a very powerful logical structure called recursion. One clever example of recursion was given by Hofstader in his book Godel, Escher, Bach.

> Hofstader's Law: It always takes longer than you expect, even when you take into account Hofstader's Law.

There are actually two types of recursion. We'll start with the easier one: recursion where the call is the last statement of the procedure. As usual, it is easiest to look at examples. Recursion can be used in place of the **REPEAT** statement.

```
TO CIRCLE
  FD 1 LT 2
  CIRCLE
END
```

When we run **CIRCLE**, the turtle moves forward one step and turns. Then **CIRCLE** is called, which causes the turtle to move forward one step and turn, etc. In principle this process could continue forever. However, every time a procedure is called some memory is used up. Eventually the memory is all used up, and we get the message

```
MY MEMORY IS TOO FULL
```

Try it.

So although recursion can be used instead of **REPEAT** in some procedures, there are some disadvantages to doing this. We have to find some way of stopping the computer, or it will run out of memory. There are also some great advantages to using this type of recursion. The following program appears in all LOGO books and manuals.

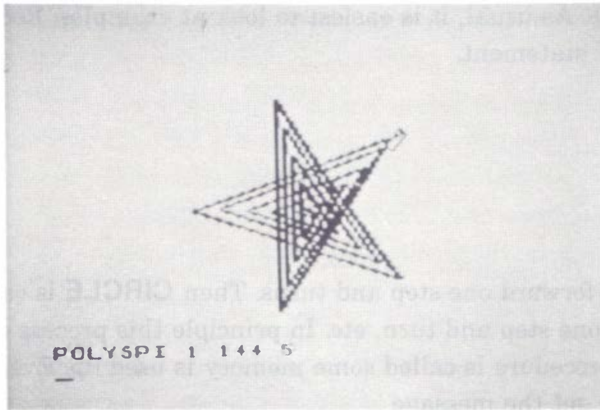
```
TO POLYSPI :SIZE :ANGLE :STEP
  FD :SIZE
  RT :ANGLE
  POLYSPI (:SIZE + :STEP) :ANGLE
          :STEP
END
```

This procedure is so much fun to play with that we think you should do so before we get involved in any explanations. One suggestion before you start: the figures created are likely to outgrow the screen long before the memory runs out. The wrap-around feature of the screen will then lead to some striking but puzzling effects. To start with, let's prevent wrap-around. Enter RUN mode and type

```
NOWRAP
```

Then try a variety of runs, for example

```
DRAW CLEAR POLYSPI 1 90 1
DRAW CLEAR POLYSPI 1 90 5
DRAW CLEAR POLYSPI 1 120 3
DRAW CLEAR POLYSPI 1 122 3
DRAW CLEAR POLYSPI 1 144 5
DRAW CLEAR POLYSPI 1 145 1
DRAW CLEAR POLYSPI 1 176 3
```



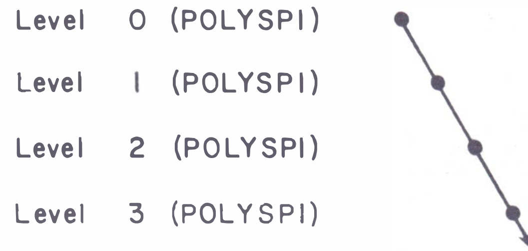
If you then want to see what happens when the computer allows wrap-around, type

```
WRAP
```

and try some more runs.

Now let's try to figure out what is going on with this **POLYSPI**. It is useful to think of a Super LOGO program in terms of levels: the main program is a procedure at level 0, a subprocedure called from level 0 is at level 1, a subprocedure called from level 1 is at level 2, etc. The operation of a program like **MANY** can be diagrammed as follows:

The transitions down and up between levels Ø and 1 are controlled by the **REPEAT** statement in **MANY** (down to level 1) and the **END** statement in **FOUR** (up to level Ø). The transitions down and up between levels 1 and 2 are controlled by the **REPEAT 4** statement in **FOUR** (down to level 2 four times) and the **END** statement in **BOX** (up to level 1). In a program like **POLYSPI** the path is actually less complex.



The transitions down are controlled by the statement

```
POLYSPI (:SIZE + :STEP) :ANGLE :STEP
```

There are no transitions back up because the procedure never reaches the **END** statement. Thus the computer sinks down level after level until it finally runs out of memory. **POLYSPI** gives interesting figures because of the difference in the values of the variables at the different levels. We begin the program with the command

```
POLYSPI 1 9Ø 5          (Level Ø)
```

The recursive call of the procedure is (when we substitute the current values of the variables)

```
POLYSPI 6 9Ø 5          (Level 1)
```

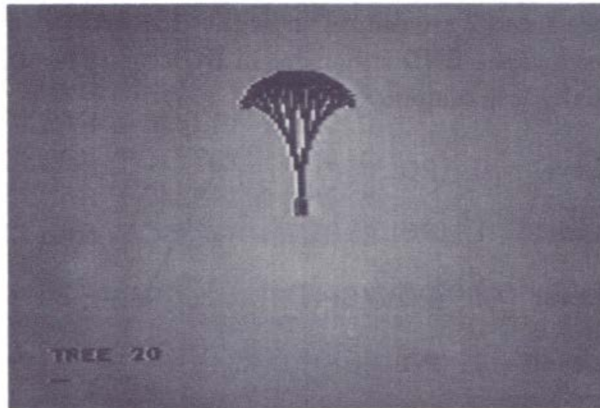
Therefore at level Ø the **FD** command is for length 1 and at level 1 the **FD** command is for length 6. The pattern continues

```
POLYSPI 11 9Ø 5         (Level 2)
POLYSPI 16 9Ø 5         (Level 3)
```

until the turtle runs off the screen or the computer runs out of memory.

We now turn to the more complex type of recursion, recursion where returns to the higher levels are actually made. Another popular program in turtle geometry is called **TREE**.

```
    TO TREE :N
1     IF :N<2 (STOP)
2     FD :N
3     RT 15
4     TREE (3*:N/4)
5     LT 3Ø
6     TREE (3*:N/4)
7     RT 15
8     BK :N
9   END
```



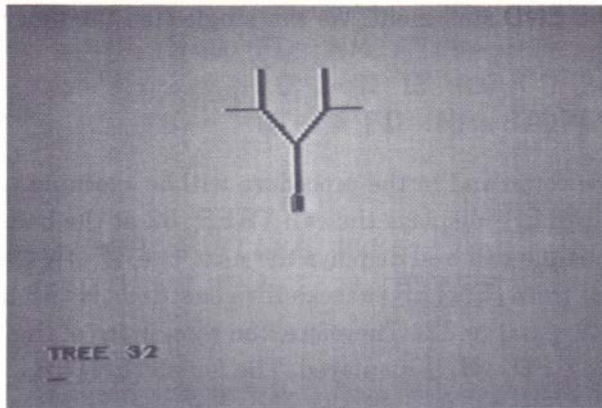
The line numbers are not part of the procedure; they are there for reference in our discussion of the procedure. Enter this procedure (without the line numbers, of course) and try running it with a value of 2Ø to 3Ø for **:N**. You might try making the numerical factors in the two calls of **TREE** (lines 4 and 6 of the procedure) somewhat different, thus producing an asymmetrical tree. You also could try changing the angles, but notice that the sum of the two right turns is equal to the left turn.

Now let's try to understand the program. We have introduced two new ideas in line 1. The first is the conditional **IF**. The **IF** must be followed by an expression which has a truth value. In **TREE** the expression is

:N < 2

If the current value of **:N** is less than 2, then this expression is true and the rest of the statement will be executed. If the current value of **:N** is 2 or greater, then this expression is false and the rest of the statement will be skipped. The rest of the statement is placed in parentheses; it may consist of many commands, and it may extend over many lines. The second new item in line 1 is the control statement **STOP**. The **STOP** statement ends a procedure. **STOP** has the same effect as an **END** statement, but an **END** statement can appear only at the end of a procedure.

It will be easier to understand **TREE** if we make some changes to simplify it. Change the condition in line 1 to **:N<18**, and change the right turns (lines 3 and 7) to **45** and the left turn (line 5) to **9Ø**. Then run **TREE 32** to get a simpler tree.



The following diagram and table outline the operation of TREE.

STEP	:N	START	END	LINES
a	32	Ø	R45	1-4
b	24	R45	R9Ø	1-4
c	18	R9Ø	R135	1-4
d	13	R135	R135	1
e	18	R135	R45	5,6
f	13	R45	R45	1
g	18	R45	R9Ø	7-9
h	24	R9Ø	Ø	5,6
i	18	Ø	R45	1,4
j	13	R45	R45	1
k	18	R45	L45	5,6
1	13	L45	L45	1
m	18	L45	Ø	7-9
n	24	Ø	R45	7-9
o	32	R45	L45	5,6
p	24	L45	Ø	1-4

etc.

First look at the diagram. The program tries to reach the lowest level of the procedure until forced to rise in level by the **STOP** or the **END** statement. We can single-step the program by running it as follows. In **RUN** mode type

TRACE TREE 32

Now every time you press **ENTER**, one command in the procedure will be executed. For example, when you start, the first **ENTER** displays the call **TREE 32** at the bottom of the screen. The second **ENTER** executes the call and displays the next line, **IF :N<18 (STOP)** at the bottom of the screen. The third **ENTER** checks the condition **:N<18** and finds it false (remember we start with **:N** equal to **32**). Therefore, the remainder of the line is to be skipped (the **STOP**), so the next line, **FD :N**, is displayed. The fourth **ENTER** executes **FD :N** (which you can see), and displays **RT 45**; the fifth **ENTER** executes **RT 45** (again visible), and the sixth **ENTER** executes the recursive call **TREE (3*:N/4)**. You can follow the table exactly, as long as you realize that the statement

IF :N<18 (STOP)

is one statement if the condition is false, but two statements if the condition is true (execute the **STOP)**. A step in the table is counted as all the lines executed from when a level is entered until the level is exited. A level can be exited in three ways: by a **STOP** or an **END**, which completes the procedure at that level and goes up; or by a call of another procedure, which leaves the current procedure incomplete and goes down.

Instead of giving an elaborate account in words of what is happening, we recommend that you run the program using **TRACE** and follow the table and the program in parallel. If you get confused, start over and try again. Recursion is a little complex, but it is so powerful that it is worth the effort to understand.

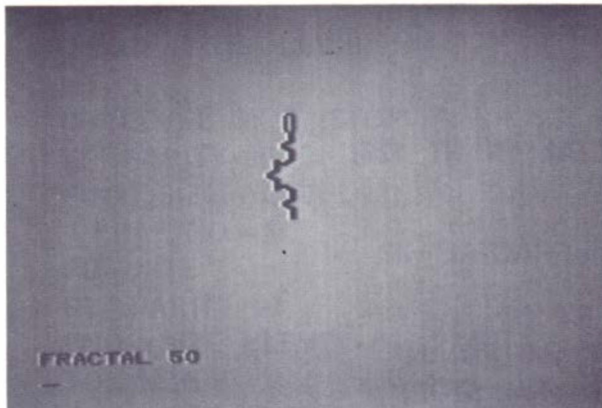
Incidentally, the recursive call **TREE (3*:N/4)** could be written **TREE (Ø.75*:N)** just as well. The first way is what is used with a system that only has integer arithmetic; the second can be used with Super LOGO because it has two-decimal arithmetic. Two cautions if you use the second: the zero before the decimal is essential, and computer arithmetic is never very accurate in the last decimal (no problem in this example). You might think that the second way would be faster (only one multiplication instead of a multiplication and a division), but it isn't. Most of the time is taken up in the bookkeeping for the recursion and for drawing the turtle many times.

The next program draws a figure which is called a fractal. A fractal is a figure which looks the same no matter what magnification is used to view it (of course we are limited by the screen resolution here). In this example we'll start with the basic shape:

The idea is that each of these four lines should be made up of that same shape. At one additional level of detail that gives:

Each of the lines in this drawing is in turn made up of four lines with the basic shape, etc. This is a place to use recursion.

```
TO FRACTAL :N
  IF :N<15 (FD :N STOP)
  FRACTAL (:N/3)
  LT 60
  FRACTAL (:N/3)
  RT 120
  FRACTAL (:N/3)
  LT 60
  FRACTAL (:N/3)
END
```



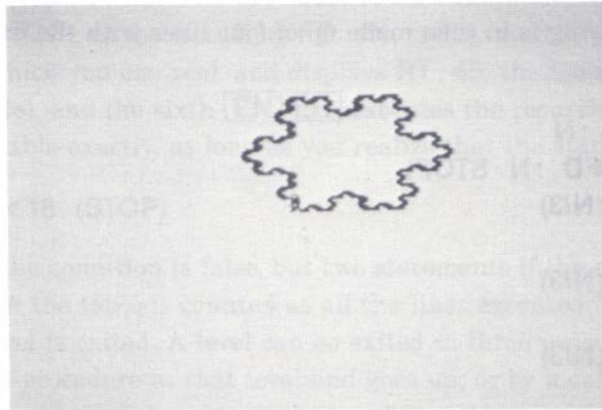
Running **FRACTAL 50** will show the pattern. (You may want to enter

```
DRAW RT 90 SX 0 FRACTAL 50
```

to turn it the way we've shown it in the manual.) Notice that the pattern divides the whole length into thirds; that is why we divide by 3 on the recursive calls. However, the resulting length will change somewhat depending on the conditional **IF** statement because of the round-off loss in arithmetic. You'll see that if you change the conditional to something finer such as **:N<4**. That pattern looks like the edge of a snowflake. Why not make it into something six-sided?

```
TO FLAKE :N
  CLEAR
  REPEAT 6 (FRACTAL :N RT 6Ø)
END
```

You may have to play with the starting position (**SX** and **SY**) and the size to get a nice figure without wrap-around. You may also prefer the figure you get when **FLAKE** is made to draw three sides at 12Ø degrees.



Other variations are possible. We can replace **FD :N** in the **IF** statement with a more elaborate series of commands. A few examples follow.

```
TO FLAKE :N
  CLEAR
  SX 5Ø
  SY 4Ø
  REPEAT 3 (FRACTAL :N RT 12Ø)
END
```

Replace the conditional statement in **FRACTAL** with

```
IF :N<9 (FD :N/4 RT 8Ø
         FD :N LT 16Ø FD :N
         RT 8Ø FD :N/4 STOP)
```

and try

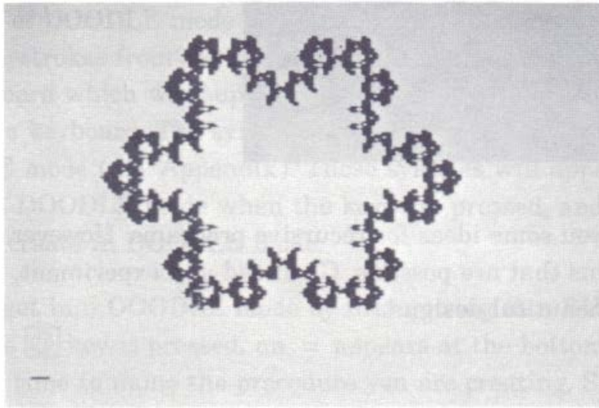
```
DRAW FLAKE 15Ø
```

Another variation is

```
IF :N<9 (FD :N/4 RT 8Ø
        FD 2*:N LT 16Ø FD 2*:N
        RT 8Ø FD :N/4 STOP)
```

and try

```
DRAW FLAKE 15Ø
DRAW FLAKE 7Ø
```

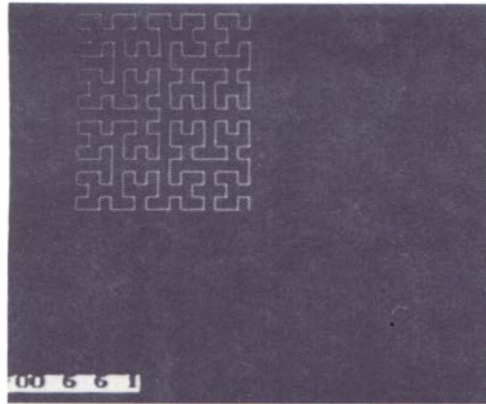


Recursion can be used to draw endless space-filling patterns. The following example is typical.

```
TO FOO :SIZE :LEVEL :PARITY
  HT
  IF :LEVEL = Ø (STOP)
  LT :PARITY*9Ø
  FOO :SIZE (:LEVEL–1)
    (:PARITY* –1)
  FD :SIZE
  RT :PARITY*9Ø
  FOO :SIZE (:LEVEL–1) :PARITY
  FD :SIZE
  FOO :SIZE (:LEVEL–1) :PARITY
  RT :PARITY*9Ø
  FD :SIZE
  FOO :SIZE (:LEVEL–1)
    (:PARITY* –1)
  LT :PARITY*9Ø
END
```

An appropriate set of numbers is

DRAW FOO 6 6 1



In this chapter we have tried to give you some ideas for recursive programs. However, we have just scratched the surface of the designs that are possible. Go ahead and experiment, and let others know if you discover any new, beautiful designs.

12. DOODLE MODE — PROCEDURES WITHOUT TYPING

Super LOGO provides a way to enter graphic procedures into the computer without typing the turtle commands like **FORWARD** and **RIGHT**. The reason for including this feature is to provide a way for children who are not ready to read or to type to use the language and to benefit from the practice in structured thinking that the language offers. The features of **DOODLE** mode are arranged with that audience in mind. Obviously the children are not going to be able to read the manual, so a parent or teacher will have to assist them in learning. In this and the next chapter we will teach the assistant the mechanics of two ways to use Super LOGO with children; actual suggestions for activities with the children are gathered together in Chapter 14. In this chapter, we will cover the mechanics of DOODLE mode.

The idea of DOODLE mode is that a minimum set of turtle commands can be entered by single keystrokes from the keyboard. Before proceeding, you should get the plastic overlay for the keyboard which was supplied with the Super LOGO package. This overlay fits over the top row of the keyboard. The symbols on the overlay show the meanings of the numeric keys in DOODLE mode (see Appendix). These symbols will appear at the bottom of the screen in DOODLE mode when the keys are pressed, and they will appear in the procedures that you create in DOODLE mode.

You can get into DOODLE mode by first getting into RUN mode, then pressing the [@] key. When the [@] key is pressed, an = appears at the bottom of the screen. This is the indication that it is time to name the procedure you are creating. Simply type the name you want to use; the name can be as simple as a single letter or number. After typing the name, press [ENTER]. The computer is now in DOODLE mode and the top row of keys has its new meaning.

The meanings of the keys are

[1]	**CLEAR**	[2]	**HOME**	[3]	**PENUP**	[4]	**PENDOWN**
[5]	**RT 45**	[6]	**LT 45**	[7]	**FD 1**	[8]	**FD 10**
[9]	**RT 15**	[0]	**LT 15**				

Try each of the keys in turn. Of course [1] (**CLEAR**) will clear the screen, so you won't have much time to see that one. Note the correspondence between the symbols on the keys, the symbols on the bottom of the screen, and the action of the turtle.

Now that you have a bit of the idea of DOODLE mode, let's try to make something useful. To get a fresh start, exit DOODLE mode by pressing [BREAK]. This puts you in BREAK mode. Clear the procedure space by pressing [SHIFT] [CLEAR]. Get into the RUN mode (press [R]) and then get into the DOODLE mode (press [@]). The reason that we always get into the DOODLE mode from RUN mode is that we may want to draw patterns on the screen in RUN mode for the child to interact with or copy in DOODLE mode. Notice that the screen is not cleared when we enter DOODLE mode. Now let's name the procedure we are creating "**S**" by typing an [S] and pressing [ENTER].

Begin by drawing a box using the top row of keys. When the box is completed, exit DOODLE mode by pressing the **BREAK** key. Then get into the RUN mode and run the procedure **S**. The only difference between this and the procedure we earlier called **BOX** is that **S** is a little slower. To see that **S** actually exists as a procedure, get into EDIT mode and look at procedure **S**. Notice that it is there with exactly the symbols used in its definition.

We can actually edit **S** in EDIT mode just as we edit any other procedure. For example, we can add a diagonal to a box in several ways. First, we can add the type of turtle commands we are already familiar with. For example, for a box with sides of length 4Ø, we might use an **RT 45** or **RT 135** followed by an **FD 6Ø**.

This shows that DOODLE mode turtle commands can be mixed with regular turtle commands. However, in some sense, this way of editing defeats the purpose of DOODLE mode because the child is not likely to be able to understand the change. To keep it understandable for the child, we edit using the DOODLE mode symbols in EDIT mode. Each DOODLE mode symbol can be made by pressing **@** followed by the appropriate key. Thus we could insert the above instructions for a diagonal by the series of keystrokes

@ 5 @ 8 @ 8 @ 8 @ 8 @ 8 @ 8

That is, an **RT 45** (**@5**) followed by six **FD 10**'s (**@8**). Try it.

Obviously this kind of editing would be most useful in a cooperative project where the child was using DOODLE mode and the helper was using EDIT mode. A more likely type of error correction or editing is to make changes during the doodling process. For example, "I should not have taken that last step forward," or "I should not have turned that far." To see how to handle this, get into DOODLE mode (**BREAK**, **R**, **@**) and enter a new name, say **B**. Now doodle out a box, but go one step too far on the last side. Left-arrow (backspace) will now erase the last step in the procedure. Note that it does not erase the symbol for that step from the list at the bottom of the screen. Try eliminating step after step by repeated use of the left-arrow key. The edited version of the procedure is stored in the memory in correct form and can be seen in EDIT mode.

Another type of editing the child may wish to do is to add on to the end of a previous procedure. There is no simple way to do exactly that, but it is easy to produce the same effect. Get into RUN mode and run the current version of the procedure. That will draw the shape on the screen. Get into DOODLE mode and give a second name. Notice that the turtle is at the home position instead of at the end of the shape. Start the new procedure with **HOME** (key 2), raise the pen (key 3), move to the end of the shape, and lower the pen (key 4). Now you are ready to proceed with completing the shape. To get the whole shape while running, either run the two procedures in sequence or, in EDIT mode, remove the **END** statement from the first procedure and the **TO** *name* statement from the second procedure. If you do the latter, you can remove all the turtle commands from **HOME** to **PENDOWN** at the start of the second procedure, as well.

Thus far we have limited ourselves to horizontal, vertical, and 45-degree lines. What about other angles? Imagine a grid of squares with the turtle in a central one. If the turtle is going to move one square forward, there are only eight adjacent squares to move into. Thus, with a step forward of one unit the only angles which give consistent visible effects are 45 degrees

apart (Ø, 45, 9Ø, . . .). In DOODLE mode, the only steps forward possible are one unit and ten units. We've already said that the only turns that are useful with one unit forward are turns which are multiples of 45 degrees. With ten units forward, turns as small as 5 degrees give consistent visible results; but these seem too detailed (and require too many keystrokes) for the typical DOODLE mode user. Therefore the smallest turns provided are multiples of 15 degrees.

If you want to make a line at some other angle in DOODLE mode, you can. It just requires more keystrokes. The combination of **FD 1** operations and either no turn or a turn of 45 degrees will allow the drawing of a line at any angle. For example, a very small angle could be drawn by repeating this series the desired number of times: **FD 1Ø** (key 8) followed by nine **FD 1**'s (key 7), an **RT 45** (key 5), an **FD 1**, and an **LT 45** (key 6). This sounds awkward, but remember that it will be necessary very seldom. Most young children will find sufficient accuracy with turns of 15 and 45 degrees.

DOODLE mode characters are not available on printers. If you try to print a procedure created in DOODLE mode, the special characters will appear as lowercase letters. Thus it is impractical to print DOODLE mode procedures. Also notice that the @ character has a special meaning in EDIT mode. If you want to include an @ character in a line of a procedure to prevent a line feed when printing, you must press @ twice.

13. ONE-KEY DOODLING

The idea of DOODLE mode can be extended to an open-ended set of single keystroke operations if we give up the ability to store and edit the child's input as a procedure. This requires a set of procedures which we hereafter refer to as the OK Set (One Key Set). To start we define a set of procedures with single-character names. To call forth the desired action, the child presses the single key and then presses ENTER. This is easier shown than described.

The first step is to define a set of procedures which match the individual keys in DOODLE mode. Because we are not going to save or edit the procedures, we do not bother to draw the special symbols at the bottom of the screen (although if there was some reason to have them we could draw them with turtle commands). Clear the memory and enter the following procedures.

```
TO 1
  CLEAR
END

TO 2
  HOME
END

TO 3
  PU
END

TO 4
  PD
END

TO 5
  RT 45
END

TO 6
  LT 45
END

TO 7
  FD 1
END

TO 8
  FD 1Ø
END

TO 9
  RT 15
END

TO Ø
  LT 15
END
```

This set of procedures will allow the child to move the turtle freely around the screen in RUN mode using the keys he or she already knows from DOODLE mode.

The advantages of this approach become evident when we add to the list of procedures. The following procedures are typical.

```
TO T
  SH Ø HT PD FD 8
  RT 15Ø FD 15
  TRI 15
  SH 3Ø5 FD 8 SH Ø PU ST
END

TO TRI :SIDE
  IF :SIDE<2 (STOP)
  REPEAT 3 (RT 12Ø FD :SIDE)
  TRI (:SIDE-2)
END
```

Note: The turtle pen is lowered at the beginning of each of these procedures and raised at the end of each to prevent a trail being drawn while the child is positioning the turtle to draw another shape. **PD** and **PU** can optionally be removed from these procedures.

T will draw a triangle. You might wonder why **T** is so elaborate; after all, we could use the following to draw a triangle.

```
TO QUICKT
  REPEAT 3 (FD 15 RT 120)
END
```

The problem here is that the orientation of a triangle drawn by **QUICKT** will depend on the prior heading of the turtle. For the applications we have in mind we want all the triangles to have one vertex pointing up (**SH 0**). We also want to color in the triangle and we want to draw the triangle around the turtle's starting position. Therefore we use the procedure **TRI** and we move forward eight units before starting the triangle; hiding the turtle gains speed. **TRI** uses recursion to make a filled-in triangle. To get complete filling-in we must start at the correct vertex of the triangle (this is not obvious, but is a consequence of the way in which the Color Computer produces color in high resolution). Thus the line **RT 150 FD 15** in **T** moves us to a different vertex. You might try replacing this line with **RT 30** to get a striped triangle, and then make **TRI** read **REPEAT 4 (RT 129 FD :SIDE)** to get an even more interesting pattern. The commands in **T** after **TRI 15** return the turtle to the starting position with a heading of 0 degrees.

A similar set of procedures can be used to define a box and a circle.

```
TO B
  SH 45 HT PD FD 10 RT 45
  BOX 14
  RT 135 FD 10 SH 0 PU ST
END

TO BOX :SIDE
  IF :SIDE<2 (STOP)
  REPEAT 4 (RT 90 FD :SIDE)
  BOX (:SIDE-1)
END

TO C
  SH 0 HT PD
  REPEAT 60 (FD 8 BK 8 RT 6)
  PU ST
END
```

All these procedures restore the turtle to its starting position. It is not always easy to compute what moves are necessary to reach the original position. In **T** we did it by experiment. Run **T** immediately upon entering RUN mode, so that the starting position for the turtle is at the home (128,96) position. When the procedure has finished, enter the line

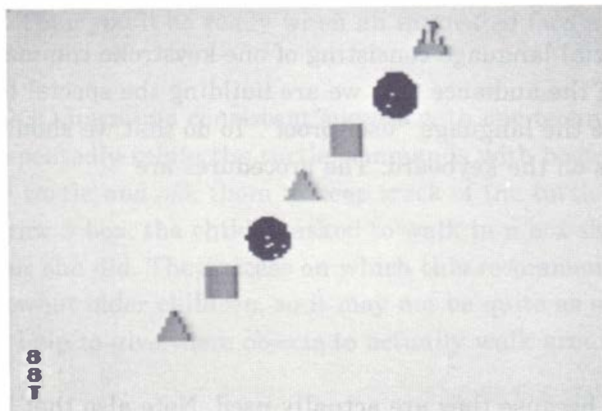
```
PRINT XCOR
```

This tells the computer to print out the x coordinate of the turtle position. Since we started at home we want to end at home, which has an x coordinate of 128. Notice that the printing occurs at the position of the turtle without moving the turtle. Therefore we can follow this with the command

```
PRINT YCOR
```

to see that we've returned to the correct vertical position as well. Of course the procedure **T** is correct; we showed you this in case you wanted to define procedures for other shapes as well. From these values we could tell what changes would have to be made in the last line of **T** to restore the turtle to the original (home) position.

With these procedures the child can move the turtle around the screen using the number keys, and the child can produce triangles by entering **T**, boxes by entering **B**, and circles by entering **C**. But, we hear you say, this is for children who don't know the letters. We suggest that you cover the selected keys with small adhesive labels on which the symbols have been drawn. In this example this would mean putting a label with a triangle on the **T** key, a label with a box on the **B** key, and a label with a circle on the **C** key. Of course you could use any other keys instead by renaming the procedures.



As in DOODLE mode, we want to have some way to erase mistakes. The way to do this is to redraw the shape with the pencolor set to the background color. We also have to pick a way for the child to control the erase. One possibility is to use double presses of the same key to specify erase. With a minor name change, then, we have the procedures

```
TO T
  PC 1
  T1
END
```

```
TO TT
  PC 3
  T1
END
```

```
TO T1
  SH Ø HT PD FD 8
  RT 15Ø FD 15
  TRI 15
  SH 3Ø5 FD 8 SH Ø PU ST
END
```

Returning the turtle to its original position makes this erase possible.

Similar changes in **B** and **C** give

```
TO B        TO BB
   PC 2        PC 3
   B1          B1
END         END

TO C        TO CC
   PC Ø        PC 3
   C1          C1
END         END
```

We have not bothered to reprint the original versions of **C** and **B**, which must be renamed **C1** and **B1**.

While we are at it, we should allow for double keystrokes of the DOODLE mode commands. One example should be sufficient.

```
TO 77
  PC 3 BK 1
END
```

We are in effect building a special language consisting of one-keystroke commands. Because of the low frustration tolerance of the audience that we are building the special language for, it is especially important to make the language "user-proof". To do that we should define a procedure for all the other keys on the keyboard. The procedures are

```
TO A
TO D
. . .
END
```

Note that we skipped **B** and **C** because they are actually used. Note also that it is not necessary to have individual **END** statements for each procedure because the following **TO** statement automatically ends each procedure. These procedures prevent the message

```
I DON'T KNOW HOW TO ...
```

if the child accidentally enters an unlabeled key.

In this chapter we have introduced the idea of building shapes or complex picture elements which the young child can call forth with single keystrokes. The examples we have given are simple, but the only limit to what is possible is your time and imagination. Let's now start thinking about ways to use these tools with very young children.

14. USE OF DOODLE MODE AND OK SET

In the last two chapters, we covered the operations of the DOODLE mode and the OK Set. What is possible and what is worthwhile are two separate questions. In this chapter we will pass along some suggestions for worthwhile activities. Our suggestions are aimed at the adult who is working with small children. We have collected ideas from a number of sources. However, we should make it clear that, because Super LOGO offers possibilities for working with much younger children than could be reached previously, no one at this time really knows what is possible or what is most beneficial. Also remember that this is a user's manual for a computer language, not a textbook on early childhood education. Don't be hesitant to question our suggestions, and don't hesitate to try out new ideas.

Perhaps the best way to start with very young children is to let them play. By "play," we mean allow them to explore the effects of the various keys. If the children are very young, this will take quite a bit of time. If you've changed the shapes available in the OK Set since the last session, then you should give the child another chance to explore the new set of keys. Keep in mind that a child's attention span is not as long as yours, so don't try to prolong the sessions. Our own first ideas for Color LOGO grew from an effort to create something for a four-year-old to do because he wanted to be like Dad and "work on the computer". This suggests that another way to start is to master DOODLE mode yourself and to prepare a set of procedures for the OK Set. Then you'll be ready when an interested face appears at your shoulder some evening.

The users of LOGO have had consistent success with one technique for getting children started. They repeatedly relate the turtle commands with body movement. That is, ask the children to play turtle and ask them to keep track of the turtle movements they make. Thus, if the task is to draw a box, the child is asked to walk in a box-shaped path and then to tell the turtle what he or she did. The success on which this recommendation is based comes from work with somewhat older children, so it may not be quite as effective with the pre-reading group. It might help to give them objects to actually walk around to make the shape less abstract.

One heavily used technique in early childhood teaching is to ask the student to copy something. A book titled *Mathematics Their Way* by Mary Baratta-Lorton (Addison-Wesley Publishing Company, 1976) makes use of this technique for beginning mathematics and is a rich source of ideas for DOODLE mode projects. Basically, the approach is to write a procedure which will draw a figure or shape. This can be placed on the screen in RUN mode. The child is then asked to copy, complete, fill in, invert, rotate, or in some way proceed with reference to the figure on the screen. If the procedure the child develops while doing this is likely to be worth keeping for future use, then the child should be working in DOODLE mode. If it is not worth keeping, or if it requires the more complex shapes, then the child should be working with the OK Set of procedures.

Let's turn now to some specific activities. A large number of exercises could be built around the idea of pattern continuation and generalization. These activities are best suited to the OK Set, so the following procedures should be added to that set. One of the simplest types is a pattern which can be imposed on a line of dots. First we draw two parallel lines of dots.

```
TO DOTS
  CLEAR
  HT RT 90
  SX 5 SY 150
  LINE-OF-DOTS
  SX 5 SY 50
  LINE-OF-DOTS
END

TO LINE-OF-DOTS
  REPEAT 20 (BIGDOT FD 12)
END

TO BIGDOT
  FD 1 PD RT 90 FD 1
  REPEAT 4 (RT 90 FD 2)
  PU
  BK 1 LT 90 BK 1
END
```

We could have used the LOGO primitive **DOT** (a predefined LOGO command described on page 172) instead of **BIGDOT**, but later on we'll want to be sure that the dot is centered on the starting point. Note that we could not use the name **DOT** instead of **BIGDOT** for the procedure, for the computer would use the primitive instead of the procedure.

Next we draw some very simple repeating pattern on the upper row of dots. We'll use **DOTS** in the pattern drawing procedure. Several examples follow.

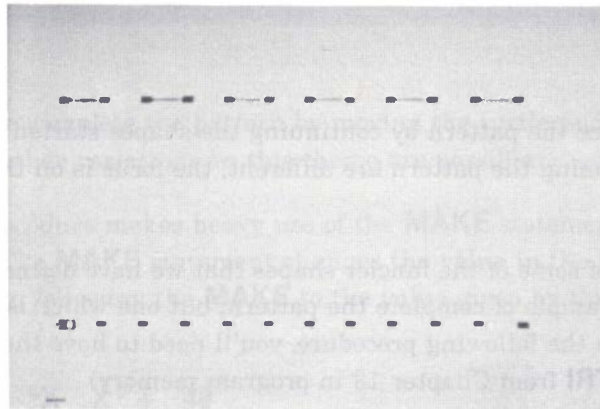
```
TO PATTERN1
  DOTS
  SX 5 SY 150
  REPEAT 10 (PD FD 12 PU
             FD 12)
  SX 5 SY 50 ST PD
END
```

Try running **PATTERN1**. The idea is that the child is to reproduce the pattern of the upper set of dots on the lower set, and then, after that is mastered, the child is to give an equivalent pattern with some other shapes. Before you try this with a child, try it yourself! Try to copy the pattern on the lower row of dots. You'll find that it is more difficult than necessary. It is needlessly difficult to move the turtle the correct number of units (12 as the procedure is written). We can make the exercise much less bothersome by some minor adjustments. Notice that these adjustments in no way detract from the point of the exercise, which is to recognize and continue the pattern.

```
TO LINE-OF-DOTS
  REPEAT 12 (BIGDOT FD 20)
END
```

The change to **FD 20** means that the child can connect dots with two keystrokes (key [8] producing **FD 10** on each stroke). We have to adjust **PATTERN1** as well.

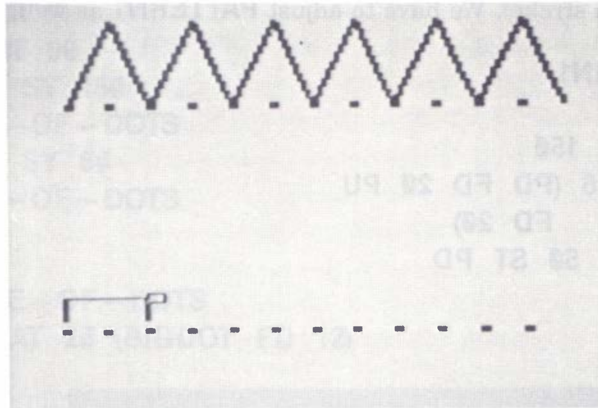
```
TO PATTERN1
  DOTS
  SX 5 SY 150
  REPEAT 6 (PD FD 20 PU
            FD 20)
  SX 5 SY 50 ST PD
END
```



You may be wondering why we didn't just give you the final versions immediately. The point is that we hope you will try creating your own exercises, and we want you to see that a little attention to detail can make the exercises much more effective. Be sure to try out the task to check the difficulty level before the children are around. This is supposed to be fun as well as instructive, not a new source of frustration.

The same pieces can be used for a slightly more difficult exercise.

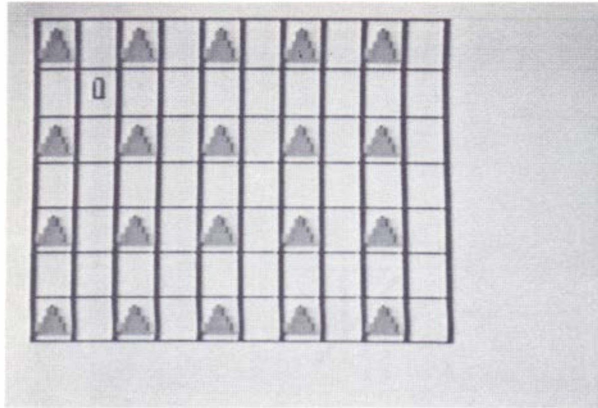
```
PATTERN2
  DOTS
  SX 5 SY 150
  REPEAT 6 (PD LT 60 FD 40
            RT 120 FD 40
            LT 60)
  SX 5 SY 55
  LT 90 FD 10 RT 90 FD 40
  RT 90 FD 10 BK 10 LT 90
  ST
END
```



Here the task is to reproduce the pattern by continuing the shapes started on the lower line. Because the two shapes forming the pattern are different, the focus is on the shape rather than straight copying.

We may as well make use of some of the fancier shapes that we have defined in the OK Set. The following is another example of complete the pattern, but one which is visually more interesting. (In order to run the following procedure, you'll need to have the final versions of the procedures **T**, **T1**, and **TRI** from Chapter 13 in program memory)

```
TO PATTERN3
  MAKE "X Ø MAKE "Y 50
  CLEAR HT
  REPEAT 1Ø
    (REPEAT 7 (SX :X SY :Y
      SQUARE MAKE "Y :Y+2Ø)
    MAKE "X :X+2Ø MAKE "Y 5Ø)
  MAKE "X 11 MAKE "Y 58
  REPEAT 4
    (REPEAT 5 (SX :X SY :Y
      T MAKE "X :X+4Ø)
    MAKE "X 11 MAKE "Y :Y+4Ø)
  SX 31 SY 158 ST
END

TO SQUARE
  REPEAT 4 (FD 2Ø RT 9Ø)
END
```



The child's task is to complete the pattern by moving the turtle and by pressing the key with the triangle. Many other variations on this theme are possible.

The **PATTERN3** procedure makes heavy use of the **MAKE** statement, and we have not discussed that before. The **MAKE** statement changes the value in the memory space referred to by the variable name following the **MAKE** to the value given by the next expression or number. For example

```
MAKE "X :X + 40
```

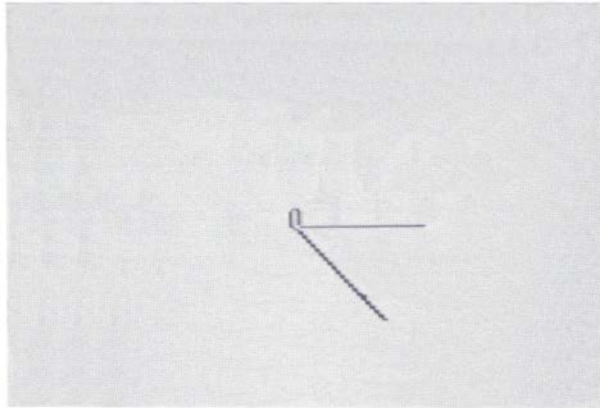
replaces the starting value in the memory position named **X** (name **X** indicated by **"X**) with a value which is 40 greater than the current value. Notice that LOGO distinguishes between the name of the variable (**"X**) and the contents of the variable (**:X**). Not all versions of LOGO make this distinction, so Super LOGO also accepts the syntax

```
MAKE :X :X + 40
```

Of course not all the tasks using the triangles, squares, and circles need to be directed towards specific goals. Ask the child to create a design, or to create a border to the screen.

Another group of projects can be based on completion of design. The screen can be thought of as consisting of four quadrants divided at the home position. The idea is to have the turtle draw a pattern in one quadrant and to have the child complete the pattern in the other three quadrants. Either DOODLE mode or the OK Set can be used here. If you've included the erase procedures for the DOODLE turtle commands in the OK Set, then that set is preferable. A simple pattern is

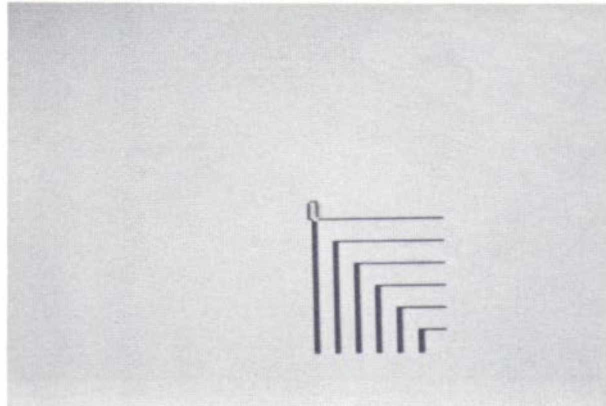
```
TO PATTERN4
  CLEAR
  RT 90
  REPEAT 2 (FD 60 SX 128 SY 96
            RT 45)
  HOME
END
```



We have written the procedure so that it is easy to add lines to make a more elaborate pattern. However, we recommend that you restrict the patterns to those using angles which are easy to produce in DOODLE mode (that is, multiples of 15 or 45 degrees). We reset the turtle to the home position with the **SX** and **SY** instead of with **HOME** so that the turtle heading is preserved. Again the **FD** should be some multiple of 1Ø to minimize the number of keystrokes needed.

This gives a more complex pattern.

```
TO PATTERN5
  CLEAR
  LINES 60 128 1Ø
  HOME
END

TO LINES :LENGTH :X :STEP
  IF :LENGTH = Ø (STOP)
  SX :X SY 36 SH Ø
  FD :LENGTH RT 90 FD :LENGTH
  LINES (:LENGTH—:STEP)
        (:X + :STEP) :STEP
END
```



The starting points for the pattern are picked so as to center the pattern on the home position. Thus because home is at 128,96 the starting point for the first line is at 128,36 which is 60 units below home. We've chosen to orient the pattern so that the child can begin drawing without turning the turtle.

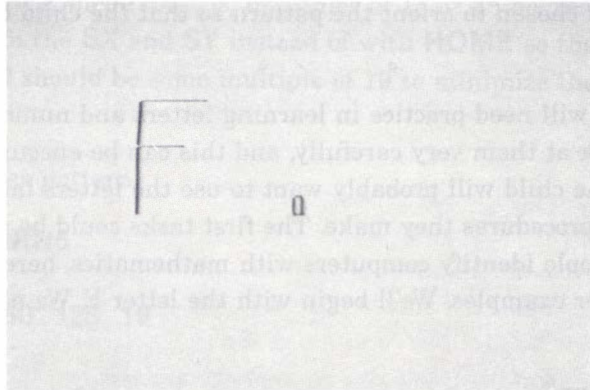
At some point the child will need practice in learning letters and numbers. Part of learning to recognize them is to look at them very carefully, and this can be encouraged by use of DOODLE mode activities. The child will probably want to use the letters later to write simple words, so we'll save the procedures they make. The first tasks could be simply copying from a model. Because most people identify computers with mathematics, here we'll counter that tendency by using letters for examples. We'll begin with the letter F. We need a procedure to draw the model.

```
TO DRAWF
  CLEAR
  SX 50 SY 146 RT 180
  FD 50
  SX 50 SY 146 LT 90 FD 30
  SX 50 SY 126 FD 20
  HOME
END
```

This will draw a large capital F, as you can see by running the procedure. However, it will draw the F so quickly that it gives the child no hint as to the order in which the lines should be drawn. The order can be indicated in several ways. Color can be used (draw the red part, then draw the blue part). We can put delays between the strokes to make the sequence on the example visible. We'll use both techniques.

```
TO DRAWF
  CLEAR
  SX 50 SY 146 RT 180
  PC 1 FD 50
  WAIT 30
  SX 50 SY 146 LT 90
  PC 2 FD 30
  WAIT 30
  SX 50 SY 126 FD 20
  WAIT 30
  HOME
END
```

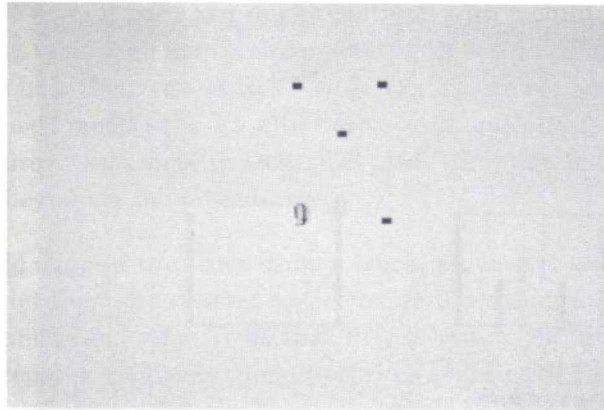
Note: The **DRAWF** procedure could not be called **DRAW–F** because LOGO command words such as **DRAW** cannot be used as part of a hyphenated procedure name.



The command **WAIT** does nothing but wait the indicated number of tenths of a second; here it waits 3 seconds each time. Be sure that the child's procedure is named **F** so that there is a simple correspondence between the name and the drawing. If you still have the OK Set in memory, you'll have to delete **F** from that set.

Once the child is familiar with the shape of the letter, or of several letters, you can let the child try making letters by connecting dots. Here the procedure must draw the dots, preferably starting at the home position. (To run the procedure **DOTM**, you'll need to have the **BIGDOT** procedure from Chapter 13 in program memory.)

```
TO DOTM
  CLEAR
  BIGDOT FD 60 BIGDOT RT 135
  FD 30 LT 45 BIGDOT
  LT 45 FD 30 RT 45
  BIGDOT RT 90 FD 60 BIGDOT
  HOME
END
```

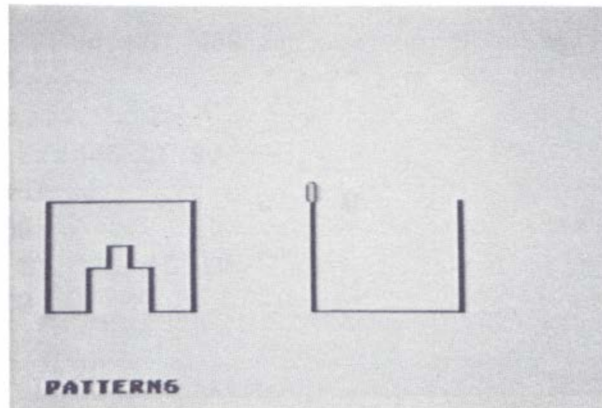


In this procedure we've been careful to always have the turtle pointing in the horizontal direction to keep the spacing of the dots perfectly regular. That may not be essential.

This dot-to-dot exercise works best for those letters and numbers where the pen never need be raised. Most letters require that the pen be raised. The dot pattern for these may be a bit of a puzzle, perhaps a worthwhile challenge. If that is too difficult, color coding the dots into two or three sets or adding intermediate dots may help.

The ability to visualize how things will look in other positions is worth developing. The idea here is to give a figure in one position and to ask the child to doodle it in another position. We are going to reuse the child's procedure for final comparison, so here we use DOODLE mode. One task is to ask the child to complete a partially drawn figure, but in another position.

```
TO PATTERN6
  RT 180
  SX 70  FD 50  RT 90  FD 20  RT 90
  FD 20  LT 90  FD 10  RT 90  FD 10
  LT 90  FD 10  LT 90  FD 10  RT 90
  FD 10  LT 90  FD 20  RT 90  FD 20
  RT 90  FD 50  RT 90  FD 70
  SX 198 SY 96  SH 180
  FD 50  RT 90  FD 70 RT 90  FD 50
END
```



Remember that in entering DOODLE mode it will be necessary to name the procedure that the child is creating. The comparison of the two figures can be made nicely. Let's assume that the child's procedure is named **ZZ**. After **ZZ** is completed, get into RUN mode and do the following.

```
RT 180 PATTERN6
SX 70 SY 146 PC 2 ZZ
```

This will rotate the original figure and draw the child's figure over the rotated original in another color. The result will be even more satisfying when the child is drawing the whole figure in the new position. Of course the above set of instructions could be combined into another procedure so as to speed the comparison.

The DOODLE mode projects can become quite complex. For example, a long DOODLE mode project could be teaching the turtle to write in handwriting. The key is to name each procedure for drawing a cursive letter with that single letter as the name. Thus, cursive a should be given the procedure name **A**, cursive b the procedure name **B**, etc. Then in RUN mode every time a letter is typed (followed by **ENTER**) the cursive letter will be drawn. Or one could define a procedure with a word spelled out (spaces between the letters) and the result would be the word in cursive letters.

```
TO CURSIVE
  C A T
END
```

will write "cat" in cursive if the procedures **C**, **A**, and **T** are correctly defined. To make it all work smoothly, the turtle will have to end up in the right position after each letter.

If that is not enough of a challenge, then how about making the computer draw letters as they appear in a manual on calligraphy? No doubt you will have to make them a bit bigger to get the desired effect. The only limitation is that you can't have both upper and lower case letters at once. There are limits, even with Super LOGO!

This chapter has given some idea of what can be done with the OK Set and DOODLE mode. At this point there are several ways you might continue with a child. One is to continue with DOODLE mode giving them ever more challenging tasks or encouraging them to create useful procedures which you help them use in RUN mode. If they are reasonably good with the keyboard, they may not need much help. An alternative is to teach them how to extend the OK Set by adding procedures they create in DOODLE mode (they can make their own key labels and attach them as they name the procedures).

For example, you could suggest that they draw a truck, a house, a tree, and a person in DOODLE mode, and then let them draw street scenes using these additions to the OK Set. Of course they'll soon want to add color. To do that they'll have to learn how to insert **PC** commands into their procedures, and before long they'll be typing and editing in the standard way.

15. ADDITIONAL EDITING FEATURES

There are some additional features of the editor which are useful when we have longer sets of procedures to edit. We'll introduce them in this chapter.

There is a way to delete more than one character at a time. When you are in EDIT mode, [SHIFT] [CLEAR] (that is, holding down the [SHIFT] key and pressing the [CLEAR] key) deletes from the current cursor position to the end of the line. If the cursor is at the left margin, this operation deletes the whole line. Before you try this, make absolutely certain that you are in EDIT mode because these same keys clear memory (delete all lines) in BREAK mode.

The [SHIFT] [↑] combination initiates one of several fast-forward sequences. The first response to the combination is that the cursor is moved to the bottom of the screen. At this point, the computer waits for additional input from the keyboard. This input can be of three types.

One possibility is to enter the [SHIFT] [↑] combination a second time. This causes the text lines to scroll up until the process is interrupted or until the end of the procedures in memory is reached. You can interrupt the process by pressing any key.

A second possibility is to enter up to 16 characters. This string of 16 characters becomes a search string. That is, the computer searches through the procedures in memory, starting from the current cursor position, for an occurrence of the search string. If a copy of the search string is found, the search is stopped at that point. Otherwise, the search continues to the end of the set of procedures in memory. This possibility can be used to locate errors (for example, mistyped procedure names), or simply to skip ahead to a known procedure name.

The third possibility is to press [ENTER]. This tells the computer to reuse the previous search string and to look for its next occurrence. This process can be repeated as often as desired. Notice that if you press [ENTER] after the first time you use the [SHIFT] [↑] combination, you are telling the computer to search again for no characters, which it will find immediately. This sounds silly, but it is easy to press [ENTER] inadvertently, especially when nothing seems to be happening.

To make sure that you know how to make use of this very useful feature, we'll give a specific example. We'll assume that you saved the procedures from last chapter and that you want to locate all occurrences of the procedure named **BIGDOT**. With the procedures in memory, get into the EDIT mode and enter

[SHIFT] [↑] **BIGDOT** [ENTER]

Remember [SHIFT] [↑] means two keys, and [ENTER] means one key. The computer will scan down through the procedures until it finds **BIGDOT**. To find the next one, enter

[SHIFT] [↑] [ENTER]

Again the computer scans until it finds **BIGDOT** a second time. To search for the third occurrence, enter [SHIFT] [↑] [ENTER]

and so on. A complete list of operations available in EDIT mode is contained in the EDIT mode section of Appendix I.

16. MULTIPLE TURTLES

So far we have been drawing on the screen with a single turtle. Complex figures were made by drawing one piece at a time, first one piece completely, then a second piece, and so on, until the drawing was complete. In Super LOGO there is another way to produce a complex drawing. We can draw all the parts at once using several turtles.

Multiple turtles provide many possibilities. Games are one obvious application. It is much easier to program games if each player is assigned a turtle which maintains its position until that player's next turn. In other applications we can make drawings by assigning to turtles the tasks of drawing individual pieces of the whole. In this way the drawing will seem to evolve instead of appearing piecemeal. At a more serious and abstract level, we can use the multiple turtles to illustate the process of multiprogramming.

Let's begin by entering a few procedures for the turtles to run. Clear the memory and enter

```
TO BOX :SIDE :X :Y
  SX :X SY :Y
  REPEAT 4 (FD :SIDE RT 90)
END

TO CIRCLE :SIDE :X :Y
  SX :X SY :Y
  REPEAT 20 (FD :SIDE RT 18)
END
```

Run each of these to verify that they are entered correctly.

We create new turtles by means of the **HATCH** command. The form of the command is

HATCH *turtle-number procedure-for-the-turtle*

or, to give a specific example of the format,

```
HATCH 1 BOX 50 30 60
```

(**HATCH** needs to be used inside a procedure, so don't try out the command just yet, or the command will be ignored.) Here the meaning of **HATCH** is obvious. The first number, here 1, is the name or label of the turtle. Turtles can be labeled with any number between 1 and 254. (Turtle 0 is the master turtle — always present, even if hidden by a **HT** command — which we have been using exclusively up to now.) **BOX** is the name of the procedure which we are telling turtle 1 to run. The numbers following **BOX** are, as usual, the values to be fed into the local variables within **BOX**.

Next we try out a simple multiple turtle program. Enter the following

```
TO TEST1
  HATCH 1 BOX 50 30 60
  HATCH 2 BOX 40 180 90
  HATCH 3 BOX 60 100 90
END
```

Notice that each turtle has its own procedure and its own set of values for the variables. Of course, several turtles can be using the same procedure, but each still has its own current set of values for the variables.

When you run **TEST1** you may get less than you expected. Why does the program stop before drawing the boxes? Notice that there are four turtles on the screen: the three you created with the **HATCH** commands, and as always, the master turtle. When there are multiple turtles, Super LOGO gives each turtle a turn in sequence. A turn is a single turtle command or a logical operation in a control statement. (A control statement is one which controls the sequence of operations in the procedure, for example an **IF** or a **REPEAT**.) Turtle Ø uses a turn to create turtle 1, and then the computer gives turtle 1 a turn. Next it is turtle Ø's turn again and it creates turtle 2; then the computer gives turtles 1 and 2 each a turn. Next turtle Ø uses its turn to create turtle 3, and the computer gives turtles 1, 2, and 3 each a turn. Again it is turtle Ø's turn and it encounters the **END**. Turtle Ø is now waiting for something to do. We have not given turtle Ø anything else to do, so it is waiting for a command from the keyboard. If we press [ENTER] (a command for turtle Ø to do nothing), then all the other turtles get another turn. Try it.

Of course we do not always want to have to sit at the keyboard pressing [ENTER]. We can get the whole thing to work as planned if we give turtle Ø some procedure to run as well. Try

```
TO TEST1
  HATCH 1 BOX 5Ø 3Ø 6Ø
  HATCH 2 BOX 4Ø 18Ø 4Ø
  HATCH 3 BOX 6Ø 1ØØ 2Ø
  BOX 2Ø 15Ø 12Ø
END
```

The last call of **BOX** has no **HATCH** preceding, so it is addressed to turtle Ø. That's more like it! If you want to see in a bit more detail what is actually happening, you might want to slow down the speed. You can slow any procedure by inserting a **SLOW** command.

```
TO TEST1
  SLOW 3Ø
  HATCH 1 BOX 5Ø 3Ø 6Ø
  HATCH 2 BOX 4Ø 18Ø 4Ø
  HATCH 3 BOX 6Ø 1ØØ 2Ø
  BOX 2Ø 15Ø 12Ø
END
```

The number after **SLOW** tells the computer how much to slow down. The number must be between Ø and 127. Zero is full speed and 127 is the slowest speed. The **SLOW** command sets a speed for all procedures which will remain unchanged until reset with another **SLOW** command or until RUN mode is exited and then reentered. **TRACE** is not as useful here because it is not always obvious which turtle is running the line displayed at the bottom of the screen.

Before we leave this example, notice that at the completion of each turtle's procedure the turtle disappears so that at the end only turtle Ø remains.

Of course we can use different procedures for the various turtles. Try

```
TO TEST2
  HATCH 1 BOX 5Ø 3Ø 6Ø
  HATCH 2 BOX 4Ø 18Ø 9Ø
  HATCH 3 BOX 6Ø 1ØØ 2Ø
  HATCH 4 CIRCLE 3 3Ø 14Ø
  HATCH 5 CIRCLE 4 18Ø 12Ø
  CIRCLE 5 9Ø 9Ø
END
```

This procedure can be used to point out one potentially troublesome point. What if we altered the procedure by making the procedure for turtle Ø **BOX** (say **BOX 8Ø 9Ø 9Ø**)? If you try this, you will find that the circles are not completed and that the two turtles drawing the circles remain on the screen. This is because turtle Ø runs out of commands before the others are finished. To avoid the problem, always put the procedure for turtle Ø last, and assign turtle Ø the most complex procedure.

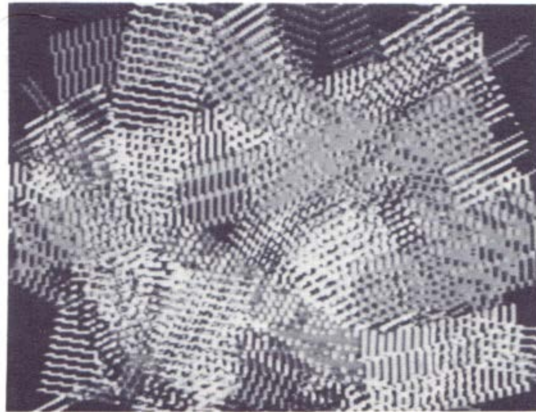
Another solution to the problem mentioned above is contained in the procedure **ABSTRACT**.

```
TO ABSTRACT
  CLEAR DRAW COLORSET 1
  RT 25
  HATCH 1 PATH 1 4 3Ø
  RT 43
  HATCH 2 PATH 2 4 2Ø
  RT 67
  HATCH 3 PATH 3 4 4Ø
  RT 1Ø5
  HATCH 4 PATH Ø 4 1Ø
  VANISH
END
```

Notice the RT commands to turn turtle Ø between each **HATCH**. The initial position and heading of each new turtle is the same as that of the parent turtle (the turtle which hatches the new one). In this example turtle Ø is the parent, so each new turtle will have the position and heading of turtle Ø at the time of **HATCH**ing. After the four new turtles are created, then turtle Ø is given the **VANISH** command. The **VANISH** command tells a turtle to go out of existence. Once turtle Ø is out of existence, it no longer gets a turn, and it cannot bring the procedure to a halt by running out of commands.

Of course, the procedure **ABSTRACT** needs **PATH** to function.

```
TO PATH :COLOR :I :L
  HT PC :COLOR
  WHILE 1=1
    (FD :L RT 90 PU FD :I
    RT 90 PD FD :L
    LT 90 PU FD :I LT 90 PD
    IF NEAR 255 > 150
      (RT 108)
    )
END
```



PATH contains some new ideas which we should explain. The first is the **WHILE** statement. The **WHILE** is somewhat like a **REPEAT**, but with a condition. The most common use is to repeat while some condition is true (for example, **WHILE :X < 3**). The computer evaluates the condition and returns the value zero if the condition is false or a non-zero value if the condition is true. Here we want it to repeat forever, so we assign the condition 1 = 1 which always is true. The parentheses following the **WHILE** enclose the commands which are to be repeated. The other new idea is the use of the **NEAR** function. The **NEAR** function returns an indication of the distance from the current turtle to the designated turtle. Actually, what you get is the total of the steps in the x direction and in the y direction to the designated turtle. In **PATH** the statement is

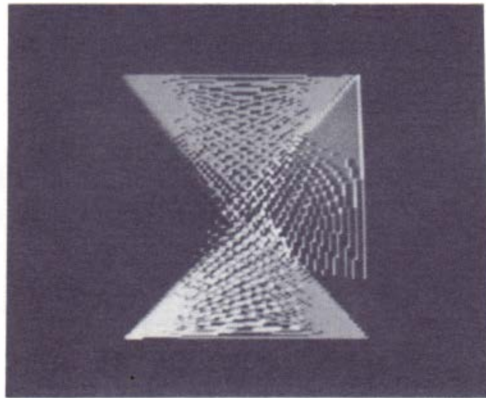
```
IF NEAR 255 > 150 (RT 108)
```

The current turtle (remember 1, 2, 3, or 4) is asking the distance to turtle 255. But turtle 255 does not exist. When you request the distance to a non-existent turtle, you get the distance to the home position. Therefore this statement says, if the current turtle is more than 150 steps away from home, then turn right 108 degrees.

After such a long explanation, we should get a program which runs a long time. **ABSTRACT** will surely fit the bill; it will run until you hit the **BREAK** key or until there is a power failure.

Perhaps you'd like a different design.

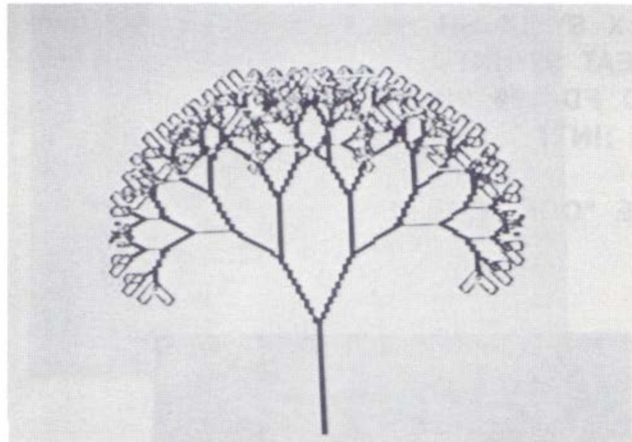
```
TO MIXIT
  DRAW COLORSET 1 BG Ø
  HATCH 1 SWEEP 1 3  6Ø  3Ø   Ø
  HATCH 2 SWEEP 2 3  6Ø 16Ø  9Ø
  HATCH 3 SWEEP 3 3 19Ø 16Ø 18Ø
  HATCH 4 SWEEP 2 3 19Ø  3Ø 27Ø
  VANISH
END

TO SWEEP :COL :INT :X :Y :H
  REPEAT 12
    (HT PC :COL
     SX :X SY :Y SH :H
     REPEAT 92/:INT
      (PD FD 1ØØ PU BK 1ØØ
       RT :INT)
    )
    MAKE "COL :COL+1
END
```



There is no question that the turtles slow down as the number of turtles on the screen increases. After all, more is going on. Thus far we haven't had so many that the slowing is that noticeable. But how about a program which generates a lot of turtles? One interesting test is to return to a recursion program and implement it using multiple turtles. **TREE1** is an ideal example. Try the following.

```
TO TREE1 :S
  IF ME=Ø (CLEAR DRAW SETY Ø)
  IF :S>6
    (FD :S LT 3Ø
    HATCH 1 TREE1 (3*:S/4)
    RT 6Ø
    HATCH 2 TREE1 (3*:S/4)
    VANISH
    )
END
```

TREE1 40 is an example that gives good results.



NOTE: Depending on the computer you are using, the **TREE1** procedure may cause you to run out of memory. You may see a partially drawn tree; or the procedure may terminate before the turtles are erased, giving the effect of a tree in full bloom. If that happens, try it again with a smaller number (for example, TREE 30).

Again we've introduced a new idea with this procedure, here the **ME** function. The **ME** function returns the identification number of the current turtle. The statement

```
IF ME=Ø (CLEAR DRAW SETY Ø)
```

says if the current turtle is turtle Ø, then clear the screen, use the full screen, and move. Because turtle Ø is subsequently told to **VANISH**, this will happen only once. This procedure recursively hatches new turtles, all named either 1 or 2. Because the recursive calls keep levels distinct, this is satisfactory, but functions like **NEAR** would give unpredictable results because the various turtles are not uniquely named.

There are a number of interesting things that can be tried with this procedure. One is to compare it in speed with the earlier version of **TREE**. In one case you have all the backing up necessary for a pure recursive program, and in the other you have the overhead necessary to keep track of all the turtles. To make the comparison meaningful you'll have to make the two versions draw the same size tree, but by now that will be easy. The comparison may give some idea of why multiprogramming is worth learning about. You can speed the multiple-turtle version by reducing the number of times the turtle has to be drawn. Simply insert a **HT** command as the first command in the procedure. One other change converts the tree into full blossom. Try

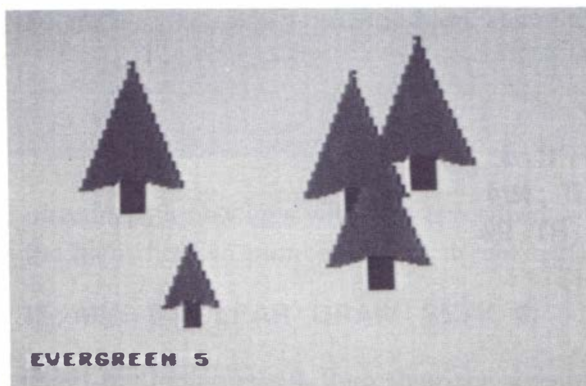
```
TO TREE2 :S
  IF ME=0 (CLEAR DRAW SY 0)
  IF :S>6
    (FD :S LT 30
    HATCH 1 TREE2 (3*:S/4)
    RT 60
    HATCH 2 TREE2 (3*:S/4)
    VANISH)
  ELSE (REPEAT 500 ())
END
```

The addition is the **ELSE** statement. The **ELSE** is a partner of the **IF** statement. The combination says if the current value of **:S** is greater than 6 then obey the commands in the following set of parentheses (from **FD :S** through **VANISH**), but if **:S** is not greater than 6 then obey the commands in the parentheses following **ELSE**. The commands following **ELSE** simply delay the completion of the procedure so that we can see the tree with a turtle at the end of each branch.

Trees are so easy to draw with multiple turtles that we may as well draw a complete forest. In fact we'll look at two forests, one a deciduous forest in winter and the other an evergreen forest in whatever season you like.

```
TO FIR1 :N :X :Y
  HT SX :X SY :Y PC 0
  BK :N/2 RT 90 FD :N/4
  LT 90 FD 6+:N/2 RT 90
  FIR11 :N :X
END

TO FIR11 :N :X
  PC 1 RT 15 FD :N
  LT 129 FD 3*:N
  WHILE XLOC ME>:X (FD 2)
END

TO FIR2 :N :X :Y
  HT SX :X SY :Y PC 0
  BK :N/2 LT 90 FD :N/4
  RT 90 FD 6+:N/2 LT 90
  FIR22 :N :X
END
```

```
TO FIR22 :N :X
  PC 1 LT 15 FD :N
  RT 129 FD 3*:N
  WHILE XLOC ME<:X (FD 2)
END

TO FIR :N :X :Y :T
  HT
  HATCH :T FIR1 :N :X :Y
  HATCH :T+1 FIR2 :N :X :Y
  IF :N>20 (STOP)
  FIR (:N+1) :X :Y :T
END

TO EVERGREEN :TREES
  DRAW HT
  WHILE :TREES > 0 (
    MAKE "X RANDOM 200 + 20
    MAKE "Y RANDOM 100 + 30
    MAKE "T :TREES*3
    HATCH :T FIR 2 :X :Y :T
    REPEAT 30 ()
    MAKE "TREES :TREES-1)
  VANISH
END
```



Try running this set of procedures, first with one tree (**EVERGREEN 1**) and then with several (**EVERGREEN 4** or **EVERGREEN 5**). With some TV sets you may be able to get green tops and brown trunks, so try playing with the color adjustment knobs. If not, you can always claim that they're intended to be blue spruce. There are a couple of new ideas in the last two procedures. In **FIR11** we have used the **XLOC** function. This returns the x screen coordinate of the designated turtle. Here **:X** is the starting point for the right half of the tree. When **XLOC** has returned to the starting point, the procedure is finished. In **FIR** notice the use of the variable **:T** to indicate the turtle number. In **EVERGREEN** we have introduced the **RANDOM** function. **RANDOM** produces a random integer between Ø and the argument –1. For example

```
RANDOM 2ØØ + 2Ø
```

adds 2Ø to a random number between Ø and 199. The result must be a number between 2Ø and 219. We do this to keep the trees away from the edge where the wrap-around will give some rather lopsided trees. Also note the use of **WHILE** in combination with the

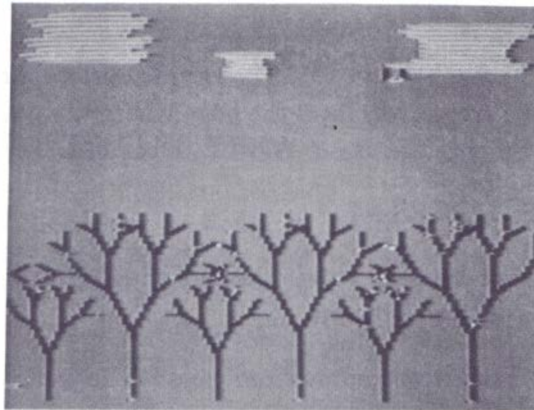
```
MAKE "TREES :TREES-1
```

This gives a number which is one less every time through the **WHILE** loop. **:TREES** is used to vary the turtle numbers for each tree drawn.

The deciduous forest uses the **TREE1** procedure from page 82. To that we must add

```
TO FOREST
  DRAW BACKGROUND 1
  SX 236
  REPEAT 3 (SY 1Ø
    SX XLOC ME + 4Ø
    HATCH 1 TREE1 2Ø
    SX XLOC ME + 4Ø
    HATCH 2 TREE1 3Ø )
  CLOUDS
END

TO CLOUD :SIZE :X
  SETHEADING 9Ø
  REPEAT (:SIZE/6)
    (MAKE "X RANDOM (:SIZE/2)
    PU FD :X/2 PD
    FD :SIZE-:X PU
    BK :SIZE-:X/2
    SY YLOC ME-2)
END
```

```
TO CLOUDS
  PC 2 SX 10 SY 180
  CLOUD 60
  SX 100 SY 164
  CLOUD 30
  SX 190 SY 176
  CLOUD 65
END
```



Again in this example we have created many turtles with the same numbers by hatching them recursively. The two multiple tree drawings show the two ways in which multiple turtles can be created. It makes no difference which way you do it unless you are going to refer to the turtle by number. In that case, each turtle must have a unique number.

Notice the statement

```
SX XLOC ME + 40
```

This has the meaning

```
SX (XLOC ME) + 40
```

not

```
SX XLOC (ME + 40)
```

Now that you are working on complex sets of procedures, you may want to interrupt the computer and pause at various times during execution, especially when you are testing. While procedures are actually running, pressing the BREAK key produces a pause. You can return to BREAK mode by pressing BREAK a second time, or you can continue execution of the procedures by pressing any other key.

17. NEW SHAPES FOR TURTLES

All turtles are created equal; at least they all look the same. In the examples we have seen so far, that didn't matter. But often we want the different turtles to look different. For instance, it would be impossible to play many games if all the pieces or players looked the same. So Super LOGO includes a way to change the shape of individual turtles. As we shall see, this gives us a bonus: a way to do simple animation.

The shape of the turtle is changed by means of the **SHAPE** statement. Following the word **SHAPE** is a list of turtle shape commands. Turtle shapes are drawn using a very limited set of turtle graphic commands, basically forward and back a single step, right or left by 45 degrees, and penup and pendown. The commands in a **SHAPE** statement have absolutely no effect on the turtle position, heading, or pen state. The symbols used for these commands are listed in the following table.

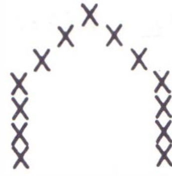
TURTLE SHAPE COMMAND	EFFECT
F	Step forward one dot. If the pen is down, complement (reverse the color of) the dot.
B	Step backward one dot. If the pen is down, complement the dot.
R	Turn right 45 degrees.
L	Turn left 45 degrees.
U	Pick up the turtle shape pen. This pen is always down at the start of a **SHAPE** command. The turtle shape pen is completely independent of the standard turtle pen; **PU** and **PD** commands have no effect on the turtle shape pen, and **U** and **D** have no effect on the turtle pen.
D	Put the turtle shape pen down. If the turtle shape pen was up, then putting it down will cause the current dot to be complemented.

Notice that because the only move forward or back possible is one dot, then the smallest turn which makes sense is a 45 degree turn.

This will be clearer if we give an example. Let us assume that the current orientation of the turtle is heading straight up. Then the command

```
SHAPE URRFFFLLDFFFFL–
FFFLLFFFLFFFF
```

will draw the following turtle shape.



Notice that the turtle shape commands can extend over more than a single line. Multiple lines must be connected with a hyphen and must start in column 1. This, in turn, means that there is no limit on the size or complexity of a turtle shape. However, the turtle shape must be redrawn every time the turtle moves, so the larger and more complex the turtle shape, the slower the system will run.

It is fairly difficult to create desired turtle shapes by trial and error at the keyboard. It is especially difficult to locate an error in the middle of a string of turtle shape commands. We have found the following to be effective ways to proceed. First design the turtle shape on a piece of graph or engineering paper. The possibility of rotating the paper as you enter the shape may save you from getting a stiff neck trying to play turtle at the keyboard. Once the shape is designed on the graph paper, there are two methods which we use. If the turtle shape is a simple one, enter the shape in DOODLE mode. Remember that the keys [3], [4], [5], [6], and [7] correspond exactly to the turtle shape commands **U**, **D**, **R**, **L**, and **F**. Only **B** is missing, and while **B** is very useful, it is not essential. Use of DOODLE mode continually shows you the current heading, which is a big help. However, the turtle drawn in DOODLE mode will not look exactly like the final turtle for several reasons. Lines drawn in DOODLE mode are two dots wide, but lines drawn as turtle shapes are one dot wide. Also, lines which cross, complement when they are turtle shapes. Lines which cross in DOODLE mode do not complement. However, you will get to see the shape in about the final form and of exactly the final size while drawing it. Once you have the shape completed in DOODLE mode, you can enter EDIT mode and convert the procedure into one to draw the new shape. Simply insert the **SHAPE** command before the command list and convert each **3** to **U**, each **4** to **D**, each **5** to **R**, each **6** to **L**, and each **7** to **F**, all by overtyping. (Of course, you will be converting the corresponding DOODLE mode symbols, not the numbers 3-7.)

Let's begin with a very simple example. We want the turtle to appear as an arrow. On graph paper we draw the dot pattern.

The actual turtle position is to be at the tail end of the arrows. Get into DOODLE mode, name the procedure **NEW**, and draw the figure. The keystrokes are

777777766677555537755747

You may be surprised at the small size of the turtle, but you can always draw a new one after you've learned the technique. Now enter EDIT mode and look at **NEW**. Insert **SHAPE** before the list of symbols and replace the DOODLE symbols with the appropriate turtle shape commands. Don't forget to include the hyphen at the end of the first line of commands. At this point your procedure should be

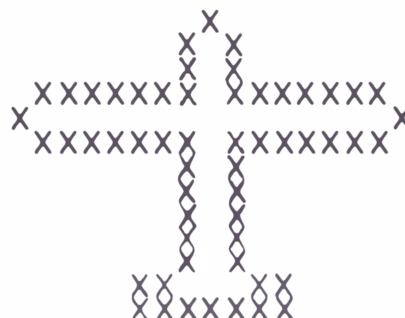
```
TO NEW
SHAPE FFFFFFFLLLFFRRRR–
UFFRRFDF
END
```

(Remember that lines which continue a **SHAPE** command cannot be indented.) To see how your shape looks, run **NEW** and then enter commands like **FD 2Ø** and **RT 9Ø**. In Super LOGO, the standard turtle can be drawn in 36Ø positions, but a turtle made with the "SHAPE" command has only 8 positions (heading up, down, left, right, or in one of four diagonal positions). Be sure to try the diagonal positions (for example, **RT 45**) because there will be some change in shape as the turtle rotates to these positions. To see why that is so, return to the graph paper and follow your turtle instructions beginning along a diagonal. It is a good idea to do this on graph paper before going to the computer, as your turtle shapes might come apart upon rotation if they are drawn in the wrong sequence. To show you that it can happen, try the following. We could have drawn essentially the same shape by the steps

FFFFFRRUFFDLLLFFLLFF

in the vertical or horizontal positions, but in the diagonal positions this pattern comes apart, as you can see if you follow the instructions on graph paper.

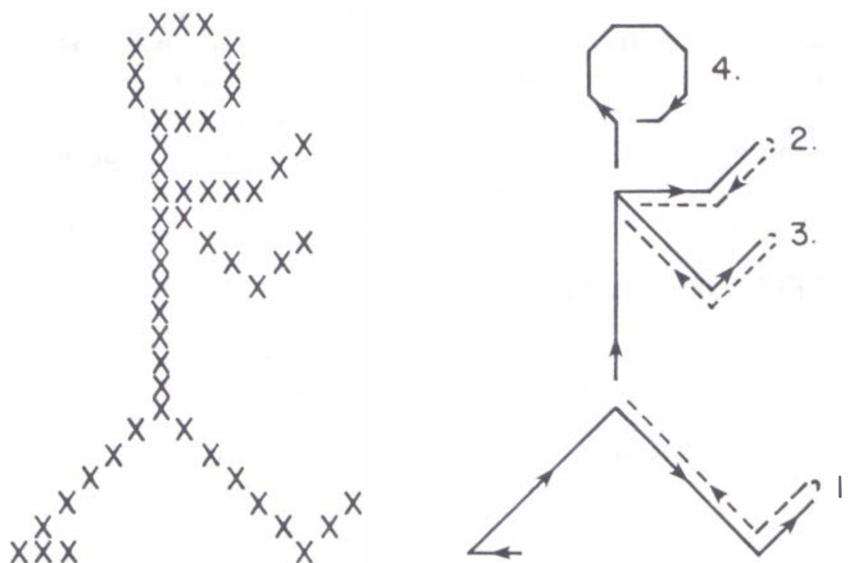
Now we move to a bit more complex example, an outline of a plane. The dot pattern is



and the procedure created in DOODLE mode and translated to EDIT mode is

```
TO PLANE
  SHAPE RRFFFLLFLLFRFR–
FFFFFFRRFFFFFFF–
LFLLFLFFFFFFFRRF–
FLFLLFLFFRRFFFF–
FFFLFLLFLFFFFFF–
FRRFFFFFFRFRFLL–
FLLFF
END
```

When we go to more complex shapes, we prefer to work with paper rather than with DOODLE mode. There are several reasons for this. One is that the **B** command is very useful. The other is that we must be very careful if we are to avoid problems when the turtle shape is turned to the 45 degree positions. As is our custom we will illustrate with an example. We want to use the following stick figure as a turtle shape.



We must choose our pathway through the figure carefully. The problem points are points where lines meet. To avoid problems, we avoid shortcuts and return to junction points by backtracking exactly. The pathway we take is indicated on the second figure. The dotted lines indicate backtracking with the pen raised. Remember, if we did not raise the pen when backtracking, then the lines would be complemented a second time (that is, erased). The only place where we don't backtrack is on the head. If a closed figure is symmetrical, then it will stay closed when rotated.

The other point to note carefully is when to raise and lower the pen. A dot will be complemented if the pen is down when we move into the dot or if the pen is lowered while we are in the dot. Notice that this means that if we draw a line and then cross that line with the pen down, the crossing dot will be erased.

Following the pathway indicated gives the following procedure. As stated earlier, we work this out on paper and enter it directly in EDIT mode so that we can use the **B** command.

```
TO ONE
  SHAPE LLULLFFFFDFFRRRFFFFFF–
RRFFFFFFLLFFUBBLLFFFFFFFRFD–
FFFFFFFFRRFFFFLFFUBBRBBBB–
RFDFFFLLFFUBBLLFFFFRFD–
FFLFRFFRFRFFRFRFFRFRF
END
```

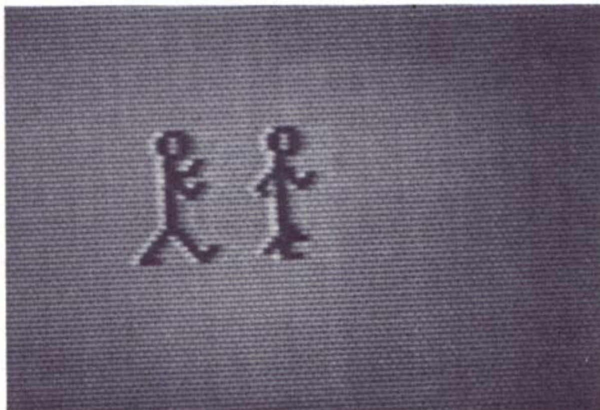
Try this out by running **ONE**. Try rotating it to other positions. Notice that when you turn it far enough it is upside down. You may not want turtles that do strange things like that for some purposes. You can sometimes avoid it, if it is a problem, by shifting your point of view. For example you might decide that it would be nice to have a turtle which actually looked like a turtle. If you draw a side view of the turtle, then it will look strange with headings like 180 degrees. But if you draw a top view of the turtle, then it looks fine in any orientation.

The reason that we drew the stick figure is that we want to show you how to use Super LOGO to do some very simple animation. We want to have a figure that will walk. We'll need another position for the stick figure, so we define another turtle shape. The process is the same as before.

Translating the indicated path into a procedure gives

```
TO TWO
  SHAPE LLURRFFDBBLLFFFFFFRRR–
FFFRRRFFFFLFFUBBRBBBBRFFFRFD–
FFFFFFFFFRRRFFFFLLFF–
UBBLLFFFFLLFDFFFLLFU–
BLLFFFFLFDFLFRFFRFR–
FFRFRFFRFRF
END
```

Now that we have the shapes, we can have some fun. First let's make them walk.



```
TO WALK
  HT PU SX 100 RT 90
  REPEAT 100 (ONE ST WAITA 100
    HT FD 6 TWO ST
    WAITA 100 HT FD 6)
END

TO WAITA :T
  REPEAT :T ()
END
```

Notice that in this case we want the turtle shape to be drawn at right angles to the turtle motion. That is taken care of in the **SHAPE** statement. Notice also that we have to slow down the process by including the **WAITA** procedure. Otherwise it runs so fast that we have trouble seeing the shape. Try other values for **:T** to vary the speed. We can make the figure climb; just enter **LT 15** before running **WALK** again. We can even make the figure walk in a circle.

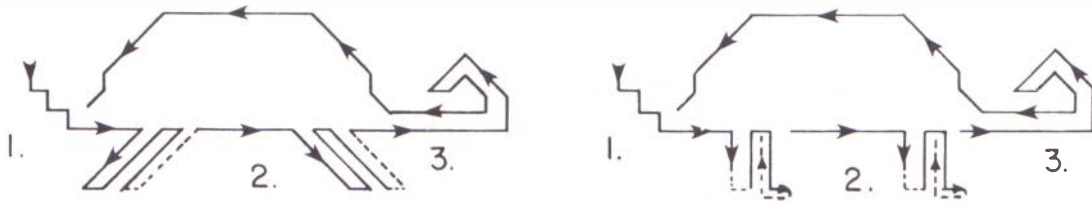
```
TO WALK-AROUND
  DRAW HT PU SX 100 RT 90
  REPEAT 100 (ONE ST WAITA 100
    HT RT 15 FD 6
    TWO ST WAITA 100
    HT RT 15 FD 6)
END
```

You may prefer the motion you get with a different control procedure. Try this as an alternative to **WALK**.

```
TO WALK1
  HT PU SX 100 RT 90
  REPEAT 100
    (HATCH 1 WALKA
    REPEAT 8 ()
    HT FD 6
    HATCH 1 WALKB
    REPEAT 8 ()
    HT FD 6
    )
END

TO WALKA
  HT ONE ST
  REPEAT 10 ()
END

TO WALKB
  HT TWO ST
  REPEAT 10 ()
END
```

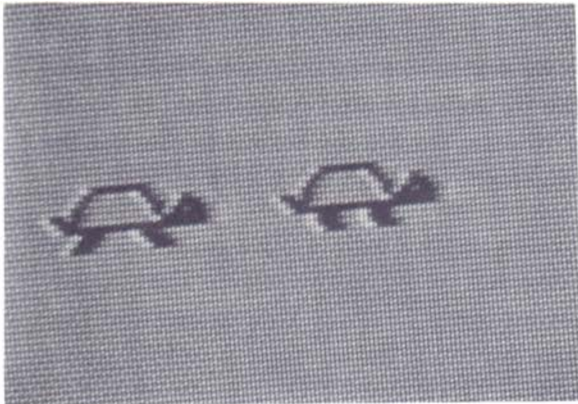
The trick here is to get the delays (that is, the **REPEATS** with empty parentheses) synchronized. The delays in **WALK1** must match with the delays in **WALKA** and **WALKB**. If there is a mismatch one direction, the two figures will appear together; and if there is a mismatch the other way, the motion will be unnecessarily jerky.

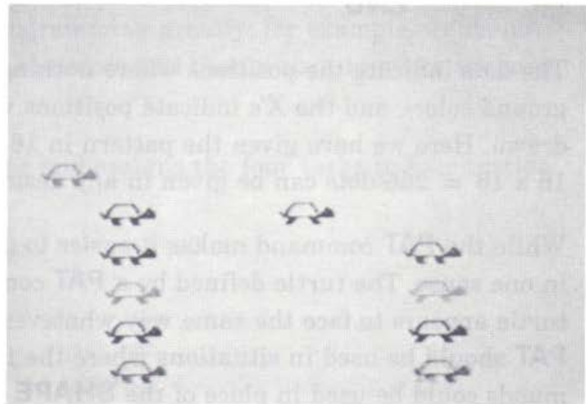
After all this talk about turtles, we feel an obligation to actually draw something which looks a bit like a turtle. As our next example, we give a **HERD** of turtles.




```
TO TURTLE1
  SHAPE LL-
BRRFRRFLLFRRFLLFFFFLBBBRFL-
FFFRFLBBBUFFFRFDFFFFFRFFF-
LFRBBBLFRFFFUBBBLFDFFFFFFFF-
LLFFLFFLLFFRBLBLLFRFRRFFFFF-
RFRFLFFFLFFFFFFFFLFFFLFRF
END

TO TURTLE2
  SHAPE LL-
BRRFRRFLLFRRFLLFFFFRRFFUBB-
LLFDRRFFFLLFFUBLLFDFFRRFFFFFFF-
RRFFUFLLFDLLFFFRRFRRFFFLLFUB-
LLFFFRRFDFFFFFFF-
LLFFLFFLLFFRBLBLLFRFRRFFFFF-
RFRFLFFFLFFFFFFFFLFFFLFRF
END

TO CRAWL :T :X :Y
  HT PU SX :X SY :Y
  RT 90
  REPEAT 100 (
     HATCH :T+1 T1
     REPEAT 8 ()
     HT FD 2
     HATCH 1 T2
     REPEAT 8 ()
     HT FD 2
     IF XLOC ME > 230 (VANISH)
  )
END

TO T1
  HT TURTLE1
  ST
  REPEAT 10 ()
END
```

```
TO T2
  HT TURTLE2
  ST
  REPEAT 10 ()
END

TO HERD
  CLEAR DRAW HT
  MAKE "I 0 MAKE "T 1
  REPEAT 20 (
    IF :I<10 (MAKE "I :I+1)
    MAKE "J 1
    WHILE :J<:I
      (HATCH :T CRAWL :T 0 (:J*18)
      MAKE "T :T+2
      MAKE "J :J+1)
    REPEAT 900 ()
  )
END
```





Super LOGO offers another way to change the turtle shape which is most useful for the creation of games. The command **PAT** is used to create a new pattern of 16 x 16 dots. The **PAT** must be followed by the pattern. For example, the sequence in the following procedure will change the turtle shape into a small person shape.

```
TO SPACEPERSON
PAT
................
.....XXXX.......
...XXXXXXXX.....
..XXXXXX..XX....
...XXXXXXXXX....
....XXXXXX......
.....XXX........
....XXXXXX......
...XXXXXXXXXXXX.
..XXXXXXXXX.....
.XXXXXXXXXXXX...
.XXXXXXXXXXXXXX.
.XXXXXXXXXXXXXX.
..XXX......XXX..
..XXX......XXX..
..XXXXX....XXXXX
PU RT 90 SETX 1
REPEAT 20 (FD 5 WAIT 3)
END
```

The dots indicate the positions where nothing should be drawn (that is, they remain the background color), and the **X**'s indicate positions where the turtle foreground color should be drawn. Here we have given the pattern in 16 rows of 16 dots. That is the easiest to see, but the 16 x 16 = 256 dots can be given in any desired arrangement (for example, 8 x 32 = 256).

While the **PAT** command makes it easier to give a sizable turtle which is filled in, it is limited in one sense. The turtle defined by a **PAT** command does not rotate on the screen. That is, the turtle appears to face the same way whatever the current turtle heading. This means that the **PAT** should be used in situations where the turtle does not rotate (for example, two **PAT** commands could be used in place of the **SHAPE** commands in **ONE** and **TWO** above for **WALK**, but not for **WALK-AROUND**), or in situations where rotation is not visible, for example, using a turtle as a sun which moves across a scene).

It is clear that while much is possible with the turtle shapes, Super LOGO is not likely to become a tool for the generation of Saturday morning TV shows. It was never intended that it should be so. It is a tool that will allow the child to produce results which can be immensely satisfying to the creator.

18. TURTLE GAMES

One of the most popular applications of computers is gaming. Super LOGO can be used to create a great variety of games. In this chapter we will give two examples of turtle games. These are included not as competitors for the local video arcade, but as illustrations of some very useful techniques for communication between turtles.

Before getting into the details of the simple game we're going to use, we want to point out a few things which may be obvious. Most of the popular video-arcade and computer games rely very heavily on speed. Things happen which force the players to react faster and faster until finally they fail. You've already gotten some feel for the speed at which animation runs in Super LOGO; it's not going to be fast enough to create shoot-em-up space games that will hold interest for long. However, it does have capabilities such that the user can create rather than just play such games. If you want to create games which will also be challenging to play using Super LOGO, then you might try to think of games where coordination of several moving objects is the challenge (thus lower speed is no limitation) or games where there is sufficient strategy that the player must think while playing.

Our first sample game is called **CATCHEM**. There are two players (or one two-handed player) who manipulate objects on the screen by pressing keys on the keyboard. The object is for the chaser to catch the runner. When the chaser catches the runner, the scorekeeper changes the score, and a new chase starts. There is an advantage to using multiple turtles here, as we can assign each turtle one task. This simplifies the programming greatly; for example, we do not have to move a cursor to the scoreboard to change the score and then return to make the next move.

The master procedure simply names the procedures and assigns the four tasks to four turtles. We use turtle Ø and three others.

```
TO CATCHEM
  CLEAR DRAW
  HATCH 1 GETKEYS
  HATCH 2 RUNNER 2Ø
  HATCH 3 CHASER
  SCOREKEEPER Ø
END
```

The names of the procedures are pretty descriptive. **RUNNER** controls the runner, and **CHASER** controls the chaser. **SCOREKEEPER** keeps the score. **GETKEYS** reads input from the keyboard. Of course the various turtles need to communicate, and that is the main new idea we will illustrate in this example.

Let's begin with the keyboard.

```
TO GETKEYS :X
  HT
  WHILE 1=1 (MAKE "X KEY
    IF :X = 'S (SEND 2 1)
    IF :X = 'D (SEND 2 45)
    IF :X = 'A (SEND 2 315)
    IF :X = 'K (SEND 3 1)
    IF :X = 'L (SEND 3 45)
    IF :X = 'J (SEND 3 315)
  )
END
```

First we see a trick we used before: the use of **WHILE 1=1** as an effective *REPEAT FOREVER*. The second new item is the **KEY** function. The **KEY** function looks at the keyboard to see if any key has been pressed. If no key has been pressed, then **KEY** returns the value Ø. Thus, if at the time turtle 1 is executing the statement

```
MAKE "X KEY
```

no key is depressed, then the variable **"X** is given the value **Ø**. If on the other hand a key is depressed, then the variable **"X** is given the ASCII value of the key. So the **KEY** function returns either the ASCII value of the key depressed or Ø if no key is depressed. The ASCII value is a number assigned to each key on the keyboard according to an industry-wide convention. In this procedure we do not have to know what the particular number is because the literal (for example, **'S**) automatically computes the ASCII value as well.

The next task for this procedure is to recognize which key has been depressed and to send a message to the appropriate turtle. We have to decide what keys to use for what actions of the runner and the chaser. We decided on the following key assignments.

S — move runner forward

A — turn runner left

D — turn runner right

K — move chaser forward

J — turn chaser left

L — turn chaser right

So now we see what (if any) key was pressed. First look at the statement

```
IF :X = 'S (SEND 2 1)
```

The literal **'S** gives the ASCII value of the argument **S**. That is, the condition **:X = 'S** in combination with the previous **KEY** function checks to see whether the [S] key was depressed. If the [S] key was depressed, then the statement **SEND** is run.

The **SEND** statement sends a message to another turtle by leaving the message in a mailbox. The first number after the **SEND** is the address of the message. In the line we are analyzing the address is 2, so this message can be picked up from the mailbox only by turtle 2. The address can be an expression as well as a number. The second number after the **SEND** is the message. Here the message is the number 1; in general the message can be any number in the range covered by Super LOGO (–32768 to 32767) or an expression which gives a number in this range. To review,

```
SEND 2 1
```

leaves the message 1 in the mailbox for turtle 2. Because the [S] key is to move the runner (turtle 2), the message 1 must mean move. We'll see that in the procedure **RUNNER**.

Although we aren't going to use it in this example, there is a way to send a general message to the first turtle that picks up its mail. We just use the turtle address 255; then the next turtle that inquires will get the message. If we wanted to send an all points bulletin to all turtles, we could do so by setting a global variable (see Chapter 8).

The rest of **GETKEYS** is just more of the same. We check for each of the keys which control the runner and send a message to turtle 2 if one of them is depressed, and we do the same for the three keys which control turtle 3. Notice that the **WHILE 1=1** causes turtle 1 to continue to poll the keyboard forever. There are certain features of Super LOGO which make this part of the programming very simple. By assigning one turtle the task of watching the keyboard at all times we make sure that the two players have equal access to control; we are very unlikely to lose keystrokes while something else is happening, and provision for regular polling of the keyboard is handled automatically by the logic which handles multiple turtles.

Now let's turn our attention to **RUNNER**.

```
TO RUNNER :X
  PU SX :X
  SHAPE FFFFFFFFUBBBRRFD–
FFUBBBDBBB
  WHILE 1 = 1 (MAKE "X MAIL 1
    IF :X
      (IF :X = 1 (FD 8)
       ELSE (RT :X)
      )
  )
END
```

RUNNER sets a starting position for the runner, lifts the pen so that the runner leaves no tracks (which makes no difference in the chase, but keeps the screen clean), and draws a shape so that the runner will look different. We then enter another **WHILE 1=1**, which will run forever.

The runner turtle now checks its mailbox by using the **MAIL** function. The number following **MAIL** (the argument) is the number of the turtle that the runner turtle will accept mail from. Here turtle 2, the runner turtle, is asking for mail from turtle 1, the keyboard turtle. If there is no message, then **MAIL** returns the value **Ø**. The statement

```
IF :X
```

checks for the value of **:X**. If it is Ø, then the statements in parentheses are skipped. Since the parentheses enclose all the rest of the commands, a Ø causes the loop to start again. Thus the turtle just keeps checking its mail until it gets a message from turtle 1.

If we look back at **GETKEYS** we see that a message 1 meant to move. Therefore if **:X = 1** the runner is moved forward 8. If at this point the message is not 1, then it must be either 45 or 315. The runner is turned right by either amount (remember that **RT 315** is the same as **LT 45**). This completes the move, so the turtle goes back to checking its mail from turtle 1.

Before going further, look carefully at the arrangement of the two **IF** statements in **RUNNER**. Notice that the parentheses after the first **IF** enclose the second **IF** and the **ELSE**. This pairs the **ELSE** with the second **IF**. The meaning is: if **:X** is not **Ø** (the first **IF**), then do one or the other of the following; if **:X** is **1**, move forward — otherwise turn.

The **CHASER** procedure is similar to **RUNNER**, but it includes the test for a successful catch.

```
TO CHASER :X
  WHILE 1=1
    (HOME PU
    WHILE NEAR 2 > 12
      (MAKE "X MAIL 1
        IF :X
          (IF :X=1 (FD 16)
          ELSE (RT :X)
          )
      )
    SEND Ø 1
  )
END
```

CHASER includes nested **WHILE** statements. The first one starts the chaser and runs forever. The inner one runs until a capture is made. The definition of a capture is that the value returned by the **NEAR** function is 12 or less. The portion of the procedure controlled by the condition **NEAR 2 > 12** is identical to that in **RUNNER**. Remember that the **NEAR** function returns the total number of **X** and **Y** steps from the current turtle to the designated turtle, here to turtle 2. Thus the inner part of the procedure says to continue checking mail and making moves as long as the runner is more than 12 steps away.

If the runner is not more than 12 steps away, then turtle 3, the chaser, sends a message (1) to the scorekeeper (turtle Ø). Having sent the message, the chaser returns to the home position and the chase begins again.

Now we turn to the procedure for the scorekeeper.

```
TO SCOREKEEPER :S
  HT SX 2ØØ SY 18Ø
  WHILE 1=1
    (PRINT CHAR(32)#CHAR(32)
      PRINT :S
    WHILE MAIL 255 = Ø ()
    MAKE "S :S+1
    )
END
```

Again there are several new ideas in this procedure. The first steps are to hide the turtle and to position it to keep the scoreboard. We set the initial score to Ø by the call of the procedure and again use a **WHILE 1=1** to keep this turtle keeping score forever. The **PRINT** statement causes what follows to be printed on the screen at the current turtle position. The turtle is not moved. However, we want to print spaces to erase the old score, and LOGO uses the space to indicate the end of something. Therefore, to print spaces we must use the **CHAR** function. The **CHAR** function returns whatever character in the ASCII convention corresponds to the number in parentheses. **CHAR(32)** gives a space. To put two sets of characters together (here two spaces together) we use the concatenation operator **#**. Thus the combination

```
CHAR(32)#CHAR(32)
```

gives two spaces.

The **PRINT** statement also can be used to print the current value of a variable. **MAIL 255** is a special version of the **MAIL** function. **MAIL 255** will accept messages from all other turtles. Here we could use **MAIL 3** just as well, since turtle 3 is the only one sending messages to the scorekeeper. The following line:

```
WHILE MAIL 255 = Ø ()
```

continues to check until mail is received.

One useful characteristic of the **MAIL** function is that, like any decent mail system, it will collect messages. Thus if several messages have collected from one or more sources, the **MAIL** function will deliver the oldest undelivered message and keep the others for future reference. A **SEND 255** goes onto every turtle's list. That message disappears from all lists when one turtle accepts it.

Now that you have the whole set of procedures entered, you can try running the game. To start it, run **CATCHEM**. Just remember that this is an educational experience, not pure entertainment. You may discover that there is a flaw in the game. If the runner is caught close to home, then, because the chaser is returned to home after each successful catch, the runner is unable to escape and the score mounts. You could fix this by moving the chaser elsewhere if the runner is too close to home, or by just incrementing the x position of the chaser by some large number (say 1ØØ) after each catch.

One interesting variation of the game uses a turtle which obeys Newton's Laws. These so-called "DYNATURTLES" are set in motion and continue in motion in the same direction until disturbed. The two turning keys now change the direction of a thrust instead of changing the direction of the turtle directly. The third key gives the turtle a thrust or push in the current direction. The following version of **RUNNER** shows how this is done.

```
TO RUNNER :X :VX :VY
  PU SX :X
  SHAPE FFFFFFFFUBBBRRFD-
FFUBBBDBBB
  WHILE 1=1
    (MAKE "X MAIL 1
    IF :X = 1 (
     MAKE "VX :VX+SIN HEADING ME
     MAKE "VY :VY+COS HEADING ME)
    IF :X = 45 (RT 45)
    IF :X = 315 (LT 45)
    SX XCOR + :VX
    SY YCOR + :VY
  )
END
```

Here **:X = 1** means there should be a move in the current direction. We move the turtle by a series of **SX** and **SY** commands; the increments (**:VX** and **:VY**) are adjusted by use of the standard trigonometric functions **SIN** and **COS**.

To use this modified version of **CATCHEM** we have to make the above changes in **CHASER** as well as in **RUNNER**, and we must set some initial values of **:VX** and **:VY** for both players. The initial values are set in the procedure **CATCHEM** by adding numbers onto the procedure calls. You might start with values about 5, and it makes a better game if the chaser is a bit faster than the runner.

The second game is called **REBOUND**. It makes use of the game controllers. The object of the game is to bounce a ball off two paddles onto a target. Here we'll need a few more turtles. We first assign four tasks: reading of the two controllers, a scorekeeper, and a trigger to start the whole thing off.

```
TO REBOUND
  CLEAR HT
  HATCH 2 PADDLE1
  HATCH 3 PADDLE2
  HATCH 6 SCOREKEEP
  TRIGGER
END
```

Let's look at the paddle controls first. The paddles can be turned to direct the ball to the target.

```
TO PADDLE1
  HT SX 60 SY 180
  TURN 0
END

TO TURN :P :X
  WHILE 1=1
    (MAKE "X PADDLE :P/2
      LINE 3
      SH 45 + 3*:X
      LINE 0
      WHILE PADDLE :P/2 = :X ()
    )
END

TO LINE :COLOR
  PC :COLOR
  FD 15 BK 15 BK 15 FD 15
END
```

PADDLE1 establishes the position of the first paddle on the screen. It calls **TURN** which actually reads the game controller. The new idea in **TURN** is the use of the **PADDLE** function. The **PADDLE** function returns a number between 0 and 63 for the designated input; the number depends on the position of the controller handle. The inputs are 0 and 1 for the horizontal and vertical positions of the left game controller and 2 and 3 for the horizontal and vertical positions of the right game controller. Thus **PADDLE1**, by the instruction **TURN 0**, tells the procedure **TURN** to read the horizontal position of the left controller (left refers to the position of the plug on the rear of the Color Computer). Because the instruction is

```
MAKE "X PADDLE :P/2
```

the variable **"X** holds a number between 0 and 31. This division of the controller reading by 2 reduces the sensitivity of the display and speeds response. Notice that, after the first pass through **TURN**, the procedure looks for a change in the controller setting. It stays in the loop

```
WHILE PADDLE :P/2 = :X ()
```

until there is a change. When there is a change, it runs through the outer loop which updates **:X**, erases the old paddle (**LINE 3**), computes a new heading (**SH 45 + 3*:X**), and draws a new paddle (**LINE Ø**). Remember that **:X** can be between Ø and 31, so the heading for the paddle can be between **45** and **45 + 3*(31) = 136**. The procedure **LINE** actually draws the paddles and erases them. The **BK** is broken into two steps so that it exactly duplicates the **FD** steps; this insures a successful erase.

The second paddle is controlled by the second controller. We can use **TURN** and **LINE** again.

```
TO PADDLE2
  HT SX 18Ø SY 12
  TURN 2
END
```

Now we have to create the ball and the target. **TRIGGER** starts a new round.

```
TO TRIGGER
  HT
  HATCH 4 BALL
  VANISH
END
```

The ball should come from a randomly selected point towards the first paddle. The easiest way to do that is to create the ball turtle at the first paddle and to move it (invisibly) in the randomly selected direction. These two tasks will be carried out by the procedures **LAUNCHBALL** and **STARTSPOT**.

```
TO BALL
  LAUNCHBALL
  WHILE MAIL 5 = Ø
    (STARTSPOT
    HATCH 5 TARGET
    REPEAT 45
      (FD 1Ø
      IF NEAR 2<2Ø
        (FD 1Ø LT (HEADING 4 -
          HEADING 2 + 18Ø)*2 FD 35)
    IF NEAR 3<25
      (FD 1Ø
        LT (HEADING 4 -
        HEADING 3)*2 FD 45)
    )
  )
  TRIGGER
END
```

At the same time we create the target at a randomly selected position (**HATCH 5 TARGET**). The **REPEAT** loop actually moves the ball. If the ball is close to the first paddle (turtle 2), the heading of the ball is changed

```
LT (HEADING 4 – HEADING 2 +180)*2
```

There is a similar change when the ball is close to the second paddle (turtle 3). Notice that when the ball has moved the maximum distance it triggers a new ball before disappearing.

```
TO LAUNCHBALL
  HT PU
PAT
................
................
................
................
......XXXX......
.....XXXXXX.....
....XXXXXXXX....
....XXXXXXXX....
....XXXXXXXX....
....XXXXXXXX....
.....XXXXXX.....
......XXXX......
................
................
................
................
  MAKE "Y RANDOM 60 + 160
END

TO STARTSPOT
  HT SH :Y SX 60 SY 180
  REPEAT 6 (FD 10)
  WHILE XLOC 4>7 & YLOC 4>7
    (FD 10)
  RT 180 ST FD 10
END
```

LAUNCHBALL creates an appropriate shape for the ball and effectively picks a random starting point by picking the heading. **STARTSPOT** hides the ball turtle, locates it at the first paddle, moves it until it reaches the edge of the screen, and finally turns it around and makes it visible.

TARGET does the scoring. First it picks a random position and creates a target shape. Then it watches for a close approach of the ball from below (if the ball approaches from above, it has not bounced off the second paddle). If the ball (turtle 4) comes close enough, then a message is sent to the scorekeeper and to the ball.

```
TO TARGET
  SH Ø
  HT SX RANDOM 1ØØ + 135
  SY RANDOM 4Ø + 12Ø
  SHAPE URRFFFFFFFFFFLLLDFFFF–
FLFFFFFFFFFLFFFFF
  ST
  REPEAT 1ØØ
    (IF NEAR 4<15 &
    ABS (HEADING 4 – 18Ø)>9Ø
      (SEND 6 1 SEND 4 1)
    )
  VANISH
END
```

The **SCOREKEEP** procedure is essentially the same as before.

```
TO SCOREKEEP :SCORE
  HT SX 2ØØ SY 18Ø
  WHILE 1=1
    (PRINT CHAR(32)#CHAR(32)
      PRINT :SCORE
    WHILE MAIL 5=Ø ()
    MAKE "SCORE :SCORE+1
    COLORSET 1 COLORSET Ø
  )
END
```

These two examples should help you to implement your own ideas for more complex games.

19. WORD AND LIST OPERATIONS

The origin of LOGO can be traced to the field of artifical intelligence. The main goal of computer scientists in the field of artifical intelligence is to design programs which will make a computer appear to have what a human being would describe as intelligence. This is an extremely difficult problem. One part of the problem is to make the computer understand English (or German, or Japanese, or whatever). The LISP computer language, on which LOGO was based, was designed for the manipulation of words and sentences for this purpose. Super LOGO includes operators and functions, drawn from LISP, for the manipulation of words as well as numbers. In this chapter we'll learn about the primitive functions.

The arguments of the functions are called words and lists. A word in LOGO is similar to, but somewhat more general than, a word in English. In LOGO a word is a series of characters ended by a space. In English a word includes only the 26 letters, and not all combinations and sequences of the letters are allowed as words. In LOGO almost all characters which can be entered from the keyboard can be part of a word, and there are no rules which limit the combinations. The following are all legal words in LOGO:

```
"FEW "MANY "DOG "ALPHABET "XXYYZZ "RT$%;9
```

The leading quotation mark (") indicates that these are words. The quotation mark is necessary to distinguish a word from a procedure name, but the quotation mark is not part of the word. We have already used words as variable names in **MAKE** commands.

```
MAKE "X 20
```

This shows that a word can have a value attached to it. We will return to this point later.

These LOGO words already give some indication of the difficulty of dealing with words via the computer. It is relatively easy to program the computer to distinguish meaningful numbers and mathematical expressions from nonsense, but it is not possible to program a microcomputer to distinguish real words from nonsense combinations of characters. Therefore, in dealing with words we are going to have to be more selective in our projects, and we are going to have to supply the computer with restricted, selected words to operate on if we are to avoid nonsense. Computers do not understand enough about English for us to use a discovery learning approach to the English language.

In LOGO we call a series of words a list. A sentence is a list, but a series of numbers is also a list. Elements of a list (words) are separated by spaces. Lists are indicated by enclosing them in square brackets. (The SHIFT 5, SHIFT 8 sequence produces the left bracket and the SHIFT 5, SHIFT 9 sequence produces the right bracket.) Several lists are shown below:

```
[ONE TWO THREE FOUR]
[1 2 3 4]
[THIS SENTENCE IS A LIST]
[OVER 556 ELEPHANT RALPH LJ;X]
[REMOVE]
[]
```

The last two examples show that a list can contain as little as a single word and that a list can be empty.

There are two limitations which govern the words and lists in Super LOGO. The maximum length of a word is 13 characters. Lists consists of words, not of other lists.

We can try out the simple word and list functions without entering procedures. Get into RUN mode. The basic command to check the result of one of these functions is the **PRINT** command. The **PRINT** command prints at the current position of the turtle without moving the turtle. This can get confusing if we do several **PRINT** commands in sequence. By entering

```
FULLTEXT
```

we convert to a pure text screen. The text window, which usually occupies only the bottom four lines of the screen, then occupies the whole screen, and the **PRINT** command prints on the next available line. So enter **FULLTEXT**.

As a first step, enter the command

```
PRINT "ABSOLUTE
ABSOLUTE
```

and notice that the word **ABSOLUTE** is printed on the next line. Next try the **FIRST** function.

```
PRINT FIRST "ABSOLUTE
A
```

Because the object of the function is a word, **FIRST** returns the first letter of the word as a new word. Try

```
PRINT FIRST FIRST "ABSOLUTE
A
```

The **BUTFIRST** function produces what its name implies. Try

```
PRINT BUTFIRST "ABSOLUTE
BSOLUTE
```

The abbreviation for **BUTFIRST** is **BF**. Combinations of the two functions can be used to select any letter in a word. The following combination will select the third letter.

```
PRINT FIRST BF BF "ABSOLUTE
S
```

Notice that the order of operation is from right to left. Compare the result with the sequence

```
PRINT BF BF FIRST "ABSOLUTE
—
```

Here the function **FIRST** is done first, returning the letter **A**. The **BF** function then returns the empty word, and the end result is a blank.

We have a similar set of functions for working on the end of a word. Try

```
PRINT LAST "ABSOLUTE
E
```

and

```
PRINT BUTLAST "ABSOLUTE
ABSOLUT
```

and

```
PRINT LAST BL BL "ABSOLUTE
U
```

So we have a set of functions for pulling words apart. You might have guessed that we have a function for building words too. The **WORD** function combines two words into a new word. Try

```
PRINT WORD "SNOW "BALL
SNOWBALL
```

and

```
PRINT WORD WORD "SNOW "BALL "ED
SNOWBALLED
```

These are the primitive functions for words. Remember that a word can be as short as a single letter or even an empty word.

> Note: In RUN mode, a list processing command cannot be longer than one line.

Most of these same functions can be used on lists. Try

```
PRINT FIRST [ONE TWO THREE]
ONE
```

Notice that the result of the **FIRST** function, **ONE**, is the first word from the list. Therefore, the command

```
PRINT FIRST FIRST [ONE TWO]
O
```

will return the word **O**. Try the other commands as well.

```
PRINT BUTFIRST [A B C D]
B C D
```

```
PRINT LAST [A B C D]
D
```

```
PRINT BUTLAST [A B C D]
A B C
```

Obviously combinations can be used, as they were used with words, to pick out any word from a list.

Words are combined into lists by the **SENTENCE** function. Try

PRINT SENTENCE "SNOW "BALL
SNOW BALL

Contrast the result here with the result using the **WORD** function (**SNOWBALL**). **SENTENCE** produces a list; **WORD** produces a word.

Lists can be combined, and words and lists can be combined. We'll use the abbreviation **SE** for the **SENTENCE** function.

PRINT SE [SLOW BOAT] [TO CHINA]
SLOW BOAT TO CHINA

The functions **FPUT** and **LPUT** also can be used to combine a word and a list to form a new list, and the function **LIST** also can be used to form a new list from two words. They are included for compatibility with other implementations of LOGO, but we'll stick with **SENTENCE** in our examples (see Appendix I for more information).

This completes the introduction of the primitive word and list functions. Notice that these are functions which produce a result which is not visible on the screen, unless printed. This is different than turtle graphic primitives which produce visible results automatically. To make full use of these new primitives we'll have to learn to pass the results between subprocedures and procedures.

20. COMMUNICATION BETWEEN PROCEDURES

Thus far all our procedures have produced graphics. Graphic procedures produce figures or patterns on the screen; the result of such procedures is a track and a final turtle position and heading on the screen. Although numbers were sent from a higher level procedure to a lower level subprocedure (for example, by commands like **BOX 50**), no information was sent back from the subprocedure to the higher level procedure. But in other applications information must be sent both ways. In this chapter we will learn to return results from subprocedures to higher level procedures. In the process we will learn a bit more about variables.

The primitive commands thus far can be separated into two classes. In one class we have the commands which can be executed on their own, without any additional information. The following commands require no additional information:

PU PD ST HT CLEAR DRAW SPLITSCREEN

Other commands require additional information before they can be executed:

FD BK RT LT PC SX SH

Each command in the second set needs a number to become a functional command; for example,

FD 30 BK 40 RT 90 LT :ANGLE PC 1 etc.

Most functions, for example

MAIL NEAR RANDOM FIRST LAST

also need more information (usually called an "argument") to operate.

MAIL 2 NEAR 3 RANDOM 5 FIRST :L etc.

Notice that the functions not only require an argument, but that they also produce a result that could be used as an argument for another function.

```
MAIL FIRST :L          NEAR RANDOM 5
RANDOM MAIL 2          NEAR FIRST :L
```

When we need an operation which LOGO does not include as a primitive, we write a subprocedure which carries out the operation and which we can use like a new primitive. For instance, in Chapter 5 we wrote the subprocedure **BOX**, and then we used the word **BOX** just as we used the primitive operations **FD** and **RT**. But how do we write subprocedures which supply new functions, which produce results and which can be used in other procedures just as we use supplied functions like **RANDOM**?

We'll illustrate the process by writing a subprocedure which duplicates the action of an existing function **ABS**. The **ABS** function returns the absolute value of the argument. That is, if the argument is positive, then no change is made, but if the argument is negative, its sign is changed. The following procedure carries out exactly those actions.

```
TO ABSOLUTEVALUE :NUMBER
  IF :NUMBER < 0
  (MAKE "NUMBER :NUMBER * -1)
END
```

This procedure does the conversion to a positive number, but in its present form it cannot be used as a new function. The problem is that the subprocedure **ABSOLUTEVALUE** does not return the value to the higher level once it has made it positive.

The command to return a value is the **OUTPUT** command. We can make the subprocedure operate as a function by adding an **OUTPUT** command at the end.

```
TO ABSOLUTEVALUE :NUMBER
  IF :NUMBER < 0
  (MAKE "NUMBER :NUMBER * -1)
  OUTPUT :NUMBER
END
```

On the receiving end (the higher level procedure) the computer must be instructed to pick up the result. For example, the command

```
ABSOLUTEVALUE -10 FD RESULT
```

gives the desired action. The **OUTPUT** command in **ABSOLUTEVALUE** transmits the new value, and **RESULT** picks it up for further use.

Note: Other implementations of LOGO do not use the **RESULT** operation because they cannot handle multiple turtles. Single-tasking implementations allow the syntax

```
FD ABSOLUTEVALUE -10
```

but Super LOGO does not.

For those experimenting with examples from books written for other implementations, the translation process is:

1. Replace the subprocedure name (and parameters, if any) with the command **RESULT.**
2. Insert the subprocedure name (and parameters, if any) before the command which will use the result.

Given the availability of the **ABS** function, the sample procedure is not really useful. However, there are a number of other mathematical functions which might be of use. For example, Super LOGO does not supply a function for raising a number to a power. The following procedure supplies it.

```
TO EXP :NUMBER :POWER :X
  MAKE "X :NUMBER
  REPEAT :POWER-1 (
  MAKE "NUMBER :NUMBER * :X)
  OUTPUT :NUMBER
END
```

Try this out with the commands

```
FT
EXP 2 3 PRINT RESULT
8

FT
EXP 6 3 PRINT RESULT
216
```

A square root function is even possible as long as you allow for the limited accuracy of the division operation. The following procedure makes use of a common process of successive approximations for the square root of a number.

```
TO ROOT :N :G
  IF :N<Ø (PRINT [NEGATIVE] STOP)
  MAKE "G :N/2
  WHILE ABS(:G*:G-:N)>1+:N/1Ø
    (MAKE "G (:G + :N/:G)*Ø.5)
  OUTPUT :G
END
```

Try this out with the following (be sure to hide the turtle first).

> Note: Clear the screen between answers, or a shorter answer will retain the last digit of a longer answer.

```
ROOT 1Ø PRINT RESULT
3.41

ROOT 1ØØ PRINT RESULT
1Ø.Ø5

ROOT 1ØØØØ PRINT RESULT
1Ø1.48
```

While the accuracy is limited by division, it is sufficient for calculating the distance between points on the screen. Notice that we include a trap for negative numbers. The accuracy can be increased within a more limited range of numbers by adjusting the condition on the **WHILE**, but if you try for too much, you will end up in an endless search.

The next step is to combine some of these procedures. For example, we might want to compute the square root of 5 raised to the 3rd power. If you have entered the procedures **EXP** and **ROOT**, the following line will print the correct result.

```
EXP 5 3 ROOT RESULT PRINT RESULT
11.46
```

Notice that the line reads exactly in the order that the computer does the operations. To make sure you have this straight we'll give another example. If we want to print 5 raised to the 3rd power and then multiplied by 6, we enter

```
CLEAR FT
EXP 5 3 PRINT 6*RESULT
750
```

In these examples we've used the **MAKE** command several times. Look at the example

```
MAKE "G :N/2
```

In the last chapter we learned that the notation **"G** indicates the word **G**. Variable names are words; any word can be a variable name. The notation **:N** designates the contents of the variable with the name **N**. The contents of a variable are completely distinct from the name of the variable. The name must be a word, but the contents may be a number, a word, or a list.

The next procedure takes advantage of this difference.

```
TO PHONEBOOK
  MAKE "DENNIS "555-3958
  MAKE "JOE "555-9935
  MAKE "CHRIS "555-9965
  MAKE "ELAINE "555-7563
END
```

First run this procedure:

```
FULLTEXT PHONEBOOK
```

then enter commands like

```
PRINT "CHRIS PRINT :CHRIS
```

The computer will respond with the two lines

CHRIS
555—9965

The first command, **PRINT "CHRIS**, tells the computer to print the literal word **CHRIS**. The second command, **PRINT :CHRIS**, tells the computer to print the current value of the variable named **CHRIS**, which is the phone number. The command

MAKE "CHRIS "555—9965

in the procedure **PHONEBOOK** loaded the word **555—9965** into the memory space named **CHRIS**. Remember that **MAKE** commands will always have a variable name (a word) followed by a value (a word, a number, a list, or an expression which evaluates to a word, number, or list).

In the examples in this chapter the values output and used as results have been numbers, mainly because this minimizes the confusion between the name of the variable and the value it holds. In following chapters we will see examples where the values can be words and lists as well.

21. INTERACTIVE PROCEDURES

Many list processing procedures (procedures that manipulate words) will be more interesting if they are interactive, that is, if the person running the procedure can feed information into the procedure from the keyboard while the procedure is running. We've already seen one way to do that, the **KEY** function. The **KEY** function is most suitable for writing game procedures because it checks the keyboard and then continues immediately. Now we want a way to tell the computer to wait until it receives appropriate information from the keyboard.

The **READCHAR** function (abbreviated **RC**) accepts a single character from the keyboard. This character becomes a word. The character does not appear on the screen when the key is pressed; we must print it if we want it to appear. We'll begin with a very simple example, a procedure which asks whether a statement is true or false.

```
TO QUES
  PRINT [COWS CAN FLY]
  PRINT [TYPE T FOR TRUE]
  PRINT [TYPE F FOR FALSE]
  IF RC = "F (PRINT "RIGHT)
  ELSE (PRINT "WRONG)
END
```

This procedure starts by printing three lists which give the statement and the instructions. The **RC** (**READCHAR**) function waits until a key is pressed and makes the key pressed into a word. If that word is the same as the word **F** (**"F**), that is, if the **F** key was pressed, then the command is to print the word **RIGHT**; otherwise print the word **WRONG**. When you enter and run **QUES** (remember **FT QUES**) you will notice that the letter selected does not appear on the screen. If we want to show the letter, then we must print it. There are several ways this can be done.

```
TO QUES1
  FT
  PRINT [COWS CAN FLY]
  PRINT [TYPE T FOR TRUE]
  PRINT [TYPE F FOR FALSE]
  IF RC = "F (PRINT SE "F "RIGHT)
  ELSE (PRINT SE "T "WRONG)
END

TO QUES2 :ANSWER
  FT
  PRINT [COWS CAN FLY]
  PRINT [TYPE T FOR TRUE]
  PRINT [TYPE F FOR FALSE]
  MAKE "ANSWER RC PRINT :ANSWER
  IF :ANSWER = "F (PRINT "RIGHT)
  IF :ANSWER = "T (PRINT "WRONG)
END
```

```
TO QUES3
  FT
  PRINT [COWS CAN FLY]
  PRINT [TYPE T FOR TRUE]
  PRINT [TYPE F FOR FALSE]
  TEST RC = "F
  IFTRUE (PRINT SE "F "RIGHT)
  IFFALSE (PRINT SE "T "WRONG)
END
```

In the first and third examples we have combined the letter and the feedback into a sentence. In the second example we have transferred the letter into the variable named **ANSWER** so that we can do several things with it (print it and use it in two conditions). The third example also illustrates a useful alternative to the **IF...THEN...ELSE** control statements. The **TEST** statement tests whether the expression is true or false and saves that information for future reference. The list of commands following the **IFTRUE** will be executed only if the result of the **TEST** statement is true.

The list of commands following the **IFFALSE** statement will be executed only if the result of the **TEST** statement is false. The **TEST...IFTRUE...IFFALSE** combination is most useful when the test and the actions which are to depend on that test are somewhat separated within a procedure.

Next we move to an arithmetic drill. The procedures will turn out to be a bit more complex than you might expect, but they form a good illustration of some important differences between numbers and words. To keep things simple we want to present some single digit addition questions. Try

```
TO ADD :SUM
  FT
  PRINT [2 + 3 = ]
  MAKE "SUM RC PRINT :SUM
  IF :SUM = 5 (PRINT "CORRECT)
  ELSE (PRINT [NOT CORRECT])
END

2 + 3 =
6
NOT CORRECT
```

This works, but we can do better. It would be nice to use the **RANDOM** function to make the computer make up questions, and it would be nice to present the question as it might appear in a book:

```
  2
+3
```

with the answer in line. We'll make the presentation of the problem the job of the subprocedure **PICK**. Then our main procedure is

```
TO DRILL :ANSWER :GUESS
  FT
  REPEAT 5 (
    PICK MAKE "ANSWER RESULT
    MAKE "GUESS RC
    PRINT SE CHAR 32 :GUESS
    IF :GUESS = :ANSWER
      (PRINT "GOOD)
    ELSE (PRINT [NOT RIGHT])
  )
END
```

We've decided to give 5 problems. **PICK** must pick the two addends, print them on the screen, and output the sum back as the result which is assigned to the variable **ANSWER**.

```
TO PICK :A1 :A2
  MAKE "A1 RANDOM 5
  MAKE "A2 RANDOM 6
  PRINT CHAR 32#CHAR (48 + :A1)
  PRINT LPUT CHAR (48 + :A2) [+]
  OUTPUT :A1 + :A2
END
```

The first two lines are familar from earlier work with turtle graphics; the **PRINT** lines contain new ideas. We want to print the value in **A1**, but not in the first column (because of a "+" sign with **:A2**). Therefore we want to make a list with a space as the first word and the digit in **A1** as the second word. We can't just enter a space from the keyboard because a space means the end of whatever precedes it, so we use the **CHAR** function to generate a space character within the list. **CHAR 32** gives a space. Lists are lists of words, and the value in **A1** is a number. It must be converted into a word before it can be used in a list. Again we use the **CHAR** function. The digit **Ø** is **CHAR 48**, and the remaining digits follow in order, so **CHAR (48 + :A1)** gives a one digit word. The concatenation operator (**#**) combines the two words (space and digit) into a list. The next **PRINT** command uses the same technique to convert the value into a word, but it uses a different function to combine the two words (+ and digit) into a list. The **LPUT** function expects a word and then a list, and it adds the word to the end of the list.

Now we return to examination of the main procedure. The response from the keyboard is assigned to the variable **GUESS**. Printing the value in **GUESS** will print it in the first column of the screen. This time we indent by making a list from a space (**CHAR 32** again), and the value in **:GUESS**, with the **SE** function. The combination here is simpler than in **PICK** because the value in **GUESS** is a word (the **RC** function always returns a word). The rest of the procedure is straightforward.

As we shall see in the next procedure, we can use the **READLIST** (abbreviated **RL**) function when we want more than a single character from the keyboard. As the name of the function indicates, the result of this operation is a list, even if the list is only a single word long. The input from the keyboard is treated as a list which is ended when the user presses the ENTER key. There is no formal limit to the length of the list other than the capacity of memory. Note that a list created by a **MAKE** instruction within a procedure must fit on a single line, but a list entered from the keyboard is not subject to this limitation. While entering a list you can backspace to correct typing errors only within a word; once a space has been entered, the preceding word cannot be changed. The **REQUEST** function (abbreviated **RQ**) is identical to the **READLIST** function. The following procedure gives a simple example of the use of **RL**.

```
TO GREET :NAME
  FT
  PRINT [WHAT IS YOUR NAME?]
  MAKE "NAME RL
  PRINT SE [WELCOME,] :NAME
END
```

```
WHAT IS YOUR NAME?
RALPH
WELCOME, RALPH
```

Let's make this a bit more elaborate.

```
TO GREET2 :NAME
  FT
  PRINT [WHAT IS YOUR NAME?]
  MAKE "NAME WORD FIRST RL ",
  PRINT SE SE [WELL,] :NAME
  [GLAD TO MEET YOU.]
END
```

```
WHAT IS YOUR NAME?
VALERIE
WELL, VALERIE, GLAD TO MEET YOU.
```

Here we want to use the name within a sentence. Notice the double use of the **SE** function to combine the three pieces. Also notice the command

```
MAKE "NAME WORD FIRST RL ",
```

This takes the name from the keyboard and adds a comma to it. **FIRST** is needed because **RL** produces a list; **FIRST** takes the first (and only) word from that list and treats it as a word. If we tried to add a comma to the list, for example with the **SE** function, there would be a space between the name and the comma. If we omit the **FIRST** function, we get an error message because we can't make a word from a list and a character.

We have now seen all the functions which can be used to make procedures interactive. We'll end this chapter with a brief summary.

KEY — reads one key from keyboard as the ASCII number if it is depressed at the moment **KEY** is executed

RC — waits until one key at the keyboard is depressed and reads it as a word

RL — reads a list, terminated by **ENTER** from the keyboard
or **RQ**

PADDLE — reads the position of one of the game paddles as a number between **Ø** and **63**

22. PLAYING WITH WORDS AND SENTENCES

One obvious application of the word and list functions is to generate and alter English words and sentences. We quickly discover that English is very complex and that most projects are far beyond the capabilities of microcomputers. In fact, programming computers to understand natural language is an active research area on the largest current machines. The experimental approach which is so effective in turtle graphics is not practical with sentences; we do not discover much about English by trial and error with words and lists. In this chapter we will illustrate some simple language procedures, and in the next chapter we will give some examples of list operations which do not attempt to combine words as sentences.

We'll begin with some procedures which will be generally useful. Often we want to pick a word from a list, either a designated word or one selected randomly. The following procedure, similar to ones which appear in most LOGO books, does it recursively.

```
TO PICK1 :N :L
  IF :N=1 (OUTPUT FIRST :L)
  PICK1 (:N-1) (BF :L)
  OUTPUT RESULT
END
```

The approach is to remove words from the front of the list until word **N** is reached and then to output that word. Notice that the **OUTPUT** function stops a procedure and returns to the next level up.

To test this procedure enter

```
FT
PICK1 3 [A B C D EF GHI]
PRINT RESULT
```

and try it with other numbers and lists.

While **PICK1** works, it is very wasteful of memory. The reason is that the computer makes another copy of the list on every recursive call. With larger lists this consumes too much space. The following procedure uses the same idea but in a **REPEAT** instead of by recursion.

```
TO PICK :N :L
  REPEAT :N-1 (
  MAKE "L BF :L )
  OUTPUT FIRST :L
END
```

To compare the two we write a test procedure so that **RL** can be used to accept a list longer than a single line.

```
TO PICKOUT :N :L
  FT MAKE "L RL
  MAKE "N FIRST RL
  PICK :N :L PRINT RESULT
END
```

```
PICKOUT
A B C D E F G H I J K L M N O P
Q R S T U V W X Y Z

10
J
```

By experimentation you will find that the first procedure (**PICK1**) runs out of memory far sooner than the second (**PICK**). Why? Because the amount of memory used by **PICK1** depends both on the length of the list and on the number of items which must be peeled off the list (the number of recursive calls).

The amount of memory used by **PICK** depends only on the length of the list. The next procedure counts the number of words in a list.

```
TO LENGTH :LIST :COUNT
  MAKE "COUNT 0
  WHILE :LIST <> []
  (MAKE "COUNT :COUNT + 1
  MAKE "LIST BL :LIST
  )
  OUTPUT :COUNT
END
```

The **WHILE** segment removes words from the list and increments the count until the list is empty (an empty list is indicated by the symbols **[]**).

LENGTH can be combined with **PICK** to select a random word from a list.

```
TO PICKRANDOM :L :X
  LENGTH :L
  PICK (1 + RANDOM RESULT) :L
  OUTPUT RESULT
END
```

For example:

```
FT
PICKRANDOM [A B C D E F G H I]
PRINT RESULT
H
```

When we call **LENGTH** we only pass it the list, but **LENGTH** expects two variables. Remember that omitted variables are assumed to be zero. **LENGTH** returns a result which is subsequently used as the argument for **RANDOM**. A result does not have to be picked up at once; it stays available until it is picked up or until it is overwritten by another **OUTPUT** from another subprocedure called by this procedure.

The next procedure tests to see if an item is a member of a list. Here we just output one of the words **TRUE** or **FALSE** to indicate the result. Instead we could count and output the position of the word in the list (or zero if it isn't there).

```
TO MEMBER :WORD :LIST
  WHILE :LIST <> []
    (IF :WORD = FIRST :LIST
      (OUTPUT "TRUE)
    MAKE "LIST BF :LIST)
  OUTPUT "FALSE
END
```

To test this enter

```
FT
MEMBER "IN [AT ON IN TO BY]
PRINT RESULT
TRUE
```

Enter other words and lists to check the **FALSE** output.

With these useful subprocedures we can begin more interesting projects. Conversion of a sentence into pig latin illustrates some techniques. There are various sets of rules for pig latin; we'll use the one which requires movement of all leading consonants to the end of the word and the addition of **AY** to every word. First we'll write a procedure which converts a single word.

```
TO PIG :W
  MEMBER (FIRST :W) [A E I O U]
  IF RESULT (OUTPUT WORD :W "AY)
  PIG (WORD BF :W FIRST :W)
  OUTPUT RESULT
END
```

We use **MEMBER** to check if the first letter of a word is a vowel. We pick the first letter (**FIRST :W**) and check if it is in the list of vowels. Remember that **MEMBER** returns **TRUE** or **FALSE**. These words can be used as conditions in expressions; for example, **IF RESULT (...)**. The commands in the parentheses will be executed if **MEMBER** outputs **TRUE**, but they will be skipped if **MEMBER** outputs **FALSE**. Again remember that **OUTPUT** is like a **STOP**; it returns to the next higher level. The final **OUTPUT RESULT** takes the result from the lower level and passes it on.

You should test this procedure before proceeding. Try something like

```
FT PIG "TRANSLATE PRINT RESULT
ANSLATETRAY
```

Now we need a procedure which will work its way through a list, picking off one word at a time, passing it to **PIG** for transformation, and replacing the original in the list with the new word. This is useful in other contexts; we are developing a model for transforming every member of a list into something related. The key here is recursion.

```
TO METAMORPHIZE :S :H
  IF :S = [] (OUTPUT [])
  PIG FIRST :S MAKE "H RESULT
  METAMORPHIZE (BF :S)
  OUTPUT SE :H RESULT
END
```

This is a bit more complex than the recursion we've used with words and lists thus far, so let's look at it in some detail. For the moment ignore the check for the empty list. At every level the first word on the list is passed to **PIG**, and the metamorphized result is stored in **H**. Then the list minus the first word is passed down recursively. Therefore, when the list is empty every higher level has one metamorphized word in **H**, and they are in order from first (highest level) to last (lowest level). When the lowest level is reached (the empty list) an empty list is output. The next-to-lowest level combines the last word with this into a sentence. This is in turn passed back up by the **OUTPUT** function. As the computer works back up through the levels, it combines the metamorphized word stored at that level with what is output from below to form a new sentence, and it sends that to the next higher level.

Of course we must try this out. Enter

```
FT
METAMORPHIZE [A TRIAL SENTENCE]
```

and follow that with

```
PRINT RESULT
AAY IALTRAY ENTENCESAY
```

The following may be obvious. One could replace **PIG** with any other procedure which implemented a transformation rule. We have here a way to metamorphize any list to another list, so long as that metamorphosis is governed by a complete set of rules.

What about going the other way; could we write a set of procedures to transform pig latin into English? There is no problem removing the **AY** from the end of each word. But then what? There is no set of simple rules which tells us whether or not to move a particular consonant back to the start.

Something which is trivial for a child playing with pig latin is extremely difficult to program. The decision is one which requires vast knowledge of English, not a few rules. As such, it cannot be programmed. This illustrates some of the difficulty of computerized language generation.

Another popular exercise is the generation of random sentences. As long as we give the computer lists which are properly divided into nouns, verbs, etc., the computer can generate sentences which have the correct form. In general they will be nonsensical, which is one reason that children find them amusing. The following procedure could be extended to form more complex sentences.

```
TO MADLIB :N
  FT
  MAKE "NOUNS [SLUGS EELS RATS]
  MAKE "NOUNS :NOUNS#[ELVES MOMS]
  MAKE "VERBS [CRAWL SWIM BITE]
  MAKE "VERBS :VERBS#[DIVE LOVE]
  PICKRANDOM :NOUNS
  MAKE :N RESULT
  PICKRANDOM :VERBS
  PRINT SE :N RESULT
  REPEAT 2500 ()
  FT
  MADLIB
END
```

You can execute this simply by entering **MADLIB** in RUN mode.

Extension of this to three word sentences of the form: noun-verb-noun could be used to introduce the idea of transitive and intransitive verbs.

A number of other projects are possible, but they quickly become so complex as to limit their value. For example, a procedure to turn a noun into its plural form might be useful. However, there are so many exceptions to the rules that its utility is limited. Instead of continuing to generate sentences we will turn to examples where the words on lists are unrelated or where the relations between the words are simpler than the relations between English words in sentences.

Note: Before continuing or shutting off the computer, you may want to save the procedures in Chapter 22 to disk. A later chapter will use some of these procedures.

23. GENERATING AND SORTING LISTS

In this chapter we give examples of manipulation of lists of numbers and words. The relationships between the words in these lists will be simpler than the relationships between words in English sentences.

It is ironic that some of the best examples of list processing are mathematical, for list processing was designed to handle sentences. A series is a list of numbers; later members of the series are derived from the earlier members by straightforward mathematical operations. One interesting example is the Fibonacci series. It is easy to generate because each new member of the series is the sum of the previous two members. And it is an accurate representation of a surprising number of systems found in nature. The interested reader should look at *Discovering Apple LOGO; An Invitation to the Art and Pattern of Nature* by David Thornburg (Addison-Wesley Publishing Co., 1983) and the references therein. Here we'll settle for writing procedures to generate the series.

We need a subprocedure to convert a number to a word. In Super LOGO the computer recognizes when a word needs to be converted to a number by context (one cannot add the characters one zero to another number, but one can add the number ten). However, the computer does not make the conversion in the other direction automatically. In true LOGO fashion, when we discover that we need a new function, we write a subprocedure to provide it.

```
TO NTOWORD :NUM :W :W1
  IF :NUM = Ø (OUTPUT "Ø)
  MAKE "W "
  WHILE :NUM > Ø (
    MAKE "W1 :NUM
    MAKE "NUM INT(:NUM/1Ø)
    MAKE "W WORD
    CHAR (:W1 - :NUM*1Ø + 48) :W
  )
  OUTPUT :W
END
```

This subprocedure uses the **INT** function to make the result of division an integer. The difference between a number and ten times the integer portion of the number divided by 1Ø is the units digit.

```
:W1 - :NUM * 1Ø + units digit
```

The **CHAR** function uses a number as its argument and returns a one character word which is the character which corresponds to that number in the ASCII sequence. Because zero is character 48, and because the digits are assigned in order in the ASCII sequence, the above **CHAR** function returns the digit selected as a word. The subprocedure repeatedly divides by 1Ø to move digits to the right, takes the integer portion, converts each digit in turn, and combines it with the previous digits to form the complete word. The value Ø is treated as a special case.

Now the program to generate the Fibonacci series is quite simple.

```
TO FIBONACCI :L :T :N
  FT PRINT :L
  REPEAT :T (
    MAKE "N LAST :L + LAST BL :L
    NTOWORD :N
    MAKE "L SE :L RESULT
    PRINT :L)
END
```

To try this out, we must feed it a two term list and the number of terms we want generated. For example, try

```
FIBONACCI [1 1] 10
1 1
1 1 2
1 1 2 3
1 1 2 3 5
1 1 2 3 5 8
1 1 2 3 5 8 13
1 1 2 3 5 8 13 21
1 1 2 3 5 8 13 21 34
1 1 2 3 5 8 13 21 34 55
1 1 2 3 5 8 13 21 34 55 89
1 1 2 3 5 8 13 21 34 55 89 144
```

The procedure first prints the starting series. Then it repeatedly selects the last two terms from the series (**LAST :L** and **LAST BL :L**) and adds them. The resulting number is converted to a word and added to the end of the list. You ask for any number of terms you like, but after about 22 you have exceeded the capacity of the computer (remember the largest number possible is about 32000), and the results become erratic.

Sorting is another list operation which is useful and simpler than forming sentences. We could use the above conversion subprocedure in a procedure for sorting a list of numbers, but for variety we'll work on alphabetizing a list of words. Because words can be made up of digits, and because the digits are ordered in the ASCII sequence, the procedures will sort numbers as well.

There are many ways one can sort a list. This is a topic of continuing interest in computer science, and you might want to write Super LOGO procedures which implement the various strategies. Here we'll sort by making a new list. We'll take each item on the old list and insert it where it belongs in the partially completed new list. We need a procedure which inserts a word into a list at a given position.

```
TO INSERT :LIST :WORD :P :J
  MAKE "J []
  REPEAT :P - 1
    (MAKE "J SE :J FIRST :LIST
    MAKE "LIST BF :LIST)
  OUTPUT SE SE :J :WORD :LIST
END
```

INSERT begins with an empty list. Words are transferred to list **J** one at a time until the position for the new word is reached. (The variable **P** is the position.) Then two **SE** functions are used to put together the copied list, the word to insert, and the rest of the list. This procedure should be tested; try

```
FT INSERT [AA CC EE GG] "B 2
PRINT RESULT
AA B CC EE GG
```

Next we'll write the main procedure. First we transfer the first word to a new list. Then we take words from the old list and decide where to put them in the new list.

```
TO ALPHA :LIST :NEW :W :P
  FT
  MAKE "NEW SE [] FIRST :LIST
  MAKE "LIST BF :LIST
  WHILE :LIST <> []
    (MAKE "W FIRST :LIST
    COMPARE :W :NEW
    INSERT :NEW :W RESULT
    MAKE "NEW RESULT
    MAKE "LIST BF :LIST)
  PRINT :NEW
END
```

We use the subprocedure **COMPARE** (as yet unwritten) to compare the word in **W** with the words in the list **NEW** and to output the position for **W** in this list. This result is used by **INSERT** to update the list in **NEW**. If you want to follow the process, insert the command **PRINT :NEW** after the command

```
MAKE "NEW RESULT
```

The next step is to write the procedure **COMPARE**.

```
TO COMPARE :WORD :NEW :W1 :N
  MAKE "N 1
  WHILE :NEW <> []
    (MAKE "W1 FIRST :NEW
    COMPWORD :WORD :W1
    IF RESULT (MAKE "N :N + 1
      MAKE "NEW BF :NEW)
    ELSE (OUTPUT :N)
  )
  OUTPUT :N
END
```

This procedure compares a word and the words on a list by·taking words from the list in order and using the subprocedure **COMPWORD** (as yet unwritten) to compare the two words. If **:WORD** precedes the word from **:NEW (:W1)**, then **COMPWORD** should return **"FALSE** so that **COMPARE** will output the current value in **N**. If **:WORD** does not precede the word from **:NEW**, then **COMPWORD** should return **"TRUE** so that **COMPARE** will increment the value in **N** and move ahead to the next word in the list. If the list is empty, **:WORD** belongs at the end of the original list, and the value in **N** points to the end of that list. Finally we must actually make the comparison of the two words and send output as described above.

```
TO COMPWORD :W1 :W2
  IF :W1 = :W2 (OUTPUT "TRUE)
  WHILE FIRST :W1 = FIRST :W2
    (MAKE "W1 BF :W1
     MAKE "W2 BF :W2)
  IF ASCII FIRST :W1 <
     ASCII FIRST :W2
    (OUTPUT "FALSE)
  OUTPUT "TRUE
END
```

First we check to see if the two words are the same. If they are, their order in the list does not matter, and we arbitrarily output **"TRUE**. We then work our way letter-by-letter down the two words until we find a pair of letters which are different. When we find such a pair, we compare the positions of the two letters in the ASCII sequence. Because the letters are in order starting with **A**, the lower value comes first in alphabetical order. If one word is shorter, this process returns zeros for the ASCII value, which gives the right order.

This completes the set of procedures. To try it out enter some lists.

```
ALPHA [DOG CAT BEE HORSE ZEBRA]
BEE CAT DOG HORSE ZEBRA
```

(If you added **PRINT :NEW** as mentioned on page 131, you'll see the stages of the sort process displayed on the screen.)

```
ALPHA [DOG ZEBRA 456 23]
23 456 DOG ZEBRA
```

The latter example shows that the procedures sort numbers as well. To really try this out you might want a procedure that allows you to enter a longer list for sorting. We've already seen an example of such in Chapter 22.

24. CARD GAMES

The sequence of cards in a deck forms a list. In principle, we can program the rules of card games into a main procedure. We may even be able to use the multitasking capabilities of Super LOGO to create multiple players with differing strategies.

The first task is to write a procedure which can shuffle a deck of 52 cards. We'll stick to simple games, ones in which suit (hearts, diamonds, etc.) does not matter. This will save us some typing, but extension to games involving suits is possible.

To shuffle the deck we'll proceed as follows. The list containing the deck will be rotated a random number of times (by rotating we mean: take the first item from the list and place it at the end of the list). The first card will be transferred to the end of the shuffled list, and the whole process will be repeated with a deck one card smaller.

```
TO ROTATE :N :L
  REPEAT :N-1
  (MAKE "L SE BF :L FIRST :L)
  OUTPUT :L
END

TO SHUFFLE :N :L :I :J
  PRINT "SHUFFLING
  MAKE "J []
  REPEAT :N
    (ROTATE (1+RANDOM :N-:I) :L
     MAKE "L RESULT
     MAKE "J SE :J FIRST :L
     MAKE "L BF :L
     MAKE "I :I+1
    )
  OUTPUT :J
END
```

ROTATE outputs the rotated list. **SHUFFLE** builds the shuffled list in **J**. Notice that the argument for **RANDOM** must be adjusted as the unshuffled list becomes shorter. **SHUFFLE** outputs the shuffled list.

The procedure **DECK** actually generates the deck, gets it shuffled and outputs the result to the main procedure.

```
TO DECK :L :J
  MAKE "L [A K Q J 1Ø 9 8 7 6]
  MAKE "L SE :L [5 4 3 2]
  MAKE "L SE SE SE :L :L :L :L
  SHUFFLE 52 :L Ø []
  OUTPUT RESULT
END
```

This set of procedures generates a shuffled deck of cards. They could be used as they stand for any card game. To test them try

```
FT DECK PRINT RESULT
```

(It takes a few moments for the program to shuffle and print.)

Now we must pick a specific game. Perhaps the simplest card game is War. War is a two player game. Each player is given half the deck. Each player plays the next card from his or her hand, and the high card takes both cards played. Captured and played cards are placed at the end of the winner's hand. The point is to capture all the cards, a process which is usually very time consuming. However, the lack of strategy makes the game a good one to start with.

We choose to make the game a four-turtle task (because it is list processing, the turtles will never appear on the screen). The master turtle will deal the cards and handle the rules. That is, the master turtle will compare the cards played and award the cards to the winner at each step. The main procedure is as follows:

```
TO WAR :L :J :C1 :C2 :OVER
  FT DECK MAKE "L RESULT
  MAKE "J []
  REPEAT 26
    (MAKE "J SE :J LAST :L
    MAKE "L BL :L)
  HATCH 1 PLAYER :L 26
  HATCH 2 PLAYER :J 26
  HATCH 3 ENDER Ø
  WHILE :OVER = Ø
  (WHILE :C1 = Ø
    (MAKE "C1 MAIL 1)
  WHILE :C2 = Ø
    (MAKE "C2 MAIL 2)
  NUMBERTOCARD :C1
  PRINT RESULT; PRINT CHAR 32;
  NUMBERTOCARD :C2
  PRINT RESULT MAKE "J RC
  IF :C1 >= :C2 (SEND 1 :C2
    SEND 2 15)
  IF :C2 > :C1 (SEND 1 15
    SEND 2 :C1)
  MAKE :C1 Ø MAKE :C2 Ø
  MAKE "OVER MAIL 3
  )
END
```

This main procedure first sets the screen display and calls for a shuffled deck (**DECK**). The **REPEAT** divides the shuffled deck into two hands by transferring the last 26 cards to the list **J**. Then the two players are hatched (using the **PLAYER** procedure, not yet written) and are given their cards. The remainder of the procedure implements the rules of the game. The four tasks (turtles) are going to have to communicate using the **SEND** command and the **MAIL** function. Only numbers can be sent and received, so some codes for the face cards (which are words like **A** and **K**) must be established. Because the highest number card has the value 1Ø we will assign the Jack 11, the Queen 12, the King 13, and the Ace 14. The subprocedure **NUMBERTOCARD** converts the numbers 11 to 14 to the appropriate card symbol.

The sequence

```
WHILE :C1 = Ø
  (MAKE "C1 MAIL 1)
```

waits until player one sends a message (plays a card). The message will be the numerical value of, or assigned to, the card. The procedure then uses a similar set of commands to wait for a second card. Then the two cards are converted back to symbols (by **NUMBERTOCARD**), and the symbols are printed out. This is useful for us when we are checking the procedures, but it is not essential for the game. Finally, the values of the two cards are compared and appropriate messages are sent. The value 15 indicates that the player lost that card; a value between 2 and 14 indicates the card that was won. The procedure as written gives player 1 an unfair advantage; player 1 wins all ties. If player 1 is originally dealt an ace, then player 1 can never lose! We'll live with that limitation here, but you might want to change the procedure to correspond to your local version of the game. Finally the values of **C1** and **C2** are reset so that **WAR** will wait for cards from the players in the next round.

The game is over when one player is out of cards. When this happens the losing turtle will send a message to turtle 3 (**ENDER**) which will in turn send a message back to turtle Ø (**WAR**). The message from **ENDER** is kept in **OVER**. The first **WHILE**

```
WHILE :OVER = Ø
```

continues to check until such a message is received from **ENDER**. You might wonder why we used a separate turtle to keep track of the end of the game when the player turtles are already sending messages directly to turtle Ø. It is easier to distinguish a game ending message by its source (from turtle 3) than to distinguish it from cards by its value. Both players use the same procedure.

```
TO PLAYER :L :COUNT :N :T :ME
  NTOWORD ME MAKE "ME RESULT
  WHILE :L <> []
    (NTOWORD :COUNT
    PRINT SE SE :ME "HAS RESULT
    MAKE "N FIRST :L
    CARDTONUMBER :N Ø
    SEND Ø RESULT
    MAKE "T Ø
    WHILE :T = Ø
      (MAKE "T MAIL Ø)
    IF :T = 15 (MAKE "L BF :L
      MAKE "COUNT :COUNT-1)
    ELSE (MAKE "COUNT :COUNT+1
    NUMBERTOCARD :T
    MAKE "L SE SE BF :L
    FIRST :L RESULT)
  )
  SEND 3 ME
END

TO ENDER :OVER
  WHILE :OVER = Ø
    (MAKE "OVER MAIL 255)
  NTOWORD :OVER
  PRINT SE SE "PLAYER RESULT
  "LOST
  SEND Ø :OVER
END
```

Let's take **PLAYER** step by step. We want each player to report on the status of his or her hand at each step. These are two independent procedures, so we cannot rely on player 1 reporting before player 2. (A winning play requires more commands to process than a losing play.) The first line of commands loads a word which is the player's number into the variable **ME**. The major portion of the procedure is a loop which repeats until the player is out of cards (**WHILE :L <> []**). The variable **:COUNT** keeps track of the number of cards. This could also be done by use of the **LENGTH** subprocedure on **:L**, but this way is much faster. **COUNT** is converted into a word so that it can be combined into a sentence like "**2 HAS 24**."

The three lines

```
MAKE "N FIRST :L
CARDTONUMBER :N Ø
SEND Ø RESULT
```

take the next card off the top of the hand, convert the card into a number (A = 14, etc.), and send the card to turtle Ø. The next three lines make the player wait for turtle Ø to tell them whether the card won another one or lost. The message 15 means that the card was lost, so the card is removed from the list **L** and the **COUNT** is decreased by 1. Any other message is the numeric code for the card that was won. In that case the **COUNT** is increased by 1, the numeric code is translated back to a card, and the card played and the card won are moved to the bottom of the hand (the list **L**).

This process continues unless the list is empty. If the list is empty—and those of you who have ever played War know how unusual it is to ever finish a game — then a message is sent to turtle 3.

ENDER waits for a game ending message, prints an appropriate message on the screen, and lets turtle Ø know that it is all over.

WAR uses several other subprocedures. We gave **NTOWORD** in the beginning of Chapter 23 — we won't repeat it here. The other two follow.

```
TO CARDTONUMBER :CARD :N
  MAKE "N :CARD
  IF :CARD = "A (MAKE "N 14)
  IF :CARD = "K (MAKE "N 13)
  IF :CARD = "Q (MAKE "N 12)
  IF :CARD = "J (MAKE "N 11)
  OUTPUT :N
END

TO NUMBERTOCARD :N :CARD
  IF :N<11 (NTOWORD :N
  MAKE "CARD RESULT)
  IF :N=14 (MAKE "CARD "A)
  IF :N=13 (MAKE "CARD "K)
  IF :N=12 (MAKE "CARD "Q)
  IF :N=11 (MAKE "CARD "J)
  OUTPUT :CARD
END
```

These two provide the conversion between the symbols we expect for cards and the number codes which can be sent between procedures as messages.

The preceding example illustrates techniques which can be used for a variety of card games. Notice that we could have as many players as we wanted by use of multiple turtles. In games which involve strategy, separate procedures which implement different strategies could be written for each player, and those strategies could be evaluated by the results. Instead of doing exactly that, we'll show a different type of game, a game in which the user is one of the players. This time one of the players is the computer, so a strategy will be implemented in a procedure, but the user is free to pick any strategy desired.

The game is the game of blackjack or 21. The rules are simple; the object is to get closer to 21 than your opponent without exceeding 21. The opponent is the dealer, one of whose cards is not visible until a hand is completed. The dealer has several advantages. The dealer wins ties, and the dealer plays last, thus giving you the opportunity to exceed 21 and lose. One other complication is that all face cards have a value 1Ø, and the aces can be counted as either 1 or 11.

The game as played seriously has a few additional rules which are included to make betting more interesting. We won't bother with those here.

The main procedure (**BJACK**) plays the role of the dealer, that is it handles the cards and it plays the dealer's hands. The first step is to shuffle the cards using **DECK**.

BJACK is very long because of the number of non-repeating steps, but when we follow it through, we find that it is pretty simple. Following **DECK**, which supplies a shuffled deck, a series of **MAKE** and **PRINT** commands deal one card face down to the dealer, one to the player, a second face up to the dealer, and a second to the player. The cards that are known to the player at this time are printed.

The procedure **CHECK** calculates the value of any hand sent to it as a list and returns the value as a number. The value of the dealer's hand is kept in **D**, and the value of the player's hand is kept in **P**. A value of 21 with just two cards (an ace and a face card or a 1Ø) is an automatic win. The sequence

```
IF :D=21 (PRINT [DEALER WINS])
ELSE (
  IF :P=21 (PRINT [YOU WIN])
```

checks for that situation. If neither **IF** condition is true, then the procedure begins executing the commands under the **ELSE** which follows the above three lines. If either of the **IF** conditions is true, the hand is over and a new hand is begun (the outermost **WHILE :L <> [ ]**).

Next the player plays his or her hand. Cards can be drawn until the value of the hand is over 21. The player is asked to respond with a **Y** if they want another card. If they do, a card is dealt off the top of the deck, the new hand is printed, and a new value of the hand is obtained from **CHECK**.

```
TO BJACK :L :DEAL :PLAY :D :P :C
  FT DECK MAKE "L RESULT
  WHILE :L <> []
    (MAKE "DEAL FIRST :L
    MAKE "L BF :L
    MAKE "PLAY FIRST :L
    MAKE "L BF :L
    MAKE "DEAL SE :DEAL FIRST :L
    MAKE "L BF :L
    PRINT [DEALER SHOWS —];
    PRINT BF :DEAL
    MAKE "PLAY SE :PLAY FIRST :L
    MAKE "L BF :L
    PRINT [YOU HAVE —];
    PRINT :PLAY
    CHECK :DEAL MAKE "D RESULT
    CHECK :PLAY MAKE "P RESULT
    IF :D=21 (PRINT [DEALER WINS])
    ELSE (
      IF :P=21 (PRINT [YOU WIN])
      ELSE (
        MAKE "C "Y
      WHILE :P<22 & :C= "Y (
      PRINT "CARD? MAKE "C RC
      IF :C= "Y
        ( MAKE "PLAY SE :PLAY
        FIRST :L
        MAKE "L BF :L
        PRINT :PLAY
        CHECK :PLAY
        MAKE "P RESULT))
    IF :P>21 (
      PRINT [DEALER WINS])
    ELSE (
      WHILE :D<17 (
      MAKE "DEAL SE :DEAL
      FIRST :L
      MAKE "L BF :L
      PRINT SE "DEALER :DEAL
      CHECK :DEAL
      MAKE :D RESULT))
    IF :D> =:P & :D<22 (
      PRINT [DEALER WINS])
    ELSE (PRINT [YOU WIN])
    ))
    PRINT [NEXT HAND]
  )
END
```

The actions are produced by the following section of the procedure:

```
MAKE "C "Y
WHILE :P<22 & :C= "Y (
PRINT "CARD? MAKE "C RC
IF :C= "Y (
  MAKE "PLAY SE :PLAY
  FIRST :L
  MAKE "L BF :L
  PRINT :PLAY
  CHECK :PLAY
  MAKE "P RESULT))
```

Now we check to see if the player has lost.

```
  IF :P>21 (
    PRINT [DEALER WINS])
```

If not, then the dealer must play according to commands following the **ELSE**.

```
WHILE :D<17 (
  MAKE "DEAL SE :DEAL
  FIRST :L
  MAKE "L BF :L
  PRINT SE "DEALER :DEAL
  CHECK :DEAL
  MAKE :D RESULT))
```

The strategy for the dealer is to continue to draw until the value of the hand is 17 or greater, regardless of what the player has. This seems like a very simple strategy; in fact it is what most casinos use as close to the optimum for the dealer.

The last few steps are straightforward. We check to see if the dealer's total is too high, and if not, who has the higher total.

There are some obvious extensions we could add to the procedure. We have not kept score, and this would be easy to do. We have only allowed one pass through the deck; in fact an incomplete last hand will make the computer do strange things. We could have the condition on the first **WHILE** check to see that there are enough cards for two hands (about 1Ø), or if we wanted to get fancy we could reshuffle with a deck minus the cards in play. These additions would make the game closer to the standard game, but they would not increase our knowledge of Super LOGO much, so we will let them pass.

We still have to write the procedure **CHECK**.

```
TO CHECK :HAND :N :CARD :T :S
  WHILE :HAND <> []
    (MAKE "CARD FIRST :HAND
    MAKE "HAND BF :HAND
    CARDTONUMBER :CARD Ø
    MAKE "T RESULT
    IF :T=13 (MAKE "T 1Ø)
    IF :T=12 (MAKE "T 1Ø)
    IF :T=11 (MAKE "T 1Ø)
    IF :T=14 (MAKE "T 11
      MAKE "S 1)
    MAKE "N :N + :T)
  IF :N>21 & :S=1
    (MAKE "N :N - 1Ø)
  OUTPUT :N
END
```

CHECK takes cards one by one off the list, and converts these cards to a number using **CARDTONUMBER**. The face cards are revalued to 1Ø, and the ace is revalued to 11 and flagged. The card values are totaled as they are processed.

```
MAKE "N :N + :T)
```

If the total is over 21 and the hand contains an ace, then the total is reduced by 1Ø which corresponds to assigning the ace a value of 1.

This completes the procedures for the game of blackjack, and it completes our discussion of card games. However, it does not exhaust the possibilities. We encourage you to try other simple games using the techniques illustrated in this chapter.

25. WORD GAMES

Word games provide a number of interesting possibilities for projects. The key to many word games is a procedure for generating all combinations of a given number of letters. The arrangements are generated by switching letters and then switching letters with other letters, etc. While it seems like this should be recursive, it is not easy to come up with the recursive procedure. We'll use this opportunity to show one way to attack such a problem. However, you must realize that this is only a potentially useful suggestion, not a foolproof method for dealing with every recursive problem.

The first thing we realized was that we would need a procedure to switch two letters in a word. Of course this is not the top-down process we advocated in Chapter 6, but that is a technique, not a rule. The following procedure switches letters in positions **P1** and **P2** of the word **W**. The assumption is that **P1** comes before **P2**.

```
TO SW :W :P1 :P2 :L :J1 :J2 :T
 MAKE "J1 "  MAKE "J2 "
 MAKE "L 1
 REPEAT :P2 - 1
   (IF :L<:P1 (
     MAKE "J1 WORD :J1 FIRST :W)
   IF :L =:P1 (
     MAKE "T FIRST :W)
   IF :L>:P1 (
     MAKE "J2 WORD :J2 FIRST :W)
   MAKE "W BF :W MAKE "L :L+1)
 MAKE "J1 WORD WORD WORD WORD
   :J1 FIRST :W :J2 :T BF :W
 OUTPUT :J1
END
```

To switch two letters within a word we have to divide the word into five pieces: the letters in front of the first switchable character (in **J1**), the first switchable character (in **T**), the letters between the two switchable characters (in **J2**), the second switchable character (**FIRST :W** after the others are peeled off), and the rest of the word (**BF :W**). The procedure **SW** uses the **REPEAT** with **IF** conditions to peel off the first three pieces and then puts the new word together with four **WORD** operations.

Next we wrote procedures for the simple particular cases. The simplest case to handle is a two letter word. There are two combinations, the original word and the word with two letters switched.

```
TO PERM2 :WORD
  PRINT :WORD
  SW :WORD 1 2
  PRINT RESULT
END
```

We can think of **PERM2** as giving the combinations generated by switching letter two with itself (that is, no change) and by switching letter 2 with letter 1.

Some examples to try:

```
FT PERM2 "GO

FT PERM2 "AT
```

Three letter words are generated by switching letter 3 with itself and calling **PERM2**, then by switching letter 3 with letter 2 and calling **PERM2**, and finally by switching letter 3 with letter 1 and calling **PERM2**.

```
TO PERM3 :WORD :NEW :N
  MAKE "NEW :WORD
  MAKE "N 3
  REPEAT 3 (
    IF :N<3 (SW :NEW :N 3
      MAKE "NEW RESULT)
    PERM2 :NEW
    MAKE "NEW :WORD
    MAKE "N :N – 1)
END
```

You might check this with a three letter word to see that it generates all the combinations of the letters. (For example, **FT PERM3 "CAT**.)

Four letter combinations are generated by switching letter 4 with itself, with letter 3, with letter 2, and with letter 1, and in each case calling **PERM3** after the switch.

```
TO PERM4 :WORD :NEW :N
  MAKE "NEW :WORD
  MAKE "N 4
  REPEAT 4 (
    IF :N<4 (SW :NEW :N 4
      MAKE "NEW RESULT)
    PERM3 :NEW
    MAKE "NEW :WORD
    MAKE "N :N – 1)
END
```

We can now write the recursive, general version by comparing **PERM3** and **PERM4**. There are several places where the number 3 appears in **PERM3**; the number 4 appears in the equivalent positions in **PERM4**. We simply replace those numbers with a variable **X**.

```
TO PERM :WORD :X :NEW :N
  MAKE "NEW :WORD
  MAKE "N :X
  REPEAT :X (
    IF :N<:X (SW :NEW :N :X
      MAKE "NEW RESULT)
    IF :X=3 (PERM2 :NEW)
    ELSE (PERM :NEW (:X - 1))
    MAKE "NEW :WORD
    MAKE "N :N - 1)
END
```

We chose to keep the **PERM2** as a special case for further applications, so the recursive call is controlled by the **IF ... ELSE** combination. The new variable **X** is put early in the list of variables to reduce the variable list on the call. (**NEW** and **N** are just local variables which do not need to be given a value on the call).

To use this we must give it a word and the length of the word. The length can be obtained by a procedure.

```
TO START :WORD :N :NEW
  FT
  SIZE :WORD MAKE "N RESULT
  PERM :WORD :N :NEW
END

TO SIZE :WORD :COUNT
  WHILE :WORD <> "
    (MAKE "COUNT :COUNT+1
    MAKE "WORD BL :WORD)
  OUTPUT :COUNT
END
```

This is now a complete set of procedures which will print out every combination of letters in a word you give it. When you try it out remember that the number of combinations becomes large very fast. There are 12Ø combinations of 5 letters, 72Ø combinations of 6 letters, 5Ø4Ø combinations of 7 letters, etc.

Now you may wonder what we could do with these procedures. One possibility is to make a prompter for Scrabble. If you give it the letters you have, it will generate every possible combination. You might want to modify **PERM2** so that it prints only those combinations which have particular letters in particular positions. We give another possible use. We'll modify **PERM2** so that it prints out every combination and allows the user to save any that are real words in a list which is printed out at the end. One could let a child make the choices and review the list at the end, or one could assign two turtles and let two players compete for finding the most words from a set of letters. We can save a lot of work by making the list of saved words a global variable (we'll call it **FIND**). Then we won't have to output it and pick it up as we move back up through the recursive stack from **PERM2**. We need to replace **START** and modify **PERM2:**

```
TO FINDWORD :WORD :N :NEW
  FT MAKE "FINAL []
  SIZE :WORD MAKE "N RESULT
  PERM :WORD :N :NEW
  PRINT :FINAL
END

TO PERM2 :WORD :SAVE
  PRINT :WORD
  MAKE "SAVE RC
  IF :SAVE= "Y (MAKE "FINAL SE
                 :FINAL :WORD)
  SW :WORD 1 2 MAKE :WORD RESULT
  PRINT :WORD
  MAKE "SAVE RC
  IF :SAVE= "Y (MAKE "FINAL SE
              :FINAL :WORD)
END
```

You might try these examples:

```
FINDWORD "REAL

FINDWORD "CATS
```

As usual there are a number of ways we could make this set of procedures better. The user could be prompted to push the Y key to save a word. The current list of saved words could be searched (using **MEMBER** as in Chapter 22) to prevent double saving of the same word, which could happen if a letter occurs twice in the original word. As usual we will leave such changes to you as projects.

26. DICE GAMES

Dice games provide further illustrations of list processing, and they provide a rich source of interesting projects. Many games are based on throwing multiple dice. Some of them allow you to select some dice for rethrowing, thus allowing you chances to improve the result. Instead of picking a particular game, we will just give the dice throwing procedures. For purpose of illustration we will assume that the games use five dice and that the games allow rethrow of selected dice two times. The main procedure is called **DICETHROW**.

Note: You may want to **MERGE** in the procedures that you created in Chapter 22 if they are not already in memory. Many of them will be used in this chapter.

```
TO DICETHROW :SET :REDO
FT MAKE "SET []
REPEAT 5 (THROW
  MAKE "SET SE :SET RESULT)
REPEAT 2
  (PRINT :SET
   PRINT [LIST DISCARDS]
  MAKE "REDO RL
  IF :REDO <> [] (
    DIFFERENCE :REDO :SET
    MAKE "SET RESULT
    LENGTH :SET MAKE "N 5-RESULT
    REPEAT :N (THROW
      MAKE "SET SE :SET RESULT)
))
PRINT SE [FINAL HAND] :SET
END
```

There are really no new techniques used in this procedure. The subprocedure **THROW** is used to actually throw the dice; the subprocedure **DIFFERENCE** is used to eliminate the words in the list **REDO** from the words in **SET**. To function properly here **DIFFERENCE** must eliminate only one occurrence in **SET**, not every occurrence. **LENGTH** is a subprocedure given in Chapter 22 which returns the length of a list.

THROW picks one number from a list of six.

```
TO THROW :N :CHOICE
  MAKE "CHOICE [1 2 3 4 5 6]
  PICKRANDOM :CHOICE 6
  OUTPUT RESULT
END
```

THROW uses the subprocedure **PICKRANDOM** and its subprocedures **PICK** and **LENGTH** to do the actual random selection; we've seen **PICK**, **PICKRANDOM**, and **LENGTH** in Chapter 22.

DIFFERENCE is a new subprocedure which might be useful in a variety of projects.

```
TO DIFFERENCE :OUT :LONG
  WHILE :OUT <> []
  (REMOVE (FIRST :OUT) :LONG
  MAKE "OUT BF :OUT
  MAKE "LONG RESULT)
  OUTPUT :LONG
END
```

The procedure works its way through the list of words to be eliminated. **REMOVE** must take a word and a list and remove the word from the list. The shortened list is returned as the result, and the processed word is peeled off the list in **OUT**.

```
TO REMOVE :WORD :LIST :N
  LENGTH :LIST MAKE "N RESULT
  REPEAT :N (
    IF :WORD = FIRST :LIST
      (OUTPUT BF :LIST)
    ELSE (MAKE "LIST SE
      BF :LIST FIRST :LIST)
  )
  PRINT SE :WORD [NOT IN LIST]
  OUTPUT :LIST
END
```

The critical part of **REMOVE** is the loop. The word in **WORD** is compared with the first word in **LIST**. If they are the same, the shortened list is output. If they are not the same, then the words in the list are rotated to make another one first. If the list is rotated completely without the procedure being completed with an **OUTPUT** command, then the word is not in the list. A message indicating that fact is printed and **LIST** is returned unchanged.

When you try this set of procedures remember that the list of dice to be rethrown is a Super LOGO list. Therefore, in typing the numbers remember to leave spaces between them.

Once all the required procedures are present, you can execute **DICETHROW** simply by entering "**DICETHROW**" in RUN mode.

For our next example we use a variation of the word search game based on dice. The differences are that the faces of the dice show letters instead of numbers and that the order in which the individual dice are thrown matters too. The objective is to produce a randomly selected and arranged square array of letters. The players are then to pick words from the array using adjacent letters. Of course the computer cannot be programmed to decide if particular combinations of letters form real English words, so we will just produce the square of letters and leave the rules of the rest of the game to the players. We could pick a square of any size, but 4x4 gives enough variety.

Each die shows six letters. The letters can be picked in any fashion, but we won't get much variety if we make all the dice the same. Ideally the letters on the 16 dice should be selected to correspond to the frequency of occurrence of those letters in English.

```
TO HIDEWORD
  FT
  HATCH 1 DICE [N I D U T K]
  HATCH 2 DICE [R A C L T E]
  HATCH 3 DICE [M D R A N T]
  HATCH 4 DICE [N A G O S V]
  HATCH 5 DICE [O C A S E U]
  HATCH 6 DICE [E M R D A C]
  HATCH 7 DICE [D I N S T W]
  HATCH 8 DICE [B T L Y O E]
  HATCH 9 DICE [L G W P U O]
  HATCH 1Ø DICE [A H Y F I E]
  HATCH 11 DICE [B I K O F R]
  HATCH 12 DICE [D V N Z E A]
  HATCH 13 DICE [J E B I R M]
  HATCH 14 DICE [O P A N T H]
  HATCH 15 DICE [Y E G U K L]
  HATCH 16 DICE [L U P A T S]
  REPEAT 2ØØØ ()
  REPEAT 4 (PRINT [ ]
  REPEAT 4 (
    PRINT CHAR (MAIL 255);
    PRINT CHAR (32);))
END
```

This procedure uses multiple turtles to randomize the order in which the dice are thrown. The results of each throw will be sent as mail to this master procedure; the **MAIL 255** reads them in the order in which they are sent, regardless of the source. The **REPEAT 2ØØØ ()** is a delay to make sure that this master procedure does not start reading mail before all the other turtles have sent their results. Without a delay, **MAIL 255** might return Ø in some cases. Because the letters are to be sent as mail from the other turtles, the letters must be sent in the form of numbers. The obvious code for the letters is the ASCII code, and the **CHAR** function converts these codes back into the letters. The last two **REPEAT** statements cause the letters to be printed out in a 4 x 4 square.

The above paragraph pretty well defines what **DICE** must do. It must select one entry from a list at random and send the ASCII code of the letter selected as a message to turtle Ø. Note that **DICE** calls on the procedure **NTOWORD** which we saw in Chapter 23.

```
TO DICE :L :A :N
  MAKE "A []
  REPEAT 6 (
    NTOWORD ASCII FIRST :L
    MAKE "A SE :A RESULT
    MAKE "L BF :L)
  REPEAT RANDOM 5Ø ()
  PICKRANDOM :A 6 SEND Ø RESULT
END
```

We first convert each letter to its corresponding ASCII number and convert that number to a word. The list **A** then contains the ASCII numbers for the letters in the original list. Next we put in a random delay

```
REPEAT RANDOM 50 ()
```

so that it is unpredictable when each of the 16 turtles returns its message to the queue of messages for turtle Ø. Finally we pick a random entry off the list and send it as a message (**PICKRANDOM** and its subprocedures are Chapter 22).

> Note: **HIDEWORD** takes a while to execute and display results. Do not worry if your screen is blank for some moments after you have entered the command **"HIDEWORD"** in RUN mode.

With this example, we complete the tutorial on list processing. You may have noticed that in the last several chapters we have not introduced much in the way of new Super LOGO features. Instead we have been making new combinations to solve a variety of problems. This is because we have all the features we need at hand; solving new problems is mainly a process of analyzing and ordering the solutions. That we will leave to you.

27. GRAB BAG

In this last chapter, we return to graphics and give a final set of sample programs which we hope will give you ideas for your own projects. We have introduced all the features of Super LOGO earlier, so we will give these without lengthy comments.

The first set is controlled by the procedure **BOND**.

```
TO BOND
  WHILE 1
    (COLORSET 1
    CLEAR HT DELAY 1000
    TUNNEL
    WALK
    PAINT)
END

TO WALK
  SX 28 MAN2 ST DELAY 2000
  REPEAT 29
    (MAN2 DELAY 100
    HT SX XLOC ME + 3
    MAN1 ST DELAY 100
    )
  MAN2
  DELAY 800 SX XLOC ME - 8
  DELAY 500 SX XLOC ME + 16
  DELAY 500 SX XLOC ME - 16
  DELAY 500 SX XLOC ME + 8
  REPEAT 3 (
    HT DELAY 20
    ST DELAY 30)
END

TO TUNNEL
  PC 1 HT SX 60 SH 0
  REPEAT 18
    (FD 20 RT 124 FD 56
    BK 56 LT 104)
END

TO MAN1
  SHAPE RRUFFFLLDFLFR-
FFLFFRRRFLLFFRRF-
LFLLLFFRRFLFRRFL-
FFLFLFLFLFFLFRFF-
FFLLFRRRFLFFRFL-
FFRRFF
END
```

```
TO MAN2
  SHAPE RRUFFFFLLDFF-
FFLFFRRRFLLFFRRF-
LFLLLFFRRFLFRRFL-
FFLFLFLFLFFLFRFF-
FFLLFRRRFLFFFFFF
END

TO PAINT
  PC 2 HT MAKE "X 1
  REPEAT 3 (COLORSET Ø
    DELAY 1ØØ COLORSET 1
    DELAY 1ØØ)
  SX 114 SY 1Ø2 SH Ø
  REPEAT 13
    (RAGGED :X
    SX XLOC ME - 6
    SY YLOC ME - 2
    MAKE "X :X+5
  )
END

TO RAGGED :X
  REPEAT 8
    (FD :X RT 135 FD 8
    BK 8 LT 9Ø)
END

TO DELAY :TIME
  REPEAT :TIME ()
END
```

The next set is for a younger audience.

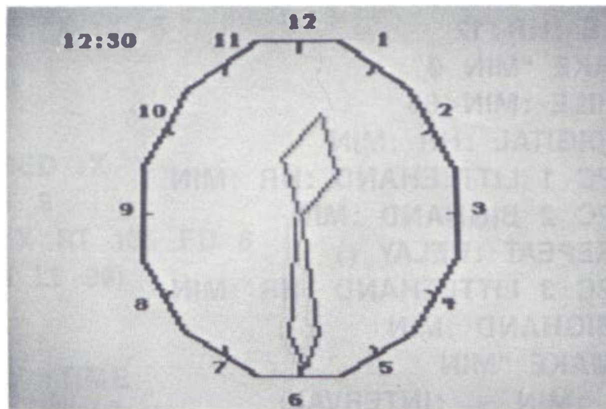
```
TO CLOCK :DELAY :INT
  CLEAR DRAW
  CLOCKFACE
  TIME :DELAY :INT
END

TO CLOCKFACE
  MAKE "NUMBER 12
  SY 180 SX 104 SH 90
  REPEAT 12
    (FD 22 RT 90 FD 5 BK 5
    PU BK 10 PRINT :NUMBER
    FD 10 PD LT 90 FD 22
    RT 30
  MAKE "NUMBER :NUMBER+1
  IF :NUMBER > 12
    (MAKE "NUMBER 1))
 END

TO TIME :DELAY :INTERVAL
  HT
  REPEAT 24
    (MAKE "HR 0
    WHILE :HR<12
      (MAKE "MIN 0
      WHILE :MIN<60
        (DIGITAL :HR :MIN
        PC 1 LITTLEHAND :HR :MIN
        PC 2 BIGHAND :MIN
        REPEAT :DELAY ()
        PC 3 LITTLEHAND :HR :MIN
        BIGHAND :MIN
        MAKE "MIN
          :MIN + :INTERVAL)
        MAKE "HR :HR + 1))
END

TO BIGHAND :MINUTE
  SX 128 SY 96 SH 6*:MINUTE
  LT 8 FD 60 RT 30 FD 18
  RT 130 FD 18 RT 32 FD 60
END
```

```
TO LITTLEHAND :HOUR :MINUTE
  SX 128 SY 96
  SH 30*:HOUR + :MINUTE/2
  LT 32 FD 30 RT 60 FD 30
  RT 120 FD 30 RT 60 FD 30
END
```

```
TO DIGITAL :HOUR :MINUTES
  SX 0 SY 180 PRINT CHAR 32;
  PRINT CHAR 32; PRINT CHAR 32;
  SX 8*(:HOUR<=9 & :HOUR<>0)
  IF :HOUR (PRINT :HOUR)
  ELSE (PRINT 12)
  SX 16 PRINT ": SX 24
  IF :MINUTES<10 (PRINT "0 SX 32)
  PRINT :MINUTES
END
```

Notice that you can set the interval to any number of clock minutes and that you can set the speed with **:DELAY**. Try running

```
CLOCK 300 5
```

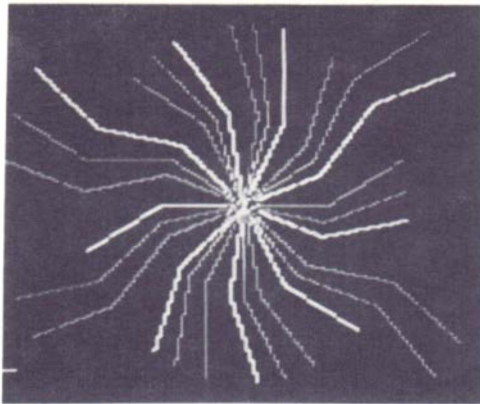


Next we give another colorful design.

```
TO SPIDER :X
  COLORSET 1 BG 0
  REPEAT 36
    (HATCH 1 OFFSET :X :C
    MAKE "C :C+1 RT 10)
  VANISH
END

TO OFFSET :LENGTH :COLOR
  PC :COLOR FD :LENGTH
  LT 30 FD :LENGTH
  RT 30 FD :LENGTH
END
```

Try this with

```
SPIDER 45
```

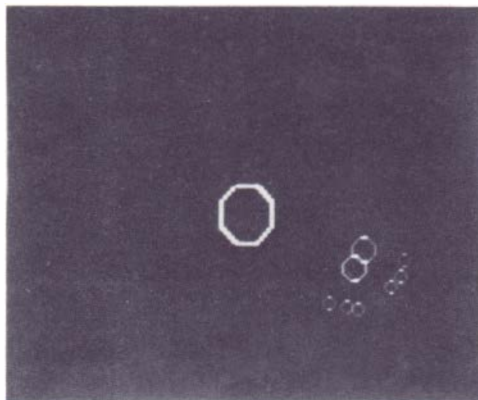


Next we give one which will remind you of the start of every science fiction film you've ever seen. There is no picture in the manual for this one, as the effect is all in the motion.

```
TO SPACETRAVEL
  DRAW
  COLORSET 1 BG Ø HT
  MAKE "X 4
  WHILE 1
    (HATCH 1 STAR1
    RT 67
    HATCH 1 STAR2
    RT 2Ø7
    HATCH 1 STAR1
    RT 114
    HATCH 1 STAR2
    RT 87
    SETX XLOC ME + :X
    IF NEAR 255>30
      (MAKE "X :X * -1
      HATCH 1 PLANET)
    )
  VANISH
END

TO STAR1
  HT
  SHAPE FFRRFRRF
  PU FD 2 ST
  REPEAT 25 (FD 3)
END

TO STAR2
  HT SHAPE F
  PU FD 2 ST
  REPEAT 35 (FD 3)
END

TO PLANET
  HT
  IF XLOC ME>128 (SETH 75)
  ELSE (SETH 3ØØ)
  FD 10 SHAPE FFRFFRFFRFFRFFR-
FFRFFRFF
  PU FD 6 ST
  REPEAT 2Ø (FD 4)
 END
```

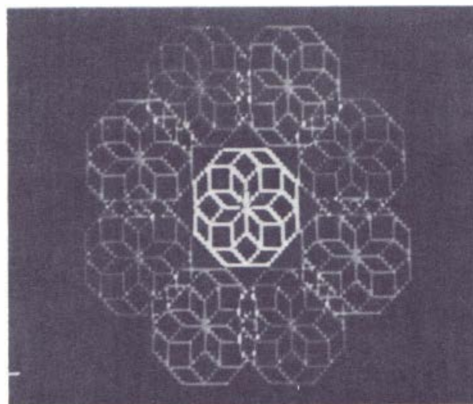
Here's one which shows the orbit of a moon around a planet and which makes use of multiple turtles to simplify the mathematics.

```
TO ORBIT
  COLORSET 1 BG Ø
  FD 1Ø RT 9Ø PC 3
  REPEAT 8 (FD 6 RT 45 FD 6)
  HOME
  PU SETH 9Ø SY 164
  MAKE "MOONPOS Ø
  SHAPE U-
FFFFFRRDFFRFFFFRFFFFRFFFF-
RFFFFRFFFFRFFFFRFFFFRF
  WHILE 1
    (REPEAT 4
      (HATCH 1 MOON :MOONPOS
      REPEAT 6 ()
      MAKE "MOONPOS :MOONPOS+20
    )
    FD 1Ø RT 9
  )
END

TO MOON :POS
  HT PU RT :MOONPOS
  FD 2Ø
  SHAPE UFFFFRRDFRFFRFFRFFR-
FFRFFRFFRFFRF
  ST
  REPEAT 9 ()
  VANISH
END
```



As our last example, we give a final pretty pattern.

```
TO SAMPLE
  COLORSET 1 DRAW BG 0
  NPOLY 8 12 3
  SX 70 SY 72
  N2POLY 8 48 12
END

TO NPOLY :N :S :C
  PC :C
  REPEAT :N
  (POLYGON :N :S
  RT 360/:N)
END

TO POLYGON :N :S
  REPEAT :N (FD :S RT 360/:N)
END

TO N2POLY :N :S1 :S2 :I
  HT PU MAKE "I 1
  WHILE :I<=:N
    (HATCH :I NPOLY :N :S2
      (1+:I-INT(:I/2)*2)
    FD :S1 RT 360/:N
    MAKE "I :I+1
  )
  VANISH
END
```



Of course, you can try this set with other inputs than those given in **SAMPLE**.

Well, we have now reached the point where you are on your own. We are sure that the examples herein have just scratched the surface of what is possible. We hope that you have as much enjoyment working out your own demonstration procedures as we have had in developing these.

APPENDIX

LANGUAGE SUMMARY OF SUPER LOGO FOR THE TANDY COLOR COMPUTER

STARTING LOGO

From plug-in: ROM pack	With the computer power *off*, plug the Super LOGO cartridge into the game slot. Then turn on the computer.

MODES IN SUPER LOGO

The Super LOGO system can be in one of four modes depending upon what the user is doing at the time. A brief explanation of each is given here.

BREAK MODE	is entered upon system startup and any time the user presses [BREAK]. [BREAK] must be pressed twice to interrupt a procedure and enter [BREAK] mode. In this mode, the user can load and/or save programs from tape, make printed copies of programs, or enter EDIT or RUN modes.
EDIT MODE	To get into this mode from BREAK mode, press the [E] key. In this mode, the user can view, create or modify programs.
RUN MODE	To get into this mode from BREAK mode, press the [R] key. In this mode, the user can enter turtle commands, call programs to be run, or enter DOODLE mode.
DOODLE MODE	To get into this mode, press the [@] key when you are in RUN mode. In this mode, the user can use specially marked keys to doodle a picture while creating a procedure.

BREAK MODE

BREAK mode is entered automatically upon starting Super LOGO, and can be entered from any other mode by pressing the [BREAK] key at any time.

It is signified by the **LOGO:** prompt on the screen. The following single-letter commands may be used in BREAK mode.

[SHIFT] [CLEAR] clears the internal program area.

[R] gets you into RUN mode

[E] gets you into EDIT mode.

[P] prints contents of internal program area on the printer connected to the serial port.

[Q] prints same as **P** command, except that the **Q** sends a line feed after a return character.

[L] allows loading of module into the internal program area. After you press **L**, the computer prompts for a module designation with the message **LOAD:**. The module is then read from the specified source into the internal program area. To load from tape, enter **T**.

[M] prompts for a module designation with the message **MERGE:**. It then reads from the specified source and appends the file to the end of the lines currently in the program area. **MERGE** works exactly like **LOAD** except that the loaded file is added to the end of the current program area.

[S] prompts for the module name with the message **SAVE:**, then writes the contents of the internal program area to the specified destination. To save on tape, enter **T**.

Normally the entire program area is saved, but it is possible to specify that only a portion of the program area be saved. To do so, use EDIT mode to insert the special START and END markers in the program text. The START marker consists of two > characters, such as >> inserted just before the first character to be written out. The END marker consists of two < characters, such as <<, just after the last character to be written. Either or both markers may be present. If the START marker is not present, then the save starts at the beginning of the program area. If the END marker is not present, then the save stops at the end of the program area. After the **SAVE** operation, it is up to the user to again use EDIT mode to remove the START and END markers.

EDIT MODE

You can get into EDIT mode by pressing [E] when you are in BREAK mode. In EDIT mode, one can edit the currently loaded modules. To start with a blank program area, press [SHIFT] [CLEAR] in BREAK mode before pressing [E].

The editor is very easy to use. It works on the principle that what you see is what you get. The first line of text (if there is one) is displayed on the bottom line. To enter lines of text, just type them on the screen. The cursor will always appear on the bottom line, but the text may be moved up or down the screen at will. The following keys cause special actions to take place.

[ENTER]	moves the text up one line on the screen, or if already on last line, then adds a new line to the text end.
[↑]	moves the text up one line unless already on the last line.
[↓]	moves the text down one line unless already on the first line.
[←]	moves the cursor left one character, unless already at the beginning of line.
[→]	moves the cursor right one character, unless already at the line end.
[CLEAR]	moves to the top line of the text.
[SHIFT] [↓]	inserts a blank line in front of the current line if the cursor is in column 1 (the current line bumps down off the screen); if the cursor is not in column 1, then the current line is split into two lines at the cursor location.
[SHIFT] [←]	deletes the character under the cursor and moves the remainder of the line left to close the gap. If the line has no characters, then the blank line is removed.
[SHIFT] [→]	inserts a blank into the line at the cursor location by moving the remainder of the line right one space. If the line is already full, then no action takes place.
[SHIFT] [CLEAR]	deletes from the cursor position to the end of the line. If the cursor is in column 1, the entire line is removed from the text.

SHIFT **↑** is used in one of several ways to move quickly forward in the text. When this key combination is pressed, the cursor moves to the very bottom line on the screen. The user has three options of what to do next:

> The first possibility is to press **SHIFT** **↑** again. This causes the text lines to scroll up continuously. The text lines continue to scroll until a key is pressed or until the last line of the text is reached.
>
> The second possibility is to enter a search string. The search string may be up to 16 characters. Next press the **ENTER** key. The text lines scroll up until a line containing the search string is located, or until a key is pressed, or until the last line of text is reached.
>
> The third possibility is to just press the **ENTER** key again. This produces another search for the search string most recently entered.
>
> Thus, to find the first occurrence of the word **BLUE**, the user would press **SHIFT** **↑**, and enter the word **BLUE**. Then when the scrolling stops on a line containing **BLUE**, the user could find the next occurrence by pressing **SHIFT** **↑** **ENTER**. This could be repeated many times, if needed.

BREAK exits EDIT mode and returns to BREAK mode.

@ allows the next character to be one of the specially marked single key command codes. To enter a real @ press **@** twice.

In general, to enter new lines just type each line followed by **ENTER**. To modify a line, move the cursor into place with the arrow keys, then modify text by typing the new text over the old or by inserting or deleting characters as described above.

Note: If the editor quits accepting new text, then the program area is full.

The editor is general enough to be used not only for writing Super LOGO programs, but also for simple word processing applications. After you edit a text file, the file may be printed or saved on disk or cassette for later use. One such use would be writing documentation for modules written in Super LOGO. Since the editor has a maximum line length of 32 characters, a facility is provided to allow for printing of longer text lines on the printer. If a line is ended with an @ character, then no **RETURN** is output at the end of the line. The result will be that the following line on the screen will be printed on the same printer line.

INTERNAL PROGRAM AREA

Super LOGO procedures are entered in the EDIT mode. They can then be saved on disk or tape and re-loaded later to be run again. The program area can have any number of Super LOGO procedures in it. Each procedure begins with a **TO** statement. The **TO** statement must be the first and only statement on a line. Other than that, any number of statements can share a line; each one is separated from the previous one by one or more spaces. Each procedure should end with an **END** statement. The work area may contain many procedures at once. It is a good idea to leave at least one blank line between procedures to improve readability. It is also a good idea to indent program lines to show the logical structure of the program. The examples in this manual are all written in this manner. When a **LOAD** command is used to read a text file into the program area, the previous contents of the program area are first erased. When a **MERGE** command is used to read a text file into the program area, it is added to the end of the current lines in the program area. This provides a way to combine text or procedures stored in different files.

TURTLE SPACE

The turtle is a creature that has a visible shape, a position and a heading. The position is defined by an (X, Y) coordinate pair. The heading is defined by an angle from Ø to 359. In general, the turtle lives on the plane of the display screen. By executing turtle graphics commands, you can make the turtle move about and, if desired, leave a trail. Initially a turtle starts at the home position. The home position is the approximate middle of the screen (X=128, Y=96). The turtle heading at home is zero degrees or straight up. The screen dimension in the X direction (across the screen) goes from Ø at the left edge to 255 at the right edge. The screen dimension in the Y direction (up and down) goes from Ø at the bottom to 191 at the top. The lower left hand corner of the screen has coordinates (Ø,Ø). The upper right corner has coordinates (255,191). The screen is normally a wrap-around space; that is, if the turtle runs off the top of the screen it reappears on the bottom. If it runs off the left, it reappears on the right, etc. In that sense the plane on which the turtle walks is infinite in any direction. The turtle may be pointed in any direction from Ø to 359 degrees. Straight up is Ø degrees, and the direction increases as the turtle rotates to the right, or in a clockwise direction.

SPLIT SCREEN

When you get into RUN mode, the screen is divided into a graphics area and a text area. The text area is the bottom three lines of the screen. The graphics area is the rest of the screen. In split screen mode, neither the turtle nor any lines drawn by the turtle appear in the text area. If the turtle is moved into this area, it becomes invisible until moved back into the graphics area. The user can use the **FULLSCREEN** or **DRAW** command to change from split screen mode to full screen mode. In full screen mode, text lines still use the bottom lines of the screen but the turtle and lines drawn are visible on the entire screen.

FULLTEXT SCREEN

FULLTEXT mode uses the entire screen for text only. It is entered by the **FULLTEXT** or **FT** command and it is exited via the **SPLITSCREEN**, **DRAW** or **FULLSCREEN** commands. The turtle is invisible in **FULLTEXT** mode, and lines are not drawn. The entire screen becomes the text viewport. This mode is normally used for text manipulation programs that do not use turtle graphics.

RUN MODE

You can get into RUN mode from the BREAK mode by pressing [R]. When RUN mode is entered, the screen is cleared and the turtle appears at the home position. A text window of three lines exists at the bottom of the screen. The user enters turtle graphics commands or calls Super LOGO procedures that have been entered earlier via the EDIT or DOODLE mode. The user can enter any of the following commands directly in RUN mode. Any number of commands may be entered as long as they fit on one line. Once the [ENTER] key is pressed, the commands are executed.

COMMANDS WHICH CAN BE ENTERED DIRECTLY IN RUN MODE

BACK	BACKGROUND	BAUD	CLEAN	CLEAR
CLEARSCREEN	CLEARTEXT	COLORSET	DOT	DRAW
ECHO	FENCE	FORWARD	FULLSCREEN	FULLTEXT
HATCH	HOME	HIDETURTLE	IFFALSE	IFTRUE
LEFT	NOECHO	NOTRACE	NOWRAP	PENCOLOR
PENDOWN	PENERASE	PENUP	PRINT	PRINTSCREEN
REPEAT	RIGHT	SEND	SETHEADING	SETPEN
SETX	SETY	SHOWTURTLE	SLOW	SPLITSCREEN
TEST	TEXT	TRACE	VANISH	WRAP

Some of these commands may be abbreviated. Other SUPER LOGO commands may not be entered in RUN mode; they may only be used within a Super LOGO procedure.

HOW TO EXECUTE A SUPER LOGO PROCEDURE FROM RUN MODE

To run a procedure entered via EDIT or DOODLE mode, enter the name of the procedure. Follow the procedure name with any arguments to be passed to the procedure, then press ENTER. Each argument is preceded by at least one space. An argument can be a number, a variable, a word, a list or an expression. If an expression is used, it must be enclosed in parentheses.

You can interrupt execution of a procedure at any time by pressing the BREAK key. Pressing any key but BREAK causes execution to resume at the point where it paused; pressing BREAK a second time leaves RUN mode and enters BREAK mode.

DOODLE MODE

You can enter DOODLE mode from RUN mode by pressing the @ key. DOODLE mode allows the creation of a turtle graphics procedure that will draw a shape without requiring that the user even know how to read. In DOODLE mode, the screen displays an = sign. The user enters a word (nonsense or otherwise) of at least one letter or number, and presses ENTER. The word is the name of the procedure to be created as a picture is drawn. Now the numeric keys (marked by the special keyboard overlay) can be used to enter turtle graphics commands. Each time a key is pressed, the specified command is executed by the turtle. At the same time, a procedure is created in the program area. This procedure can be viewed in EDIT mode. When entering commands, you can use the left-arrow key (←) to erase the last command. In this case, the entire screen is erased and the shape is re-drawn without the last entered command. To exit the DOODLE mode press BREAK . A procedure created in DOODLE mode can be called out from RUN mode to re-draw the picture again. To do so, just enter the name that was given when DOODLE mode was entered.

The DOODLE mode commands are:

1	CLEAR	2	HOME	3	PU	4	PD	5	RT 45
6	LT 45	7	FD 1	8	FD 10	9	RT 15	0	LT 15

SPECIAL CHARACTERS

The characters [and] are not on the keyboard. However these characters are used in Super LOGO. To enter a [, hold down SHIFT and press 5, followed by SHIFT and then 8. Similarly, to enter], hold down SHIFT and press 5 followed by SHIFT and then 9. The square brackets, [and], must be used in conjuction with entering a **LIST** of words. Another possible use of these characters is in the grouping of statements after an **IF**, **ELSE**, **REPEAT** or **WHILE**. In these cases either parentheses or brackets may be used interchangeably.

To enter a % character, hold down SHIFT and press 5 twice.

SUPER LOGO STATEMENTS AND COMMANDS

CONTROL STATEMENTS

In the statements below, the list of statements referred to may be enclosed in **( )** or **[ ]** symbols. The **( )** symbols provide compatibility with Color LOGO. The symbols **[ ]** provide compatibility with other LOGOs.

END	This is the last statement in a procedure. Execution of an **END** is equivalent to that of the **STOP** statement.
ELSE **(***list of stmts***)**	This statement can appear only after an **IF** statement. If the expression value on the **IF** statement is false, then the list of statements after **ELSE** is executed. Otherwise it is skipped.
FENCE	Same effect as **NOWRAP**.
HATCH *expr procname arglist*	Creates a new turtle. The turtle will start at the same (X, Y) position as its parent (the turtle that **HATCH**ed it) and will be pointed in the same direction. It will have the standard turtle shape. The *expression value* becomes the new turtle's identification number (a number from 1 to 254). The *procname* specifies the procedure to be executed by the new turtle. The *arglist* is optional; it specifies the arguments to be passed to the procedure. The new turtle runs simultaneously with the other active turtles.
IF *expr* **(***list of stmts***)**	The *expression* is evaluated. If the value is true (non-zero), the *list of statements* in parentheses is executed. If it is false (Ø), then the list of statements is skipped. The word **THEN** may be inserted after the expression if desired. The **IF** statement may be followed by an **ELSE** statement. The list of statements denoted here and under **ELSE**, **IFTRUE**, **IFFALSE**, **REPEAT** and **WHILE** can be zero or more statements except the **TO** statement. There may be multiple statements per line, and any number of lines may be used.
IFTRUE **(***list of stmts***)**	If the value of the previous **TEST** statement is **TRUE**, the *list of statements* is executed; otherwise it is skipped.

IFFALSE (*list of stmts*)

If the value of the previous **TEST** statement is **FALSE**, the *list of statements* is executed; otherwise it is skipped.

MAKE :*var expr*
or
MAKE "*var expr*

The value of the *expression* is assigned to the *variable*. The value may be a number, word or list.

NOTRACE

This turns off trace mode and causes normal execution to resume.

NOWRAP

Normally, the screen is in wrap mode. That is, a turtle which runs off the screen will come back on the opposite edge. Execution of the **NOWRAP** statement takes the screen out of wrap mode. If a turtle then runs off the screen, the program will terminate with an **OUT OF BOUNDS** error message.

OUTPUT *expr*

The *expression value* can be a number or word or list. The value is saved as the function result; then control is returned to the calling procedure in the same manner as via the **STOP** statement. The calling procedure can use the saved value via the **RESULT** function.

PRINT *expr [;]*

If the screen is in **FULLTEXT** mode, then **PRINT** acts exactly like **TEXT**, as described below. Otherwise, the expression value is displayed at the turtle location. The turtle is not moved. The value may be a number, word or list. Words in a list are separated by one space.

procname arglist

This is referred to as a **CALL** statement, even though it does not contain the word **CALL**. To **CALL** any procedure, just code its *name* followed by any *arguments* to be passed. If arguments are present, they are separated by one or more spaces. Each argument may be a number, variable, word, list, function reference or an expression contained in parentheses. The argument's values are passed to the parameter variables on the **TO** statement of the called procedure. If there are fewer arguments than there are parameters on the **TO**, then extra parameters are set to Ø.
If the called procedure executes a **STOP**, **OUTPUT** or **END**, then control continues with the next statement after the call statement.

REPEAT *expr*
(*list of stmts***)**

The *expression* is evaluated; if it has a value less than or equal to zero, then the *list of statements* is skipped. Otherwise the list of statements is executed the specified number of times.

SEND *expr expr*

A message is sent to the specified turtle. The *first* expression value denotes the identification of the turtle to which the message is sent. A value of 255 denotes that the message is being sent to the first turtle that requests its mail. Any other value denotes that the message can be received only by a turtle with the specified identification (see also the MAIL function). The value of the *second* expression is the value sent to the other turtle.

SLOW *expr*

The **SLOW** statement causes execution to slow down so that it can be watched more closely. The value of the *expression* denotes how slow to go. A value of 127 is the slowest speed. A value of Ø is full speed.

STOP

This terminates the execution of a procedure. Control is returned to the calling procedure if there is one. If the procedure was called from RUN mode, then control returns to RUN mode. If the procedure was called by a **HATCH** statement, then the associated turtle goes out of existence.

TEST *expr*

The expression is evaluated to **TRUE** or **FALSE**. The result can be used by the **IFTRUE** and **IFFALSE** statements.

TEXT *expr [;]*

The *expression* value is displayed in the text window, which is either the bottom three lines if the screen is not in **FULLTEXT** mode, or the entire screen if it is in **FULLTEXT** mode. If the *optional semicolon* is placed after the expression, then no new line sequence is displayed, and the cursor remains positioned after the last character displayed.

TO *procname parmlist*

This statement defines the start of a Super LOGO procedure. It must start in column 1 of a line and must be the only statement on the line. The *procname* may be any name of one or more letters. The parameters in the *parmlist* may be Ø or more variables. Each one consists of a colon (**:**), followed by any word of one or more letters.

TRACE

This turns on **TRACE** mode. When in trace mode, execution pauses prior to each statement. The statement is displayed, and the user must press the **ENTER** key to procede.

VANISH	**VANISH** takes the current turtle out of existence.
WAIT *expr*	This causes execution to pause for the number of tenths of seconds indicated by the expression.
WHILE *expr* (*list of stmts*)	The *expression* is evaluated; if it is **FALSE** (Ø), then the list of statements is skipped. If it is **TRUE** (non-zero), then the list of statements is executed. After the list is executed, control returns to the **WHILE** again. The expression is then evaluated again. The list of statements is executed repeatedly until the expression is found to be false.
WRAP	Puts the screen back in wrap mode.

TURTLE GRAPHICS AND DISPLAY COMMANDS

STATEMENT	ABBREVIATION	REMARKS
BACK *expr*	**BK**	moves the turtle backward the number of steps denoted by the value of the *expression*. If the turtle's pen is down, then a line of the current pen color is drawn as the turtle moves.
BACKGROUND *expr*	**BG**	sets the background color of the screen to color Ø, 1, 2 or 3. The default background color is 3.
BAUD *expr*		This sets the serial port baud rate at location $ØØ95 in memory. It affects all data sent to the printer. This includes program listings from BREAK mode, and the **PRINTSCREEN** command. The possible values are: (see table below) When the computer is first turned on, the **BAUD** rate is set at 6ØØ baud.
CLEAN		paints the entire display area the background color without moving the turtle.

BAUD RATE	expr VALUE
3ØØ	18Ø
6ØØ	87
12ØØ	41
24ØØ	18

CLEARSCREEN	**CLEAR**	paints the entire display area the background color and moves the current turtle to the home position.
CLEARTEXT	**CT**	Erases the text window.
COLORSET *expr*		selects color set Ø or 1. For each set there are four distinct colors. The default colorset is Ø.
DOT		Draws a one-pixel dot of the current pen color at the current turtle location.
DRAW		Erases the text window and places the screen in **FULLSCREEN** mode.
ECHO		turns on **ECHO** mode. When echo mode is on, then all characters displayed on the screen via **TEXT**, **PRINT** or **REQUEST** commands are also printed on the printer. If the printer is not ready, then data is displayed only, not printed.
FORWARD *expr*	**FD**	moves the turtle forward the number of steps denoted by the value of the *expression*. If the turtle's pen is down, then a line of the current pen color is drawn as the turtle moves.
FULLSCREEN	**FS**	places the screen in **FULLSCREEN** mode. This allows the turtle and lines to be visible on the entire screen.
FULLTEXT	**FT**	places the screen in **FULLTEXT** mode. This disallows the drawing of any turtle graphics and uses the entire screen as a text viewport.
HIDETURTLE	**HT**	makes the turtle invisible.
HOME		sends the current turtle to position (128,96) with heading Ø.
LEFT *expr*	**LT**	turns the turtle left (counter-clockwise) the specified number of degrees.
NOECHO		turns off **ECHO** mode.

PAT

```
................
.....XXXX.......
...XXXXXXXX.....
..XXXXXX..XX....
...XXXXXXXXX....
....XXXXXX......
.....XXX........
....XXXXXX......
...XXXXXXXXXXXX.
..XXXXXXXXX.....
.XXXXXXXXXXXX...
.XXXXXXXXXXXXXX.
.XXXXXXXXXXXXXX.
..XXX......XXX..
..XXX......XXX..
..XXXXX....XXXXX
```

allows selection of the turtle pattern to be used. The pattern is made up of 16 rows of 16 **X** or **.** characters. When the turtle shape is set in this way, the turtle does not visibly rotate on the screen to reflect its current heading, but always displays in the same orientation. The **X** and **.** characters may be arranged as 16 lines of 16, or any other way that adds up to 256 bits. Intervening spaces are ignored.

PENCOLOR *expr* — **PC** — sets the pen color of the current turtle to color Ø, 1, 2 or 3. The default color is Ø. The actual color depends on the current color set. If the pen color is set to the same color as the screen background color, then the turtle pen will erase as it moves.

PENDOWN — **PD** — tells the current turtle to draw a line as it moves in response to **FORWARD** or **BACK** commands.

PENERASE — sets the current pen color to color 3 (background).

PENUP — **PU** — tells the current turtle not to draw a line as it moves in response to **FORWARD** or **BACK** commands.

PRINTSCREEN *expr* — **PS**

causes a printer screen dump to a dot matrix or color printer. The screen dump produces a paper copy of exactly what is shown on the display screen. The expression value should be set depending on the type of printer in use:

1—RS DMP 11Ø single wide or Line Printer 7

2—RS DMP 11Ø double wide

3—RS Color Printer with colors:
- Ø—red
- 1—yellow
- 2—blue
- 3—black

4—RS Color Printer with colors:
- Ø—green
- 1—purple
- 2—orange
- 3—white

On the Color Printer, characters displayed on the screen are not clearly drawn on paper, but have colored ghosts. The appropriate baud rate must first have been set — either by the **BAUD** command, or by setting location $ØØ95 prior to running LOGO. Since **PRINTSCREEN** can take several minutes to complete, you can cancel it by holding down the **[BREAK]** key while the printer is printing. If the printer is not ready when this command is issued, the command is ignored.

RIGHT *expr* — **RT**

turns the turtle right (clockwise) the specified number of degrees.

SETHEADING *expr* — **SETH** and **SH**

points the turtle in the direction specified by the expression. The heading can be from Ø to 359 degrees. Ø degrees is straight up.

SETPEN *state color*

sets pen state (up = Ø or down <> Ø) and pen color.

SETX *expr*	**SX**	moves the turtle by changing its X coordinate to the value specified. No line is drawn. The value may be from Ø (left edge) to 255 (right edge).
SETY *expr*	**SY**	moves the turtle by changing the Y coordinate to the value specified. No line is drawn. The value may be from Ø (bottom) to 191 (top).
SHAPE *shape list*		changes the shape of the current turtle to a shape denoted by the shape list. See TURTLE SHAPE LIST below.
SHOWTURTLE	**ST**	makes the turtle visible.

EXPRESSIONS

The *expr* designation above denotes a place in which an expression can be substituted. An expression can be a number, a variable, a function reference, a word, a list or a combination of these and the operators shown below.

Expressions may contain parentheses to denote the grouping of operations or sub-expressions.

NUMBERS

A number may have a value from -32768 to 32767 plus up to 2 decimal places. A decimal point must be preceded by at least one digit. Examples of valid numbers are:

12　　**Ø.53**　　**12345.67**　　**—99.99**

WORDS

A word consists of a quote character ("), followed by from 1 to 13 letters or digits. Examples of valid words are:

"HAPPY　　**"X**　　**"ABC123DEF456**

VARIABLES

A variable is a word which has a value associated with it. The value may be a word, a number or a list. To refer to the variable name use the notation

"TOTAL

That is, a quote followed by the name. To refer to the value associated with a variable use the notation

:TOTAL or **THING TOTAL**

That is, a colon followed by the name or the word **THING** followed by the name.

If a variable is given on a **TO** statement, then that variable is said to be a *local* variable. That is, each time the procedure is invoked, a new storage location is assigned to the variable. Thus, if a procedure is invoked recursively or by several turtles at once, then each invocation has its own set of local variables which, though they have the same name, are kept distinct. There may be any number of parameters on a **TO** statement; thus there may be any number of local variables in a procedure.

If a variable is referenced in a procedure but is not on the **TO** statement for the procedure, then the variable is said to be a *global* variable. There is only one storage location assigned to each particular global variable. Thus all references to the global variable refer to the same storage location even if the references are in different procedures. This provides a way of sharing information among procedures or among turtles.

LISTS

A list is a value consisting of any number of words stored in some given order. A list with no entries is called a null list. Super LOGO restricts the entries in a list to being words, not other lists. Examples of valid lists are:

[ABLE BAKER CHARLIE DOG] **[THIS IS A LIST]** **[ ]**

ARITHMETIC OPERATORS

These operators result in a number from -32768 to 32767 plus up to two places after the decimal point. Division uses only the integer part of the arguments and produces a result of limited accuracy. The other arithmetic operators act on the integer and fraction and produce more accurate results.

+ addition	— subtraction
* multiplication	/ division

LOGICAL AND RELATIONAL OPERATORS

These operators always result in TRUE or FALSE. Within an expression context a numeric Ø is considered FALSE and all other numbers are considered TRUE.

& logical AND　　　　**!** logical OR

NOT logical negation

RELATIONAL OPERATORS

< less than	> greater than
= equal to	<> not equal to
< = less than or equal to	> = greater than or equal to

For relational operators the arguments may be numbers, words or lists. Two lists are considered equal if they each contain the exact same list of words.

CONCATENATION OPERATOR

arg1 **#** *arg2*

arg1 and *arg2* may be words or lists. The **#** operator combines the two arguments into one list consisting of the elements in *arg1* followed by the elements in *arg2*.

LITERALS

'C — A quote (') followed by one character is called a *literal*. It can be used anywhere a number can be used. The value of a literal is the ASCII value of the character. For example, **'A** is equal to 65. A literal is particularly useful in checking for values returned by the **KEY** function.

FUNCTIONS

ABS *arg* — returns the absolute (positive) value of the argument.

ASCII *arg* — *arg* is a word; this returns the number from Ø to 255 representing the first character of the word.

BUTFIRST *list*
BF *list* — returns a list consisting of all words in the argument list except the first word.

BUTFIRST *word* **BF** *word*	returns a word consisting of all letters in the argument word except the first letter.
BUTLAST *list* **BL** *list*	returns a list consisting of all words in the argument list except the last word.
BUTLAST *word* **BL** *word*	returns a word consisting of all letters in the argument word except the last letter.
BUTTON *arg*	returns Ø if selected paddle button is not depressed, or 1 if it is depressed. *Arg=Ø* selects the right button, *arg=1* selects the left.
CHAR *arg*	arg is a number from Ø to 255; this returns a one-letter word consisting of the selected ASCII character.
COS *arg*	returns the cosine of *arg* degrees
DIFFERENCE *x y*	returns x–y
FIRST *list*	returns the first word in the list.
FIRST *word*	returns a word consisting of the first letter in the argument word.
FPUT *word list*	returns a list consisting of the word followed by all the elements in the list.
HEADING *arg*	returns the heading (Ø to 359) of the turtle with the specified identification (note HEADING ME gives your own direction). If no turtle exists with the identification, then Ø is returned.
INT *arg*	returns the greatest integer less than or equal to the argument value.
KEY	returns Ø if no key is depressed. If a key is depressed, then the value is the ASCII value of the character.
LAST *list*	returns the last word in the list.
LAST *word*	returns a word consisting of the last letter in the argument word.
LIST *a b*	returns a list consisting of the input words
LPUT *word list*	returns a list consisting of all the elements in the list followed by the word.

MAIL *arg* — returns a number value. **MAIL** is used to check for and receive messages sent via the **SEND** command. The argument of the **MAIL** function denotes the source from which messages are to be received. If the argument is 255, then mail is received from any turtle that has sent mail addressed to the current turtle. If the argument is not 255, then it denotes the identification of the turtle from which mail is to be received. If more than one message is available for delivery, then the oldest undelivered message is the one returned. If no messages are available, then a value of zero is returned.

ME — returns the identification of the current turtle. The main turtle is number Ø. The others are numbered from 1 to 254.

NEAR *arg* — returns a measure of the distance from the current turtle to the one with the specified identification. The measure is equal to the number of steps in the X direction plus the number of steps in the Y direction. If no turtle exists with the specified identification, then the distance to **HOME** is measured.

PADDLE *arg* — returns a value from Ø to 63 denoting the position of one of the game paddles (joysticks). The arg is a value from Ø to 3. **PADDLE Ø** gives the up/down of the left paddle. **PADDLE 1** gives the right/left of the left paddle. **PADDLE 2** gives the up/down of the right paddle. And **PADDLE 3** gives the right/left of the right paddle. For up/down, the minimum value is up. For right/left the minimum value is left.

PRODUCT *x y* — returns *x*y*

QUOTIENT *x y* — returns **INT(INT(x) / INT(y))**

RANDOM *arg* — returns a random number from Ø to *arg–1*.

READCHAR
RC — accepts one character from the keyboard and returns a word consisting of that character. The character is not echoed to the screen when it is typed.

READLIST
RL — synonym for **REQUEST**

REQUEST
RQ — accepts a sentence (or list of words separated by spaces) from the terminal and returns the result as a list. The words may be up to 13 characters each. The left arrow key may be used to backup and correct errors only within individual words. The **ENTER** key is used to end the list.

RESULT — returns the value last saved by the **OUTPUT** statement. It will be the value returned by a called procedure.

ROUND *arg*	returns the nearest integer to the *arg* value.
SENTENCE *a b* **SE** *a b*	*a* and *b* represent words or lists. This function returns a list consisting of all elements of the input lists.
SIN *arg*	returns the sine of *arg* degrees
SUM *x y*	returns $x + y$
THING *word*	returns the value associated with the word
WORD *a b*	*a* and *b* must be words. This outputs a word consisting of the characters in *a* followed by the characters in *b*.
XCOR	returns the X coordinate of the current turtle
YCOR	returns the Y coordinate of the current turtle
XLOC *arg*	returns the X coordinate of the turtle with the specified identification (note **XLOC ME** gives your own X coordinate). If no turtle exists with the identification, then 128 is returned.
YLOC *arg*	returns the Y coordinate of the turtle with the specified identification (note **YLOC ME** gives your own Y coordinate). If no turtle exists with the identification, then 92 is returned.

TURTLE SHAPE LIST

The **SHAPE** statement is used to assign a new shape to the current turtle. The shape of a turtle is made up of a pattern of dots on a grid. The shape list tells Super LOGO how to draw the turtle pattern. The turtle shape is automatically rotated to face in the direction the turtle is headed. Drawing the turtle shape is similar to using normal turtle graphics commands to draw any shape. The difference is that the commands which make up the shape list are a restricted and simplified form of the normal turtle graphics commands. The commands allow a step of one pixel (one square on a piece of graph paper) in any of the 8 possible directions. The 8 directions are: up, down, right, left, and the four diagonal directions. The one-letter commands that may be used in a shape list are shown below. The shape list can be any length. If it runs over a line boundary, put a hyphen (—) at the end of the line, then continue in column 1 of the next line. The turtle shape drawing pen complements the affected pixels. That is, the complement of a dot present is no dot present and vice versa. This allows a turtle to pass over a picture without destroying the picture.

TURTLE SHAPE COMMAND MEANING

F step forward one pixel; if the pen is down, complement the pixel.

B step backward one pixel; if the pen is down, complement the pixel.

R rotate right by 45 degrees.

L rotate left by 45 degrees.

U pick up the turtle shape pen; this pen is always assumed down at the start of a shape list. This pen should not be confused with the turtle pen that draws when a **FORWARD** or **BACK** turtle graphics statement is executed.

D put the turtle shape pen down; if the pen was previously up, then putting it down will cause the current pixel to be complemented.

MULTIPLE TURTLES

Normally, one turtle exists. The user can create additional turtles by using the **HATCH** statement. Each turtle then runs its procedures independently of the other turtles. The **HATCH** statement assigns an identification number to each turtle. That number may be used by other turtles to send mail or request location information about the turtle. The main turtle is always number Ø. Other turtles can have a number from 1 to 254. When a turtle other than the main one exits from the procedure given it when it was **HATCH**ed, it goes out of existence, leaving behind only the lines it drew on the screen. The **VANISH** statement also causes the turtle to go out of existence. The main turtle, in contrast, can only go out of existence by executing a **VANISH** statement.

If the main turtle exits from the procedure given to it from RUN mode, it will return to RUN mode where the user can then enter its next command or procedure to run. If when the main turtle is in RUN mode there are other turtles, then the other turtles cease to move. Each time the **ENTER** key is pressed, each of the turtles executes one program statement. This has the effect of stepping the hatched turtles along at a controlled pace. A useful debugging method is to **HATCH** a turtle from RUN mode and tell it to run the procedure which is to be tested. Then the procedure is run by pressing **ENTER** repeatedly. If you enter a **VANISH** command, then the main turtle will disappear and the hatched turtle will run at full speed.

ERROR MESSAGES FROM SUPER LOGO

In BREAK mode a **?** is printed if any key other than a valid command letter is pressed. A load or save command may also print a digit followed by a ?. These messages are:

1? memory error

2? tape checksum error (probably a bad tape or the volume not set correctly)

3? attempt to load a tape that is not a Super LOGO program

4? attempt to load a module that is too long for memory

In RUN mode there are several possible messages that may be issued. These messages attempt to identify the error in the program, but remember that the message is only a guess as to what is wrong. It is possible that the message does not exactly fit the problem. If the statement in error is from within a procedure then the line in error is displayed after the error message. After one of the following messages is displayed, the user must press a key to continue.

MESSAGE	PROBABLE MEANING
I DON'T KNOW HOW TO ...	"..." is filled in with the name of what Super LOGO thought was a procedure name to call; but the procedure name is not found in the program area. If the name is one which should be in the program area, make sure that it is preceded by **TO** — not **TØ** (zero). Also make sure **TO** is in column 1. Check also that the name is correctly spelled. If the name was not supposed to be a procedure, then probably there is something wrong with the immediately preceding command.

I CAN'T FIGURE OUT ...

"..." is filled in with the word that caused the confusion. Super LOGO was attempting to compute the value of an expression when it encountered the problem. Possibly the syntax of the expression is in error; or a colon is left out before a variable name; or a function name is misspelled.

I DON'T KNOW HOW MUCH

This message means that a command such as **RIGHT** or **FORWARD** which should be followed by a number is not followed by a number. Either an expression is not present where one should be, or the very first item in the expression is not valid.

"(" OR ")" NOT RIGHT

A left parenthesis is not found as expected after an **IF**, **WHILE** or **REPEAT** expression or after an **ELSE**. Or, unbalanced parentheses are detected.

I CAN'T DO THAT IN THIS MODE

A command other than one of the ones allowed (for example, **WHILE**), is entered directly from the keyboard in RUN mode. Remember, some commands may be executed only within a Super LOGO procedure.

MY MEMORY IS TOO FULL

The internal program and work area is filled. This will always happen eventually if a program is allowed to do infinite recursion (call itself repeatedly forever). In general, procedure calls, hatching turtles and sending messages consume memory. The longer the text in the program area, the less available memory for these operations.

OUT OF BOUNDS

The screen has been placed in **NOWRAP** mode and a turtle has run off the boundaries of the screen.

I EXPECTED A ... HERE

"..." is **WORD**, **LIST** or **NUMBER** and this indicates that the improper type of data was given to a function. For example, **SENTENCE** expects words or lists for arguments, not numbers.

IMPROPER LIST

A list within the program does not end before the end of the line, or it contains a "**[**".

INDEX

A

@, at end of line 43
@, as literal character 43
ABS 112, 177
adding procedures 42
address 99
alphabetize 131
all points bulletin 99
AND 177
animation 87, 91
append 42, 162
ARC 38
argument 111
arithmetic 31
arithmetic drill 118
arithmetic operators 176
arrow keys 163
ASCII 98, 101, 129, 177
assignment 67

B

BACK 9, 171
backspace 56
BACKGROUND 33, 171
BAUD 42, 171
baud rate 42
baud default 42
BF 108, 177
BG 33, 171
BK 9, 171
BL 108, 178
blackjack 140
bottom-up 26
break line 17
BREAK Mode 6, 161
BUTFIRST 108, 109, 177
BUTLAST 108, 109, 178
BUTTON 178

C

call 169
camera settings 43
cassette 41
cassette volume 41
CHAR 101, 119, 129, 178
circle 12, 38
CIRCLE 27
CLEAN 171
clear 7, 172
clear memory 19
CLEARSCREEN 172
CLEARTEXT 172
colon 29, 176
color 33
color printer 44
COLORSET 33, 172
combinations of letters 147
combining procedures 42
commands, RUN Mode 166
commands, single key 55
COMPARE 130
compare word 132
complement 91
concatenation 101, 119, 177
condition 80
control statements 11, 79, 168
convert number to word 129
COS 102, 178
count words 124
cursive 72
cursor 15
cursor position 16
CT 172

D

DECK 135
delete character 16, 163
delete to end of line 75, 164
delete line 41, 75, 164, 184
DIAMOND 24
dice games 149
DIFFERENCE 149, 179
DOODLE mode 55, 167
DOODLE mode used to design turtle shapes 89
DOT 38, 172
double space 43
DRAW 25, 165, 172
DSKINI 41, 162
dynaturtles 102

E

ECHO 44, 172
edit, DOODLE Mode 56
EDIT Mode 15, 163
editor summary 17
ELSE 83, 168
empty word 109
end-loop recursion 45
END 19, 47, 168
erase, DOODLE Mode 56
error messages 182
exchange letters 145
EXP 113
exposure 43
expression 31, 175

F

fast search 75
FD 6, 172
FENCE 168
Fibonacci series 129
film 43
find 75, 164
find again 75, 164
FIRST 108, 178
format 184
FORWARD 6, 172
FPUT 110, 178
fractal 50
FRACTAL 51
FS 25, 172
FT 172
full screen 25

FULLSCREEN 25, 165, 172
FULLTEXT 108, 166, 172
functions 112, 177

G

game controller 102
games 97
games, dice 149
general message 99
global find 164
global variable 30, 176

H

HATCH 77, 168
heading 37
HEADING 102, 178
HIDETURTLE 9, 172
Hilbert curve 53
Hofstader 45
home 37, 69, 165
HOME 38, 172
horizontal size 11
HT 10, 172
hyphen 27, 88

I

IF 50, 168
IFFALSE 118, 169
IFTRUE 118, 169
indentation 21
INSERT 131
insert character 16, 164
insert line 17, 163
INT 129, 178
interrupt 86

J-K-L

KEY 98, 178
keys, **DOODLE** mode 55
LAST 108, 178
LEFT 9, 172
LENGTH 124
length, list 120
lens opening 43
level 25, 47
line feed 43, 162
LIST 178
lists 107, 176
list length 120
literal 98, 177
load 41, 162
local variable 30, 176
logical operators 177
LPUT 110, 119, 178
LT 9

M

MAIL 100, 179
MAIL 255 101
mailbox 99
MAKE 67, 114, 169
marker 42, 163
master procedure 25
master turtle 77
ME 82
MEMBER 125
memory full 45
METAMORPHIZE 126
merge 42, 162
message 99
mode 15, 161
mode map 15
multiple turtles 77, 181
multiprogramming 77
multitasking 77, 181

N

NEAR 80, 179
nesting 31, 100
Newton's Law 102
NOECHO 44, 172
NOT 177
NOTRACE 169
NOWRAP 45, 169
number to word 129
numbers, range 175

O

OK Set 59
One-Key Set 59
OR 177
order of operations 86
OUTPUT 112, 123, 169
overlay 55
overtype 16

P

PADDLE 103, 179
paddle sensitivity 103
partial save 43
PAT 95, 173
pause 86
PC 33, 173
PD 37, 173
PENCOLOR 33, 173
PENDOWN 37, 173
PENERASE 173
PENUP 37, 173
permutations of letters 147
pictures 43
PICK 123
PICKRANDOM 124
pig latin 125
photographs 43
polygons 11
POLYSPI 45
power 113
primitives 164
PRINT 60, 101, 109, 169
printer 43, 162
printers 43
printing text 44
PRINTSCREEN 44, 175
procedure names 19
PRODUCT 179
PS 43, 175
PU 37, 173

Q

queue 101
quote mark 176
QUOTIENT 179

R

RANDOM 85, 179
random sentences 127
random words 123
READCHAR 117, 179
READLIST 120, 179
RECTANGLE 19
recursion 45
RC 117, 179
relational operators 177
REMOVE 150
REPEAT 11, 170
REQUEST 120, 179
RESET 6
RESULT 112, 179
return results 111
RIGHT 8, 175
RL 120, 179
ROUND 180
RQ 120, 179
RT 8, 175
RUN Mode 15, 19, 166

S

save 41, 162
save, partial 42
scale 11
scan 17
screen coordinates 165
screen dimensions 165
screen dump 43
scroll 17, 25, 75, 164
scrunch 11
SE 110, 180
search 75, 164
search string 75
SEND 99, 170
SEND 255 101
SEND, words 137
SENTENCE 110, 180
SETH 37, 175
SETHEADING 37, 175
SETPEN 175
SETX 37, 175
SETY 37, 175
SH 37, 175
SHAPE 87, 175, 181
SHIFT up-arrow, in EDIT Mode 75
SHOWTURTLE 10, 175
SHUFFLE 135
shutter speed 43
SIN 102, 180
single key commands 55
  in DOODLE mode 167
  in EDIT Mode 164
single space 43
single-step 49
SLOW 78, 170
sorting 130
spaces 101
split line 163
split screen 7, 25, 165
square brackets 109, 167
square root 113
ST 10, 175
start marker 42, 163
starting Logo 161
stick figure 90
STOP 48, 170
stop marker 42, 163
stopping procedures 86
store 41
subprocedures 23, 112
SUM 180
switch letters 145
SX 37, 175
SY 37, 175
synchronize 93

T

TEST 118, 170
TEXT 170
text, printing 43
text window 25, 166
THING 176, 180
thrust 102
TO 170
top-down 26
top-level 26
TRACE 50, 170
transform list 126
TREE 47, 81
truth value 48
turtle heading 9
turtle name 77
turtle number 77
turtle shape 87

U-V

VANISH 79, 171
variable 29, 176
variable, global 30, 176
variable, local 30, 176
variable names 67
vertical size 11
volume, on cassette recorder 41

W

WAIT 70
WALK 92
WAR 136
WHILE 80
WORD 109, 180
word search 150
words 107, 175
WRAP 46
wrap-around 7

X-Y-Z

X, Y proportions 11
XCOR 60, 180
XLOC 85, 180
YCOR 60, 180
YLOC 85, 180

RADIO SHACK, A DIVISION OF TANDY CORPORATION

U.S.A.: FORT WORTH, TEXAS 76102
CANADA: BARRIE, ONTARIO L4M 4W5

TANDY CORPORATION

AUSTRALIA	BELGIUM	U. K.
91 KURRAJONG AVENUE MOUNT DRUITT, N.S.W. 2770	PARC INDUSTRIEL 5140 NANINNE (NAMUR)	BILSTON ROAD WEDNESBURY WEST MIDLANDS WS10 7JN

7/85-TM

Printed in U.S.A.