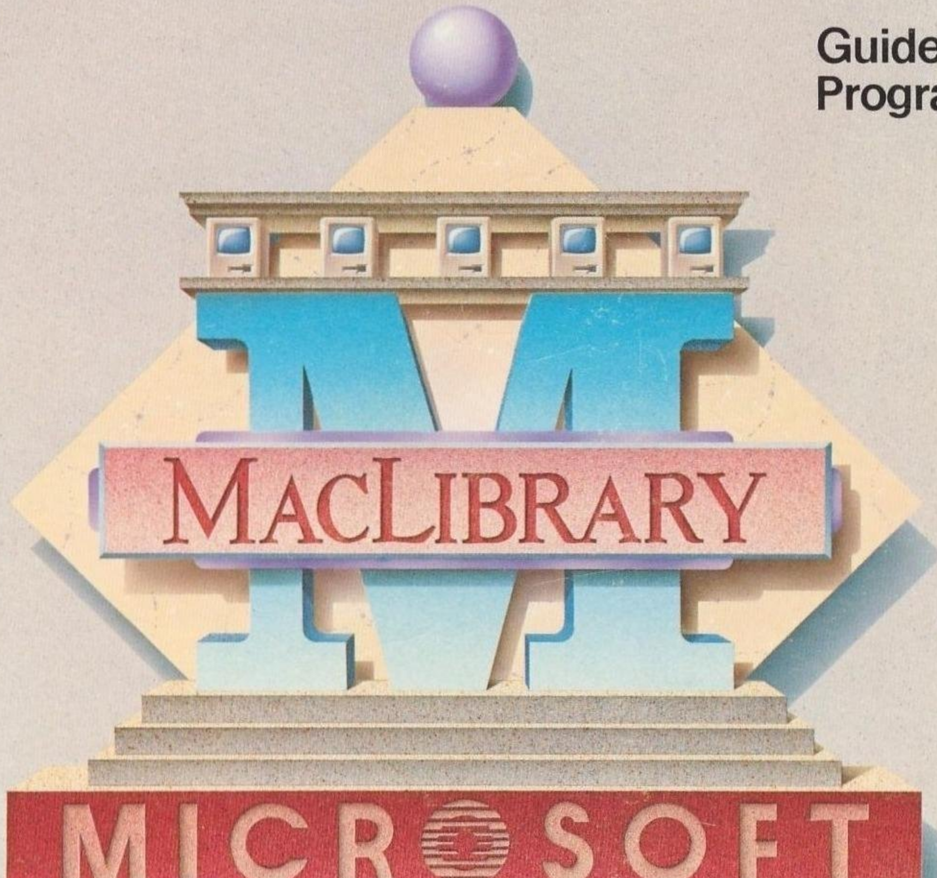
Logo

LOGO

by Logo Computer Systems Inc.

Guide to Programming



Microsoft® MacLibrary™ Software Series for Apple® Macintosh™

Logo

Guide to Programming

by Logo Computer Systems Inc.

version 1.0
for Apple® Macintosh™

Published by the Microsoft® MacLibrary™
Software Series

Microsoft Corporation

If you have comments about this documentation or the enclosed software, complete the form at the back of this manual and return it to Microsoft MacLibrary.

Document number: 690080-001-00

Acknowledgements

Authors
Eric Brown
Sharnee Chait

Contributing Editor
Seymour Papert

Graphic Design and Layout
Lorraine Lavigne
Richard Lavigne
Julien Perron

Founded in 1980, Logo Computer Systems Inc. is the world's leading developer of Logo, having developed the Logo language for a wide range of microcomputers. Logo Computer Systems Inc. has made the Logo language international, with translations into many languages. Noted for superior documentation and software design, the products of Logo Computer Systems Inc. have become the world standard.

Contents

Introduction ix

What You Need x

Before You Begin xi

1 Getting Started 1

Starting Up 1

Typing Instructions 3

Getting Help From Logo 6

Exploring Further: A Demonstration 6

2 Communicating With Logo 7

Action 7
- Controlling the Turtle 7
- Fixing Typing Mistakes 10
- Printing Text on the Screen 11
- Using the Repeat Command 11
- Filling Shapes With Patterns 12
- Opening Windows 13
- Calculating With Logo 16

Reflection 17
- Turtle Geometry 17
- Inputs 17
- Bugs 17

Exploring Further 18

Logo Vocabulary 18

3 Defining Procedures and Using Subprocedures 21

Action 21
- Defining Procedures 21
- Fixing Bugs in a Procedure 25
- Using Cut and Paste to Edit 26
- Drawing a Starry Sky 26
- Writing a Superprocedure 28
- Printing Your Pictures 29

Reflection 29
- Naming 29
- More Turtle Geometry 30
- Superprocedures and Subprocedures 30
- Commands and Operations 30

Exploring Further 31

Logo Vocabulary 32

4 Examining Your Workspace and Saving Files 33

Action 33
- Examining Your Workspace 33
- Erasing From the Workspace 34
- Saving Your Workspace 35
- Listing Files 35
- Clearing Your Workspace 36
- Loading Files 37
- Erasing Files 37
- Saving and Loading Windows 37

Reflection 39
- Distinguishing Workspace From File Space 39
- Naming Files 39

Logo Vocabulary 39

5 Using Variables 41

Action 42
- Defining Procedures With Inputs 42
- Checking for Possible Bugs 43
- Defining a Text Procedure With Inputs 44
- Creating a Variable Sized Star 44

Reflection 47

Exploring Further 47

Logo Vocabulary 48

6 Drawing Polygons and Spirals 49

Action 49
  Drawing Polygons 49
  Defining a Sun 51
  Drawing Spirals 52

Reflection 54
  Experimenting 54
  Total Turtle Trip Revised 55
  Recursion 55

Exploring Further 56

Logo Vocabulary 56

7 Exploring Recursive Procedures 57

Action 57
  Creating Stop Rules 57
  Writing a Stop Rule for Spi 58
  Writing a Stop Rule for Poly 58
  Writing a Stop Rule for Words and Lists 60
  Adding Instructions After the Recursive Line 62

Reflection 63
  Conditions, Actions, and Predicates 63
  Recursion With Words and Graphics 64
  Thinking About Recursion 65

Exploring Further 66

Logo Vocabulary 66

8 Creating a Bar Graph Project 67

The Scenario 67

The Plan 70

Action 70
  Step 1: Setting Up the Windows 70
  Step 2: Drawing the Axes and the Bars 71
  Step 3: Determining the Bar Height 74
  Step 4: Labelling the Graph and the Bars 77
  Step 5: Writing the Superprocedure 80
  Step 6: Setting Up the Initial Windows 81
  Program Listing 83
  Program Structure of BarGraph 85

Reflection 85
  Operations 85
  Some Notes on ReadWord 85
  Window Coordinates and Screen Coordinates 86

Exploring Further 87

Logo Vocabulary 87

9 Manipulating Text 89

Action 90
  Generating Random Sentences 90
  Step 1: Creating Lists and Picking a Random Word 90
  Step 2: Writing the Sentence Generator 92
  Step 3: Extending the Sentence Generator 93
  Generating a "Dialect" 94
  Step 1: Examining and Replacing Part of a Word 94
  Step 2: Writing a Superprocedure to Replace Words in a List 96

Reflection 97
  Global Variables 97
  Operations Written in Logo 97
  Recursive Operations 98

Exploring Further 99

Logo Vocabulary 99

10 Building a Phone Directory 101

Action 102
  Step 1: Entering the Data 102
  Step 2: Printing Out the Phone List 103
  Step 3: Adding and Changing Listings in the Phone Directory 106
  Program Listing 107
  Program Structure of PhoneList 108

Reflection 109
  The Elements of a List 109
  Replacing an Element in a Property List 109

Exploring Further 110

Logo Vocabulary 110

A Concluding Note by Seymour Papert 111

Other Books About Logo 113

Index 115

Introduction

Logo is a language for computers and people. Using Logo, beginners can get dramatic and interesting results quickly. Experienced programmers will find rich material in Logo with which to develop their skills.

Use the guide

The *Guide to Programming* is an introduction to Logo programming. It is intended for both new computer users and people who already know about computers. This guide shows you how to build and change programs, store and retrieve your work, and also provides examples of Logo programs that you can write.

Most chapters in this guide are divided into the following sections: "Action", "Reflection", "Exploring Further", and "Logo Vocabulary". "Action" introduces you to important *primitive procedures* – the basic words of Logo's vocabulary – and provides sample programs to work on at the computer. "Reflection" gives you additional information on related Logo concepts that you can read when you want to take a break from programming. "Exploring Further" suggests activities to try on your own. "Logo Vocabulary" lists the primitive procedure names, menu items, and special characters and keys introduced in each chapter.

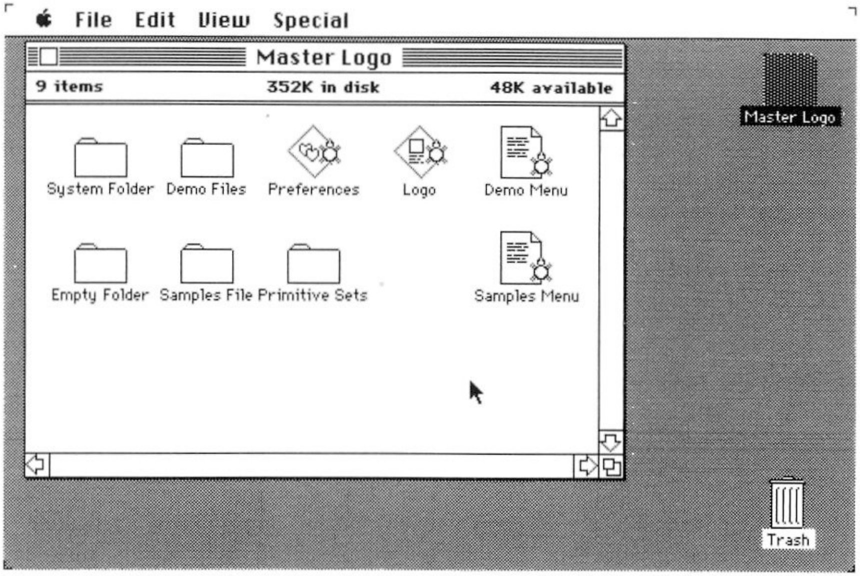
Don't worry if some new concepts seem unclear to you when they are first introduced. As you become familiar with Logo by working through the guide, these concepts will be clarified.

Note When interaction with Logo is shown in the guide, red text represents what you type on the computer. Black text represents what the computer displays.

Use the *Reference Manual*

The *Reference Manual* works with the guide to expand your understanding of Logo. It provides a complete description of the Logo language and should be used for reference purposes, not as a guide for new users.

The Master Logo disk contains:



Examine demonstration programs

In addition to the Logo program, there are three files that will be of interest to you. Choose the Demo Menu file from the Finder for a demonstration of some of the powerful effects you can produce with Logo. Choose the Samples Menu file for ideas of the different kinds of Logo programs that you can write; more complex programs are included in this file. Choose the Exploring Further file for suggested program listings of the sample graphics found in each "Exploring Further" section of the guide.

What You Need

To use Logo, you need an Apple® Macintosh™ computer with 128K or 512K of memory and a disk drive, or an Apple® Lisa™ with MacWorks™. Any options you may have can be useful:

Optional hardware

- a second disk drive
- a printer
- a modem
- anything that plugs into the serial input-output plugs

Note All the procedures and examples in this book work with the original Logo product that arrives in the package. But Logo can be easily customized and certain commands removed. So, if you are not the first person to use the Logo disk, don't be upset if the graphics instructions, for example, don't work. They are easy to restore in Logo. Refer to Appendix C, "Using the Preferences Program", of the *Reference Manual.*

Before You Begin

Make at least one copy of the master disk by moving the Master Logo disk icon over the icon for the other disk. See *Macintosh,* your owner's guide, for details.

We assume that you have read *Macintosh* and understand the basic Macintosh terminology: "clicking", "dragging", "selecting" and the use of the Menu bar. If you are not familiar with these terms, take time to review them now.

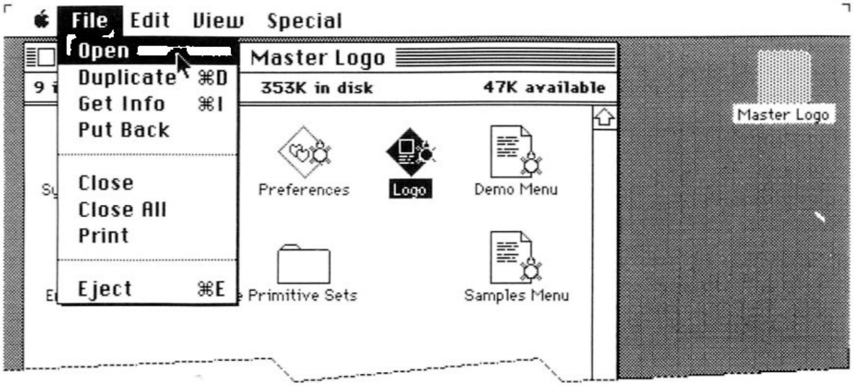
1 Getting Started

This chapter tells you how to start up Logo and type in instructions.

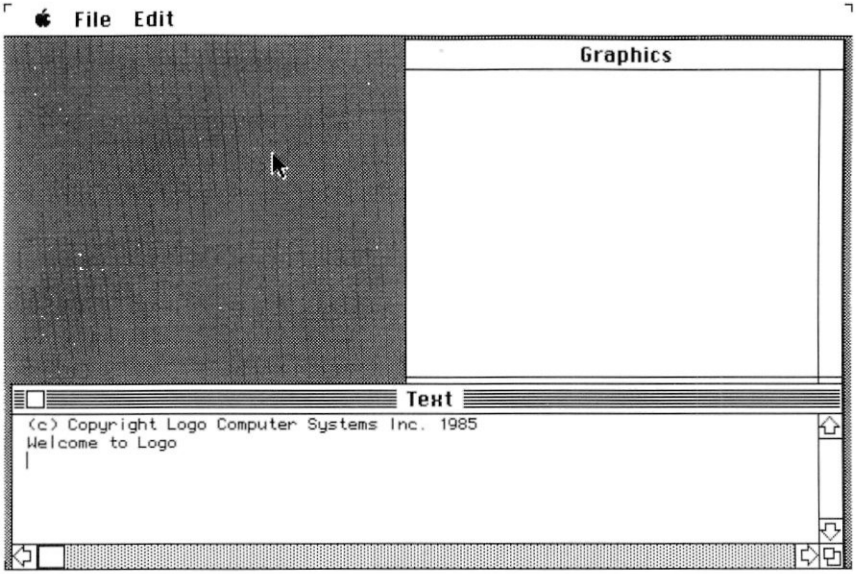
Starting Up

Load Logo

1. Insert the Logo disk into the drive.
2. Turn on the computer.
3. Open the Logo file from the Finder.



Logo is loaded when you see this on the screen:



```
(c) Copyright Logo Computer Systems Inc. 1985
Welcome to Logo
|
```

Two windows are displayed on the screen: one called Graphics and one called Text. What you type appears in the text window. What you draw appears in the graphics window.

Move the insertion point

The flashing line on the text window is called the *insertion point.* This is where what you type will appear. You can move the insertion point to the next line by pressing the Return key, or to any location in the window by moving the mouse pointer and clicking the button.

Typing Instructions

Type a line. With the insertion point still on the line, press the Enter key. Logo will treat the line as an instruction, and try to perform, or "run" it.

For instance, type:

`hello` Press the Enter key.

Logo responds:

```
I don't know how to hello
```

Logo is correct. Hello isn't in Logo's initial vocabulary, but that's not important. You can't hurt Logo by experimenting.

Type:

`print [Greetings!]` Press Enter.

Logo responds:

```
Greetings!
```

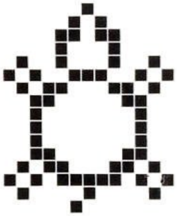
If nothing happened, you may have pressed the Return key, not the Enter key. There is a big difference between these keys.

The Return key signals the end of a line, and moves the insertion point to the next line.

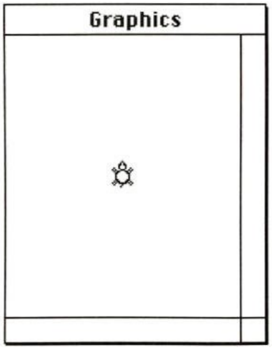
The Enter key takes the current line, or a selected area of text, and gives the text to Logo as instructions to run. Do It in the Edit menu has exactly the same function.

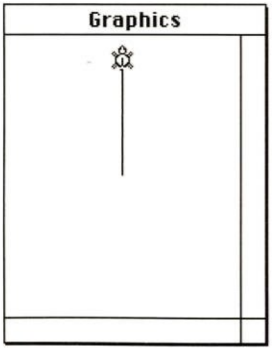
Now try some graphics commands:

Meet the turtle



`showturtle`	Press Enter.
`forward 70`	Press Enter.



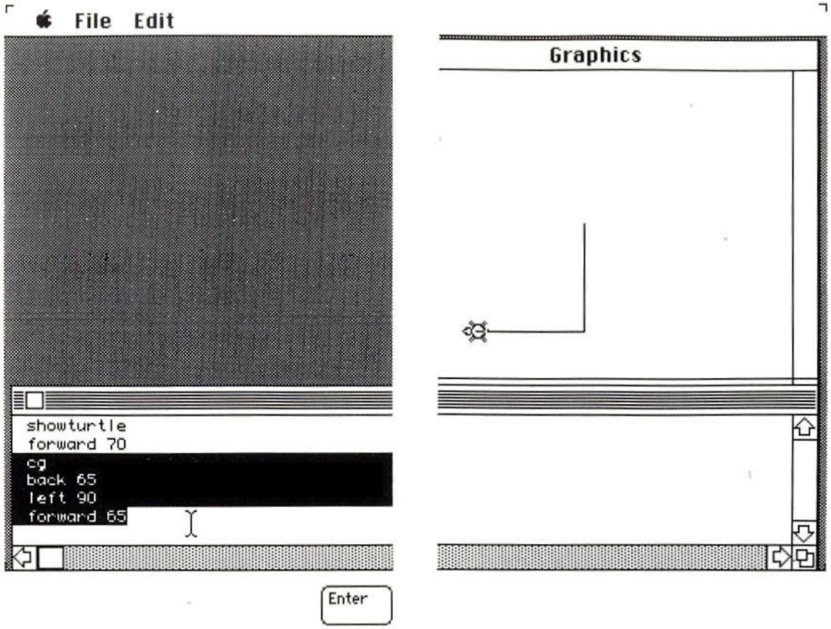


You can type several instructions at one time using the Return key to separate them:

`cg`	Press Return.
`back 65`	Press Return.
`left 90`	Press Return.
`forward 65`	Press Return.

Select those four lines with the mouse. (Point to just before *CG*, click and hold the button down while you drag. The text will be highlighted as you select it.)

With the lines selected, press Enter. You should see:



```
showturtle
forward 70
cg
back 65
left 90
forward 65
```

Remember the Enter and Return keys

Remember:

- The Return key moves the insertion point to the next line
- The Enter key tells Logo to run the current line or selection

Now clear the graphics window by typing:

```
cg
```

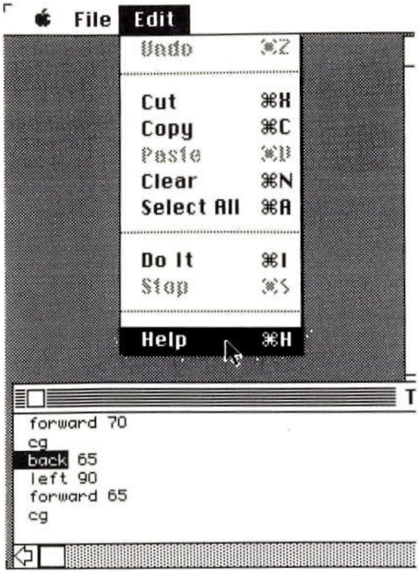
Press Enter.

Getting Help From Logo

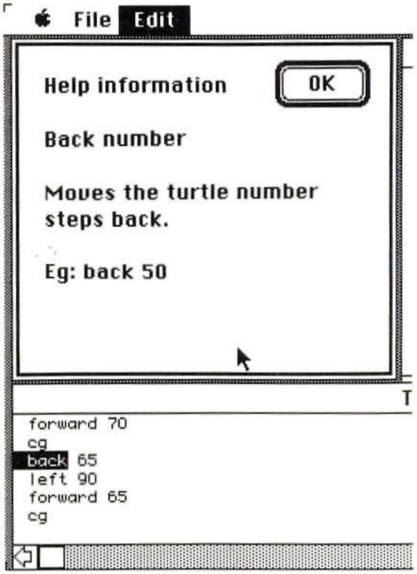
The Help item in the Edit menu provides information about procedure names.

Choose Help

To use Help, simply select the procedure name on the text window, then choose Help. A box will appear with the Help information. When you've read the information, click the OK button in the corner.



```
forward 70
cg
back 65
left 90
forward 65
cg
```



Help information

Back number

Moves the turtle number steps back.

Eg: back 50

```
forward 70
cg
back 65
left 90
forward 65
cg
```

Exploring Further: A Demonstration

You've just taken your first steps into the Logo world. This is a good time to look at where those steps can lead. To see the demonstration program:

See a demonstration

1. Choose Load from the File menu.
2. Click Workspace.
3. Select the file named Demo Menu.
4. Click Load.
5. Follow the instructions that appear on the screen.

2 Communicating With Logo

To communicate with Logo, you type instructions. As you saw in Chapter 1, "Getting Started", Logo responds to instructions by producing effects on the screen.

This chapter introduces graphics commands, most of which control a computer creature called a turtle. Graphics is a good context in which to start learning Logo since you can *see* how your instructions work. You will also learn to open new windows and begin doing arithmetic.

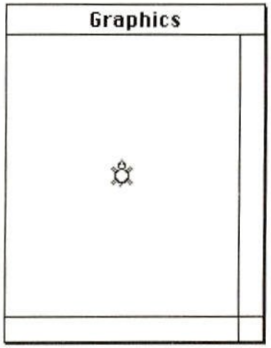
Action

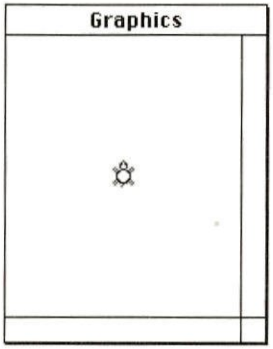
Controlling the Turtle

To see the turtle on the graphics window, use the command ShowTurtle (or ST for short). The command HideTurtle (or HT) makes the turtle invisible. Type:

Show the turtle

`showturtle`	Press Enter.
`hideturtle`	Press Enter.
`st`	Press Enter.





Move the turtle

Now, move the turtle with the Forward (Fd) command. Forward needs an *input* – a number indicating how many steps the turtle is to move. Try:

```
forward 50
```

Press Enter.

Notice that the turtle changed its position, but its heading (the direction it was facing) remained the same.

Change the turtle's heading

To change the turtle's heading, you can use the command Right (Rt) or Left (Lt). Like Forward, the Right and Left commands each need an input – a number indicating how many degrees the turtle is to turn. Type:

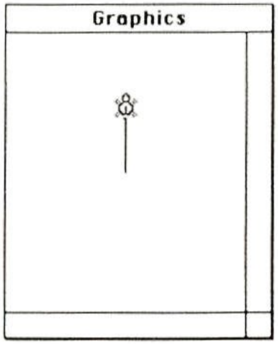
```
right 90
```

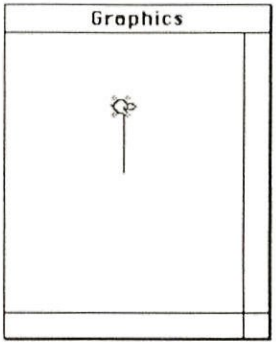
Press Enter.

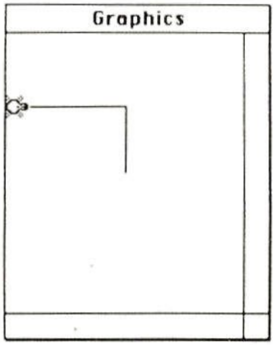
The turtle turned 90 degrees to the right of its previous heading. Notice that the turtle changed its heading, but *not* its position on the screen.

Back (Bk), like Forward, moves the turtle away from its current position without changing its heading. For example:

```
back 80
```







Left turns the turtle left:

```
left 45
```

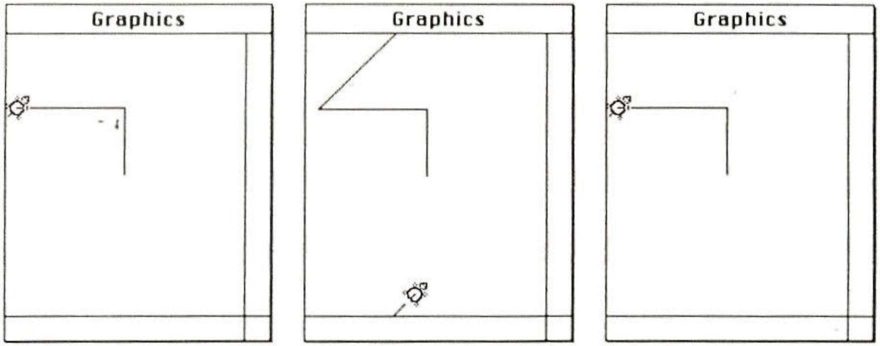
The turtle turned 45 degrees to the left. Its heading changed, but not its position. Move the turtle forward to see the effect of the turn:

```
forward 100
```

Erase a line

If you don't like the length of that last line, erase it by tracing over the line with PenErase (PE).

`penerase`	Press Enter.
`back 100`	Press Enter.



Use PenDown (PD) to put the drawing pen down. Otherwise, the turtle will continue to erase any lines it passes over.

`pendown`	Press Enter.
`forward 50`	Press Enter.

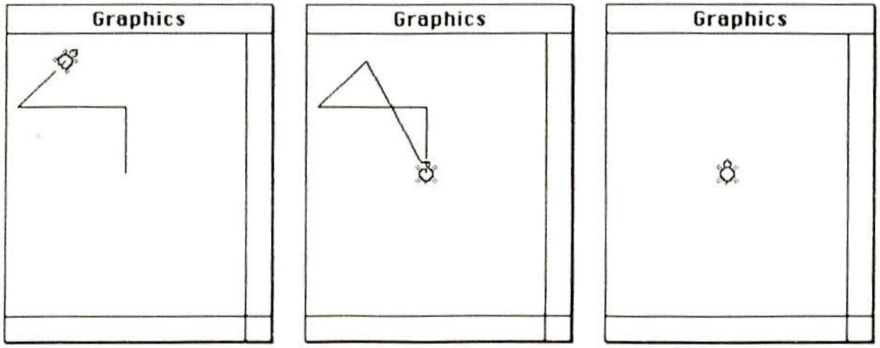
Home sends the turtle back to the center of the window, pointing straight up.

```
home
```

Clear graphics and text

At this point, you may want to clear the lines from the graphics window and experiment with the commands you have learned. CG (which stands for Clear Graphics) erases all the lines on the graphics window:

```
cg
```



To clear the text from the text window, use CT (for Clear Text):

```
ct
```

Choosing Clear from the Edit menu will also clear the text.

Fixing Typing Mistakes

If you make typing mistakes, Logo won't understand your instructions and will print a message to tell you so. For instance, if you type:

```
frward 100
```

and then press the Enter key, Logo will respond:

```
I don't know how to frward
```

Spaces between a command and its input are very important. If you forget a space, Logo won't understand your instructions. For example, if you type:

```
right90
```

Logo will respond:

```
I don't know how to right90
```

Logo interpreted right90 as one word, and printed a message indicating its incomprehension.

Correct mistakes with the Backspace key

If you've made a typing error, use the Backspace key to erase the error and retype to correct it.

In general, editing text in a text window is the same as editing text in a Macintosh word processor or other text program. The mouse is used to move the insertion point or to select, the Backspace key is used to erase, and Cut, Copy, and Paste in the Edit menu are used to move blocks of text around.

Printing Text on the Screen

Print text

Print (or Pr for short) is the command that prints text. Try:

`print 5`	Prints a number.
`print "Hello`	Prints a word.
`print [Tom Jerry Seymour]`	Prints a list.

Notice that if Print's input is a word, the word must be preceded by a quotation mark. If Print's input is a list (a group of words), the list must be enclosed in brackets.

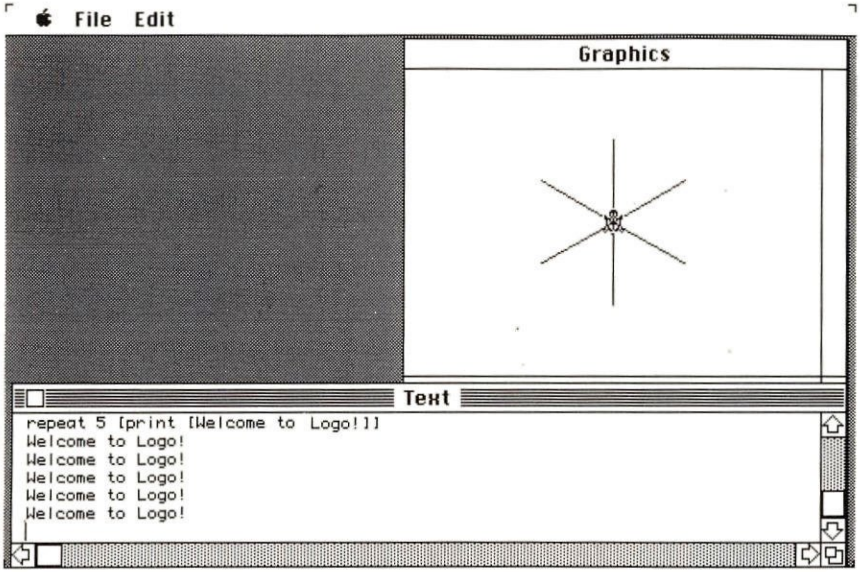
Using the Repeat Command

Run a list of instructions

The Repeat command takes a list of Logo instructions, and runs them again and again, as if they had been entered separately.

Try:

```
repeat 6 [forward 50 back 50 right 60]
repeat 5 [print [Welcome to Logo!]]
```



```
repeat 5 [print [Welcome to Logo!]]
Welcome to Logo!
Welcome to Logo!
Welcome to Logo!
Welcome to Logo!
Welcome to Logo!
```

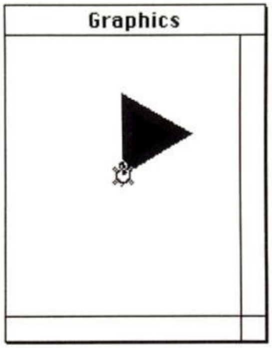
Remember, Repeat's first input is the number of repetitions. Repeat's second input is an instruction list enclosed in brackets.

Filling Shapes With Patterns

There are many commands you can use to add "special effects" to the drawings you create. FillSh (for Fill Shape) and SetPPattern (for Set Pen Pattern) are two such commands. FillSh takes an instruction list as its input (like Repeat). It fills in the shapes created by the instructions in the instruction list. To get a more textured effect, you can give the pen a pattern to draw with. SetPPattern sets the pen's pattern. For example:

Fill in a shape

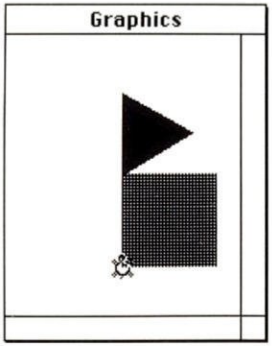
```
cg
fillsh [repeat 3 [fd 70 rt 120]]
```



Change the pen's pattern

```
pu bk 80 pd
setppattern 4
fillsh [repeat 4 [fd 80 rt 90]]
```

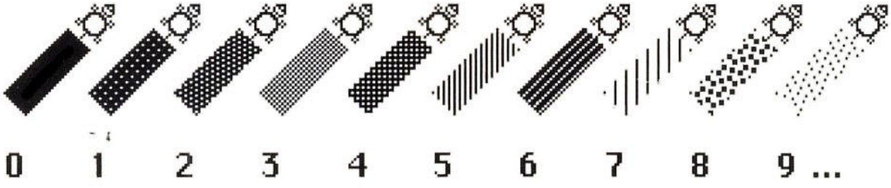
The number selects a particular pattern.



Restore the pen's pattern

`setppattern 0` The pen pattern is back to a solid line.

SetPPattern's input is a number representing a pattern. Here are some patterns to experiment with:



(See Chapter 10, "Graphics", in the *Reference Manual* for a complete list of the pen patterns.)

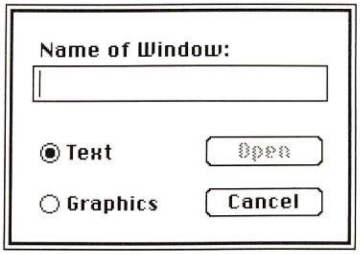
Opening Windows

When you first start Logo, there is one graphics window and one text window. Any window may be opened and closed using the menu bar, and named anything you wish... as long as no other window exists by that name.

Open a new window

To open a new text window:

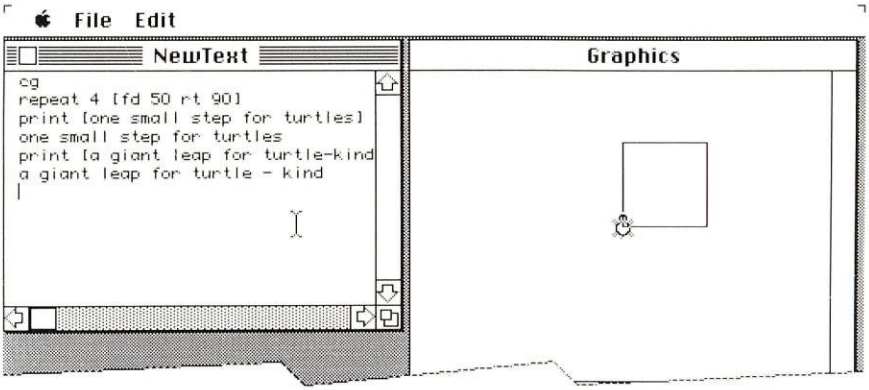
1. Choose Open Window from the File menu. A dialog box appears.
2. Select "Text" as the kind of window and enter any name you like.
3. Click the Open button.



In the top left corner of the screen a new window appears. Click the window, then place the pointer on the bottom right corner. With the button pressed, drag the corner to make the window any size you like. Don't cover the graphics window – you'll erase your drawings if you do.

In the new window, try a few Logo commands:

```
cg
repeat 4 [fd 50 rt 90]
print [one small step for turtles]
print [a giant leap for turtle-kind]
```



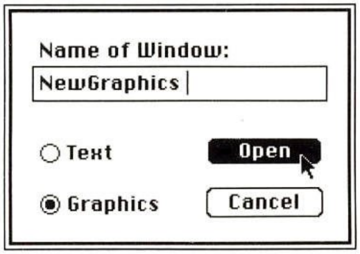
```
cg
repeat 4 [fd 50 rt 90]
print [one small step for turtles]
one small step for turtles
print [a giant leap for turtle-kind
a giant leap for turtle - kind
```

Now click the close box at the top left corner of the window to make it disappear.

Open a new graphics window

To open a new graphics window:

1. Choose Open Window from the File menu. A dialog box appears.
2. Select "Graphics" as the kind of window, and type any name you like.
3. Click the Open button.

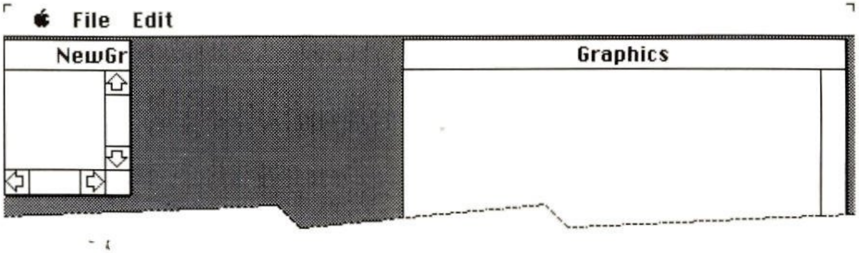


Name of Window: NewGraphics

○ Text

● Graphics

The new graphics window will appear at the top left corner of the screen.

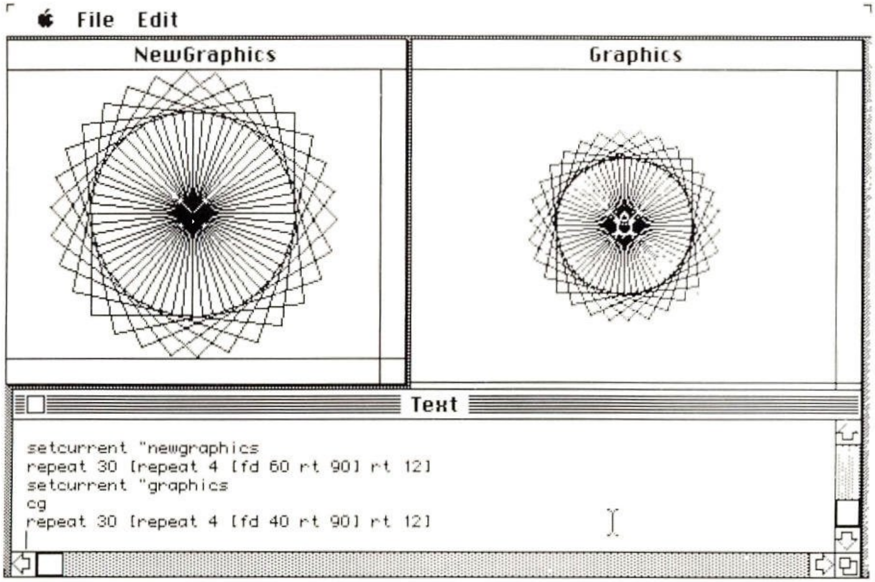


To draw on the new graphics window, use the command SetCurrent. Suppose you called the new window NewGraphics. This means that SetCurrent's input is *"NewGraphics.*

Try out the following instructions in the text window:

Draw on two windows

```
setcurrent "newgraphics  Sets the new graphics window.
repeat 30 [repeat 4 [fd 60 rt 90] rt 12]
setcurrent "graphics     Restores the original window.
cg
repeat 30 [repeat 4 [fd 40 rt 90] rt 12]
```



```
setcurrent "newgraphics
repeat 30 [repeat 4 [fd 60 rt 90] rt 12]
setcurrent "graphics
cg
repeat 30 [repeat 4 [fd 40 rt 90] rt 12]
```

Close the window

You can restore the screen to its original appearance by closing the new graphics window. Make the window active by clicking it, then click the close box at the top left corner.

Calculating With Logo

Logo has a full set of mathematics operations built in. To start with an easy one, type:

Add

```
ct
print 4 + 5
```

Logo responds:

```
9
```

Type:

Divide

```
print 30 / 3
```

Logo responds:

```
10
```

You can use the result of a calculation as an input to a command. For instance:

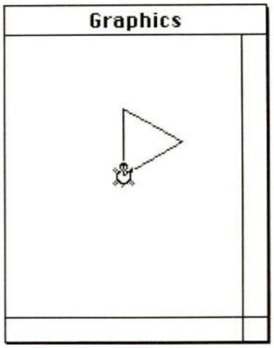
```
cg
forward 30+40
```

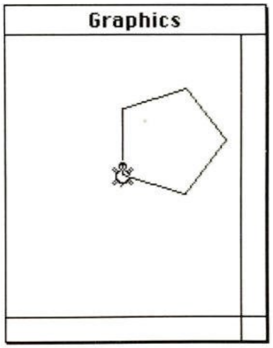
The turtle goes forward 70 steps.

You can draw polygons without doing the math yourself:

```
cg
repeat 3 [fd 50 rt 360 / 3]
```

```
cg
repeat 5 [fd 50 rt 360 / 5]
```





Experiment by inventing your own polygons. Polygons will be explored in more depth in Chapter 6, "Drawing Polygons and Spirals". For a complete listing of all Logo mathematical operations, refer to Chapter 8, "Mathematics", in the *Reference Manual.*

Reflection

Turtle Geometry

You probably think of shapes as static objects. But, with the turtle, geometric shapes have a dynamic element because of the *process* the turtle goes through to make a shape. The basic turtle commands – Forward, Back, Right, and Left – describe this process of constructive geometric shapes. These primitive procedures change the state of the turtle by changing its position or heading.

Inputs

Many Logo procedures need an input in order to produce an effect. In this chapter, you have already experimented with a few: Forward, Right, Back, Left, and Print. If you forget the input and merely type:

```
forward
```

Logo tells you:

```
Not enough inputs to FORWARD
```

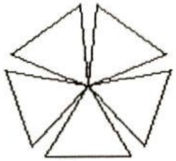
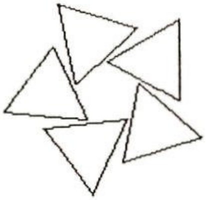
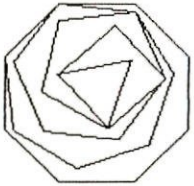
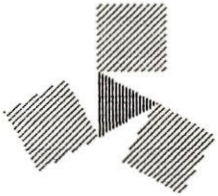
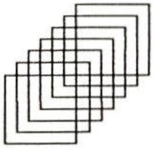
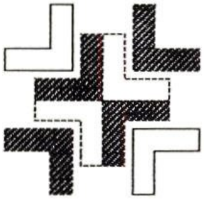
Forward, Right, Back, and Left need a number as their input, while Print can use a word, a list, or a number as its input.

Bugs

As you learn Logo, you will inevitably make mistakes or "bugs". Bugs indicate that something unexpected has happened. Most of the time you "debug" by finding out what happened and correcting it. Sometimes, a bug gives you a new idea and makes you aim for a different result. Investigating bugs can be one of the best ways to learn.

Exploring Further

Try drawing these designs with the turtle:

Note The procedures which create these graphics and the graphics in the other "Exploring Further" sections are on the Master Logo disk, in a file named "Exploring Further".

Logo Vocabulary

Note that Logo doesn't differentiate between capital and lower case letters. Thus:

```
FORWARD
forward
FoRwArd
Forward
```

all have the same effect.

You can see an explanation of any of the procedure names shown in this section. Select the name and choose Help from the Edit menu.

Commands

Back Bk
CG (for Clear Graphics)
CT (for Clear Text)
FillSh (for Fill Shape)
Forward Fd
HideTurtle HT
Home
Left Lt
PenDown PD
PenErase PE
PenUp PU
Print Pr
Repeat
Right Rt
SetCurrent
SetPPattern (for Set Pen Pattern)
ShowTurtle ST

Special Characters

" (quotation mark) for quoting a word
[ ] (brackets) for enclosing a list

Operations

+ (plus)
/ (divided by)

Menu Items

Clear
Open Window

3 Defining Procedures and Using Subprocedures

You have already instructed the turtle to draw a design such as a square or a hexagon. To draw it again, you could retype all the Logo instructions. It would be simpler if you could type one word and get the same result. This can be done by writing a *procedure.* Writing a procedure means giving a name to a series of instructions. Every time you want to run the procedure, you can just type the procedure's name rather than all the individual instructions.

Action

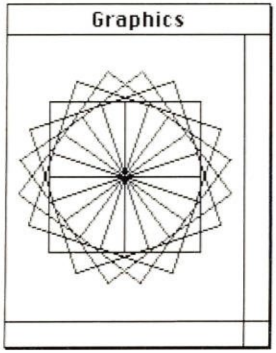
Defining Procedures

Here's an instruction which draws a square:

```
repeat 4 [fd 50 rt 90]
```

Here's a longer instruction which draws 20 squares, each at a different angle:

```
repeat 20 [repeat 4 [fd 50 rt 90] rt 18]
```

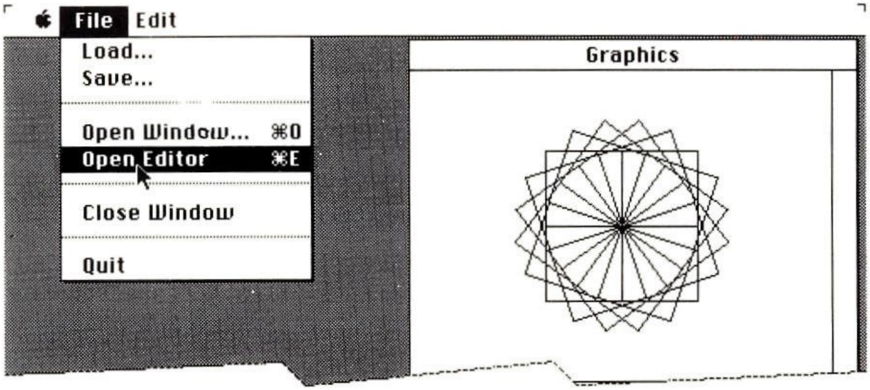


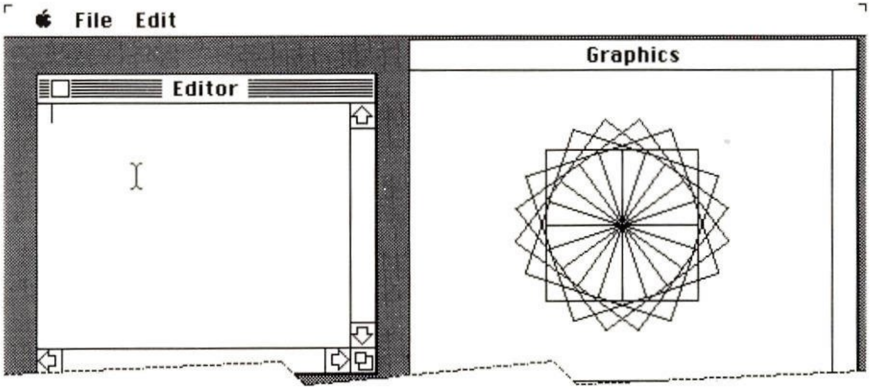
You could continue to write longer instructions that do more complex graphics, but at a certain point, the logic becomes difficult to follow. It is easier to simplify instructions by separating them into individual functions and naming them.

Define a procedure

The instruction that draws a square, for instance, may be defined as a procedure called Square.

First, choose Open Editor from the File menu.





Open the Editor

A new kind of window called the Editor is on the screen. You can define a new procedure in the Editor. Choose a name (Square, in this instance), then type:

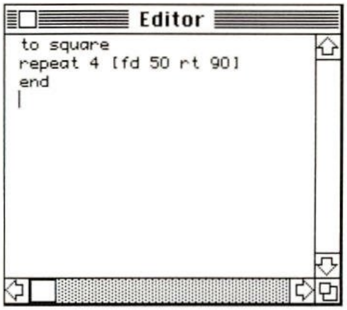
```
to square
```

Press Return.

To Square is the *title line*. *To* tells Logo that the text that follows is part of a procedure. *Square* is the name of the procedure.

Now type the Repeat instruction as shown below. *End* is always the last line of the procedure.

```
to square
repeat 4 [fd 50 rt 90]
end
```



```
to square
repeat 4 [fd 50 rt 90]
end
```

While the Editor is active, it is just *storing* lines of Logo. It does not try to run them.

In the Editor, all of the text-editing features like Cut, Copy, and Paste are still available. You can even copy text into or from a text window.

Once you have typed "End", the procedure definition is complete. Press Enter. The Editor becomes inactive. Logo responds on the text window:

```
SQUARE defined
```

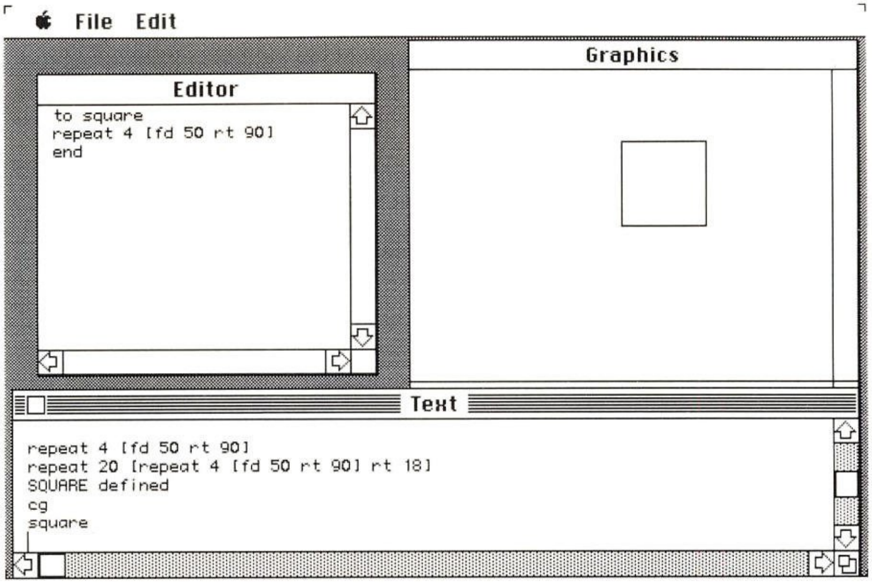
Note While you are defining a procedure in the Editor, press Return to separate each line. If you accidentally press Enter before the procedure definition is complete, Logo will respond in the text window:

```
________ defined
```

Try the new procedure

Try your new procedure by typing (on the text window):

`cg`	Press Enter.
`square`	Press Enter.



```
to square
repeat 4 [fd 50 rt 90]
end
```

```
repeat 4 [fd 50 rt 90]
repeat 20 [repeat 4 [fd 50 rt 90] rt 18]
SQUARE defined
cg
square
```

Now you can use Square as a command like Forward or Right. If you turn the turtle slightly and type *Square* again a new square will appear:

```
rt 30
square
```

With the name Square replacing the instruction Repeat 4 [Fd 50 Rt 90], the complex Logo instruction:

```
repeat 20 [repeat 4 [fd 50 rt 90] rt 18]
```

becomes:

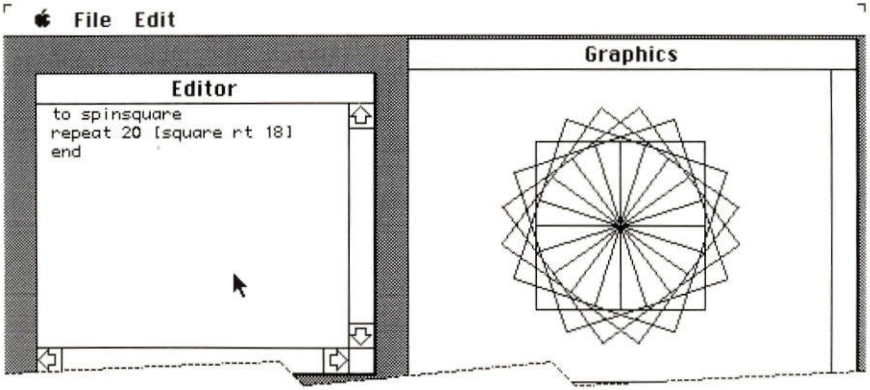
```
repeat 20 [square rt 18]
```

These spinning squares may also be defined as a procedure.

Click the Editor to make it active. Choose Clear from the Edit menu to clear this window. Then type:

`to spinsquare`	Title line.
`repeat 20 [square rt 18]`	Body of procedure.
`end`	Last line.

Press Enter. Now clear the graphics window with CG and try SpinSquare.



```
to spinsquare
repeat 20 [square rt 18]
end
```

Fixing Bugs in a Procedure

Edit a procedure

When you try out Square or SpinSquare, you may not get the expected result because there is a bug in the procedure. The bug may be a typing mistake, incorrect spacing, or the absence of an input. For instance, if there is a bug in SpinSquare, just click the editor window. The procedure is still there. You can then edit the procedure to fix the bug.

If the procedure has been cleared from the editor window, you can put it back by typing:

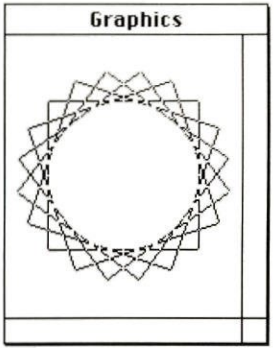
```
edit "spinsquare
```

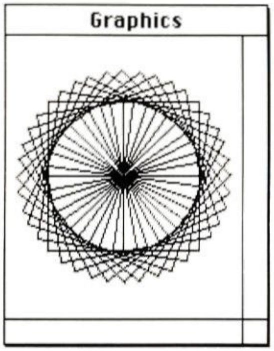
in the text window. Logo will make the Editor active with only the SpinSquare procedure in it.

When you have finished editing, press Enter. The text window becomes active again, and Logo responds:

```
SPINSQUARE defined
```

Using Square and SpinSquare, create designs like these:





```
cg
penreverse spinsquare
spinsquare
pendown
repeat 2 [spinsquare rt 8]
```

Note If you hide the turtle, it will draw even faster.

Using Cut and Paste to Edit

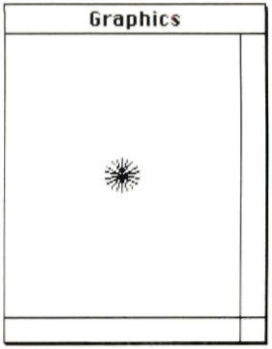
You don't have to retype instructions each time you enter the Editor. If there is something you like in a text window that you want to define, just select the line or lines and choose Cut or Copy from the Edit menu. Click the Editor. Then type *To* and the name you choose. Press Return and choose Paste from the Edit menu. The line or lines that you cut or copied will be pasted in at the insertion point in the Editor. Type *End* on the last line and press Enter. The new procedure is now defined.

Drawing a Starry Sky

The power of Logo programming comes from using procedures to build other procedures, just as Square was used to create SpinSquare.

Click the Editor to write a procedure named Star that draws a number of lines around a single point:

```
to star
repeat 18 [fd 10 bk 10 rt 20]
end
```



Use the Random operation

You can use this procedure to draw several stars, each at a different place. To ensure that the turtle moves to a new location before drawing each star, use the Random operation. Random produces a random number that's less than the number given as its input. On the text window try:

```
print random 6              Press Enter.
```

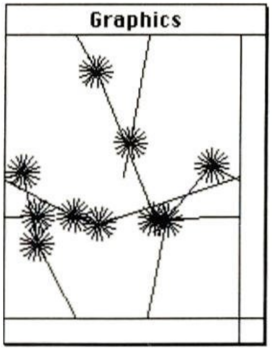
Move the insertion point back to Print Random 6 and press Enter again. A different number will probably result.

By using Random as an input to Forward and Right, you can turn the turtle an unpredictable amount and move the turtle an unpredictable distance. Type this procedure in the Editor:

```
to move
rt random 360               Random turn.
fd random 150               Random distance.
end
```

With Move defined, try:

```
cg
repeat 10 [move star]
```



Fix a bug

A bug! The turtle draws a line as it moves, spoiling the illusion of a starry sky. Edit the Move procedure by clicking the Editor:

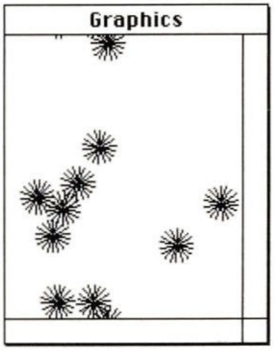
```
to move
penup
rt random 360
fd random 150
pendown
end
```

Lift the pen before moving.

Put the pen down afterwards.
Press Enter to redefine Move.

Try it again:

```
cg
repeat 10 [move star]
```



Writing a Superprocedure

Now that a sky full of stars is working, a single procedure could run Move and Star. The name for a procedure which uses other procedures is a *superprocedure.* The name for a procedure which is used by another procedure is a *subprocedure.* Here's a simple superprocedure called Sky.

```
to sky
repeat 8 [move star]
end
```

If you want more stars, change the Repeat number.

Important It's best not to quit Logo or turn off your computer now. If you do, you'll lose all your procedures. Before quitting, read the next chapter, "Examining Your Workspace and Saving Files", which explains how to save your procedures on a disk.

Printing Your Pictures

If you create a picture you like, it's easy to print it out on a printer. The key combination ⌘-SHIFT-4 prints the active window (click a window to make it active). To print the entire screen, press the Caps Lock key before pressing the ⌘-SHIFT-4 keys.

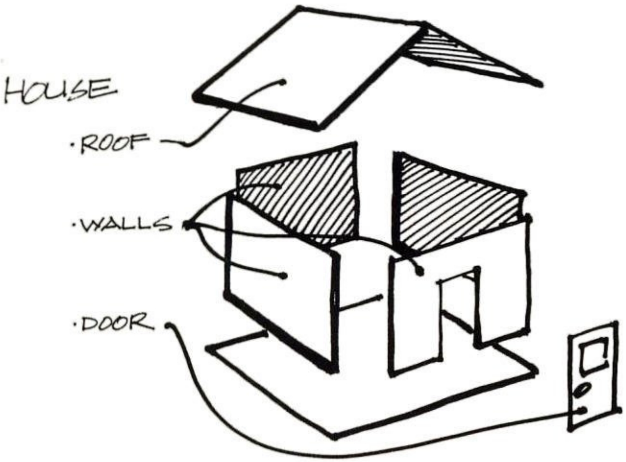
Reflection

Naming

Naming a procedure is an essential part of the Logo language. Logo starts with a basic set of words, known as *primitive procedures.* Each time you define a procedure, you add a word to Logo's vocabulary. This lets you customize the language.

It is helpful to name a procedure in terms of its function. For example:

```
to house
walls
roof
door
end
```



More Turtle Geometry

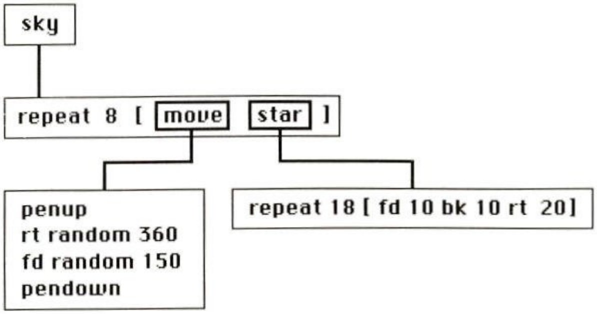
Total Turtle Trip

By now, you have probably noticed that the turtle turns a total of 360 degrees when drawing a square or a triangle or when it goes around and ends up where it started, as in Star. A general principle of turtle geometry called the *Total Turtle Trip* states that the turtle turns a total of 360 degrees to draw any closed figure if the turtle starts and ends facing the same direction. Therefore, each turn of a triangle equals 360/3; a square, 360/4; a hexagon, 360/6, and so on.

Superprocedures and Subprocedures

Sky has two subprocedures, Move and Star. Subprocedures make superprocedures more concise and make debugging easier. When Logo prints an error message, it indicates in which procedure the bug occurred.

How does Logo run a superprocedure having subprocedures? Sky's instructions are run one by one. When the instruction Move is called, Move's instructions are also run one by one. When they are finished, Sky continues with its next instruction to call Star. Star's instructions are then run one by one.



```
sky
```

```
repeat 8 [ move star ]
```

```
penup
rt random 360
fd random 150
pendown
```

```
repeat 18 [ fd 10 bk 10 rt 20]
```

Commands and Operations

There are two kinds of Logo procedures. Most of the procedures you have used so far are *commands.* Forward is the command to move the turtle, Right to turn, Print to print, PenUp to raise the turtle's pen.

You have also used the second kind of Logo procedure called an *operation.* An operation produces or "outputs" something to be used as an input. Random is an operation that produces a random number. The + sign is a familiar operation that produces the sum of its two inputs.

Use an operation as an input

Operations like Random and + can only be used as inputs to other procedures. For example:

`print 5 + 6`	11 will be printed.
`fd random 50`	The turtle will go forward a random amount.

If an operation is the only thing on a line, as if it were a command, Logo complains:

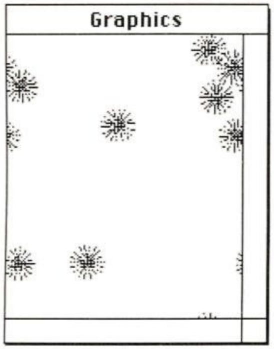
```
random 50
You don't say what to do with 23

11 + 29
You don't say what to do with 40
```

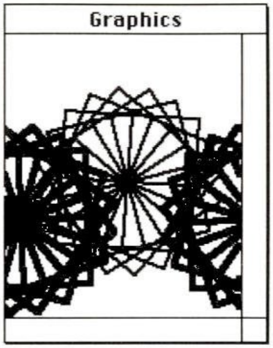
The first word on an instruction line must always be a command.

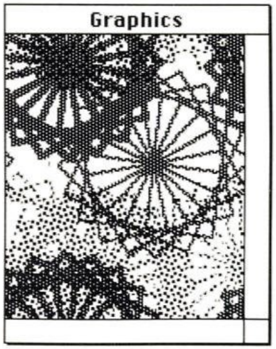
Exploring Further

As you have seen, once you have defined a procedure, it can be used as a tool for building other procedures. Here are some ideas for using SpinSquare and Sky.

Using SetPPattern, change the pattern of squares in SpinSquare, or stars in Sky.



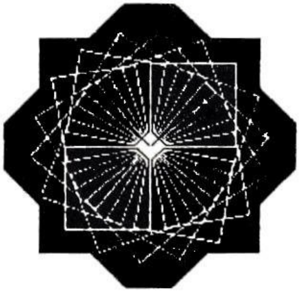




FatSquares uses Move and SpinSquare as subprocedures. The command, SetPWidth (for Set Pen Width), changes the width of lines drawn.

```
to fatsquares
setpwidth 2                  Sets the pen width to 2.
move spinsquare
setpwidth 4                  Sets the pen width to 4.
move spinsquare
setpwidth 1                  Restores the pen width to 1.
end
```

Try using PenReverse with SpinSquare to create an unusual effect.



Note The procedures which create these graphics and the graphics in the other "Exploring Further" sections are on the Master Logo disk, in a file named "Exploring Further".

Logo Vocabulary

Commands

Edit
PenReverse PX
SetPWidth (for Set Pen Width)

Operations

Random

Special Words

End
To

Menu Items

Open Editor

4 Examining Your Workspace and Saving Files

When you define procedures, Logo puts them in your *workspace* – a space in computer memory. When you quit Logo or turn off the computer, the information in the workspace is destroyed. To store your procedures permanently, you must copy them onto a disk. Procedures saved on a disk can be compared to files kept in a filing cabinet for permanent storage.

This chapter explains how to examine your workspace, and how to save procedures on a disk.

Action

Examining Your Workspace

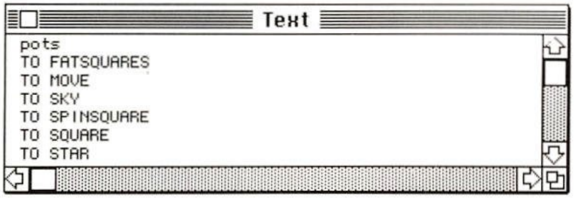
There are several commands that allow you to examine your workspace. To print the titles of the defined procedures that are in your workspace, type:

Print out procedure titles

```
pots
```

for Print Out Titles.

If you haven't quit Logo since Chapter 3, all the procedure names you defined are displayed in the text window.



```
pots
TO FATSQUARES
TO MOVE
TO SKY
TO SPINSQUARE
TO SQUARE
TO STAR
```

Note If POTS doesn't have any effect, you've probably just started up Logo and there are no procedures in your workspace for POTS to display. In this case, it's a good idea to go back to Chapter 3 and write a few procedures, so you can try out the new commands in this chapter.

Print out procedure definitions

To print out the definition of a procedure, use the command POP (for Print Out Procedure). For example, if the Square procedure is in your workspace:

```
pop "square
```

A quotation mark precedes a name.

prints:

```
TO SQUARE
repeat 4 [fd 50 rt 90]
END
```

You can also print the definitions of a list of procedure names:

```
pop [star sky]
```

Brackets enclose a list.

To list all the procedure definitions in your workspace, use the Procedures operation as POP's input. Procedures outputs all the procedure names currently in your workspace. Try:

```
pop procedures
```

The procedure listings may scroll right off the text window, but you can move your viewing area up and down with the scroll bar.

Erasing From the Workspace

As you view your procedures, you may notice "buggy" procedures or procedures you no longer need and don't want to save. Use the command EraseProc (for Erase Procedure) to erase one procedure or a list of procedures. First, clean the Editor by choosing Clear from the Edit menu, so you won't accidentally redefine the procedures in the Editor.

Note The names used here are not names of procedures you've defined, since you don't want to erase anything useful.

Erase procedures

```
eraseproc "Fred
```

Remember the quotation mark.

would erase the procedure called Fred.

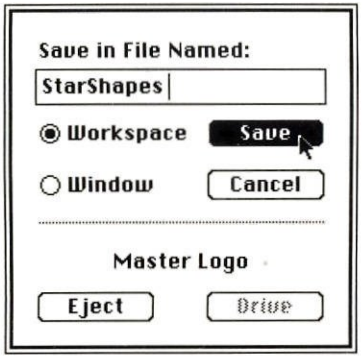
```
eraseproc [setup starcircle]
```

Remember the brackets.

would erase the Setup and StarCircle procedures.

Saving Your Workspace

Choose Save from the File menu to save your workspace into a disk file that you will name.



Save in File Named:
StarShapes

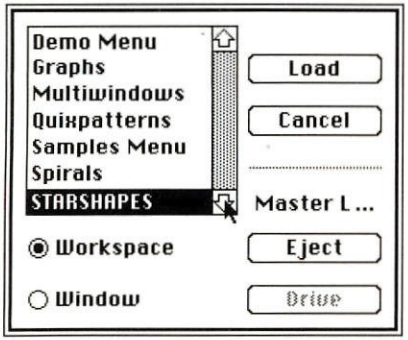
◉ Workspace
○ Window

Click Workspace, then enter a name, such as StarShapes. Click Save. Now the workspace will be saved on a disk, in a file with the name StarShapes.

Listing Files

To check which files are on your disk, choose Load from the File menu. The Load box appears. Click Workspace to list the names of all the program files on the disk.

You should see the filename you just saved on this screen of information:



Demo Menu
Graphs
Multiwindows
Quixpatterns
Samples Menu
Spirals
STARSHAPES

◉ Workspace
○ Window

Click Cancel to make the Load box disappear.

Clearing Your Workspace

The ErAll command

Normally, you save your workspace when you've finished a project, or at the end of a programming session. Before beginning a new project or retrieving other files from your disk, it's a good idea to clear out your workspace. To do this, use ErAll (for Erase All).

Note Use ErAll only after you have saved your workspace on a disk.

Try:

```
erall
```

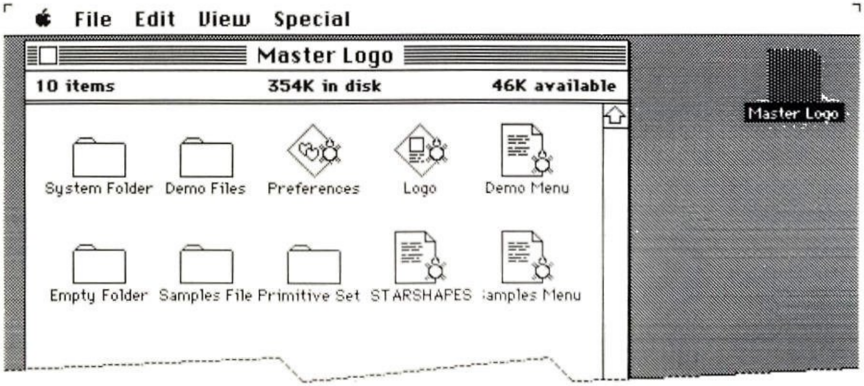
Now type:

```
pots
```

No procedure titles are printed because everything has been erased.

Once you have saved your procedures, you can choose Quit from the File menu to exit Logo without losing any of your procedures.

The Finder will display your file:



Loading Files

When you want to retrieve your procedure files from a disk, choose Load from the File menu. When the Load box appears, click Workspace and then the filename (for example, StarShapes).

Use POTS to see what is in your workspace now. Loading a file doesn't erase what is in your workspace. If you already have procedures in your workspace, the procedures loaded from the disk are added to those already in the workspace. Any procedure in your workspace with the same name as one in the file being loaded will be replaced by the new procedure.

Erasing Files

To erase a file permanently from disk, use the command EraseFile (ErF for short). Its input is a filename, as in:

```
erasefile "Fredfile
```

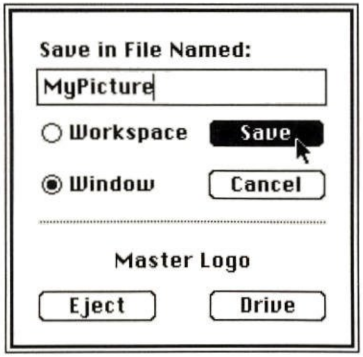
Warning Use this command with caution because its effect is permanent.

Saving and Loading Windows

The drawings in the graphics window can be saved in the same way as your workspace. If there is a picture in the window (for example, the stars):

1. Click the graphics window to make it active.
2. Choose Save in the File menu.
3. When the Save box appears, select Window (instead of Workspace).
4. Enter a name, such as MyPicture.
5. Click the Save button.

The picture in the graphics window will be saved under the filename you choose.



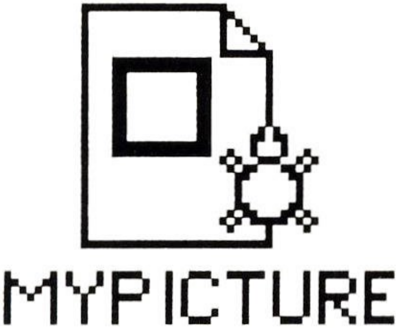
Save in File Named: MyPicture

○ Workspace

● Window

Master Logo

Note that graphics files have a unique icon. Double-clicking this icon does not load the file.



To load the file back into the graphics window, simply choose Load from the File menu, and choose the filename. Make sure a graphics window is active when you want to load a graphics file; otherwise, the Load box will display only the names of text files. With a graphics window active, the Load box will display the names of graphics files.

For more information on workspace and file handling, refer to Chapter 5, "Workspace Management and Disk Drive Control", and Chapter 9, "Device Management", in the *Reference Manual.*

Reflection

Distinguishing Workspace From File Space

Defined procedures exist in your workspace only while the computer is on. Procedures in a disk file are permanently recorded. When you save a file, you are putting a *copy* of your workspace on the disk. When you load a file from a disk, you are putting a *copy* of the file in your workspace. The file on your disk remains the same.

Remember, when you edit procedures that have already been saved in a file, you must replace the old file with the updated version or simply save a new version of the file. When saving a new file, you shouldn't use two words for a name, or use the name of a file that already exists on the disk.

Naming Files

Most often, a file is a set of procedures, which are each part of the same program. For example, the procedures in the file StarShapes are part of the Sky program. Giving your file a meaningful name helps you retrieve it later. The examples here avoid naming a file by the same name as one of the procedures, to remind you that your file is a *set* of procedures and not only one procedure.

Logo Vocabulary

Commands

ErAll (for Erase All)
EraseFile ErF
EraseProc ErP (for Erase Procedure)
POProc POP (for Print Out Procedures)
POTS (for Print Out Titles)

Menu Items

Load
Save

Operations

Procedures

5 Using Variables

Some primitive procedures require inputs. For example, Forward needs an input to tell Logo how many steps to make the turtle move; Right and Left need inputs to tell Logo how many degrees to turn the turtle. The function of these primitives is constant. When executed, Right always turns the turtle clockwise. However, the input to Right is *variable*. Whatever value is given as Right's input determines the amount the turtle turns.

CT is a primitive procedure that requires no input. The defined procedures Square and Star, like CT, need no inputs. Square and Star produce exactly the same actions on the screen each time they are run.

If you wanted the turtle to draw squares of different sizes, you could write a series of procedures like Square10, Square20, Square30, etc., but that would be cumbersome. Instead of many procedures to draw squares of specific sizes, you can define a general procedure that will draw squares of any size, by writing a procedure that uses an input to specify the size of the square. This means that:

Square 20 will make a small square
Square 200 will make a huge square

Action

Defining Procedures With Inputs

Choose Load from the File menu and select the StarShapes file that you saved in the previous chapter. Now edit Square to give it an input for its size:

```
edit "square
```

If the Square procedure is in your workspace, it will appear in the Editor:

```
TO SQUARE
repeat 4 [fd 50 rt 90]
END
```

What is the : (colon)?

The length of each side of the square is determined by Fd's input. To make the procedure create all sizes of squares, Fd's input must be made variable. This can be done by giving a name to the input. The name Size would be appropriate. Replace 50 with *:Size* in the procedure. The : (colon) preceding Size tells Logo that Size is a name that represents a value, not the name of a procedure. Think of the : (colon) as saying "the thing that is called".

```
TO SQUARE
repeat 4 [fd :size rt 90]
END
```

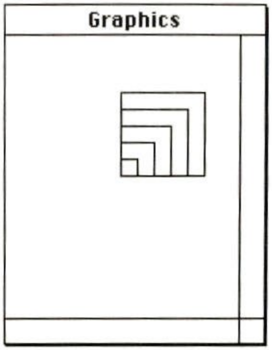
An important detail – on the title line (To Square), you must indicate that Square has an input called Size. This is what the new Square procedure should look like:

Add an input to Square

```
TO SQUARE :size              :Size on the title line.
repeat 4 [fd :size rt 90]    :Size as Fd's input.
END
```

Press Enter to define the revised procedure, then try it:

```
square 10
square 20
square 30
square 40
square 50
```



What happens if you forget to give Square an input? If you type:

```
square
```

Logo responds:

```
Not enough inputs to SQUARE
```

Checking for Possible Bugs

If your new Square procedure doesn't work, check for the following bugs:

- No : (colon) preceding Size
- A space between : (colon) and Size
- A typing mistake. For example, :Size on the title line and :Sise within the procedure
- When running Square, you put a : (colon) preceding the input number. For example, Square :50

Defining a Text Procedure With Inputs

Define a procedure with two inputs

Inputs are useful for all kinds of defined procedures, not just graphics. An input can be a word or a list as well as a number. For example, define a new procedure called Many that has two inputs.

```
to many :times :message
repeat :times [print :message]
end
```

Note Use the horizontal scroll bar to see a line that's longer than the editor window width.

Try:

```
many 3 "Judy
```

Logo responds:

```
Judy
Judy
Judy
```

Since Print will print words, numbers, or lists, Many's second input can be any of these. Since Repeat's first input must be a number, Many's first input also must be a number.

```
many 10 [Alphonse Q. McKoy]
many 10 2000
```

Creating a Variable Sized Star

Now edit Star to take an input.

```
edit "star
```

If the Star procedure is in your workspace, it will appear in the Editor:

```
TO STAR
repeat 18 [fd 10 bk 10 rt 20]
END
```

Edit Star and add an input for the length of the lines:

Add an input to Star

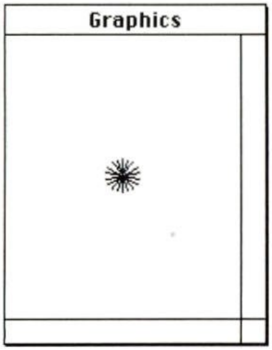
```
TO STAR :length
repeat 18 [fd :length bk :length rt 20]
END
```

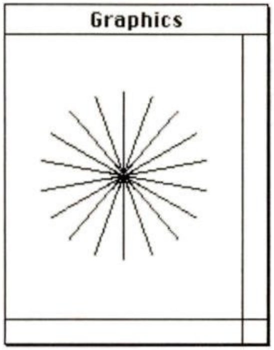
:Length on the title line.

:Length as input for Fd and Bk.

Now try:

```
star 10
cg
star 50
```





You may want to edit Sky so you can choose the size of stars in the sky. Here are the original definitions of Sky and Move (Sky's subprocedure):

```
TO SKY
repeat 8 [move star]
END
```

```
TO MOVE
penup
rt random 360
fd random 150
pendown
END
```

If you run Sky as is, Logo complains:

```
Not enough inputs to STAR in SKY
```

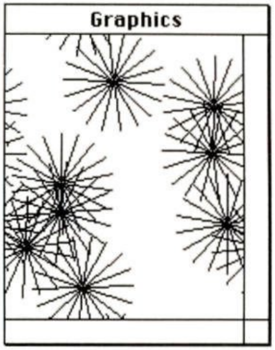
Of course, Star requires an input in order to work. Sky also needs an input on the title line:

Add an input to Sky

```
TO SKY :size                  Input on the title line.
repeat 8 [move star :size]    Input for Star.
END
```

Experiment with Sky now:

```
sky 40
```



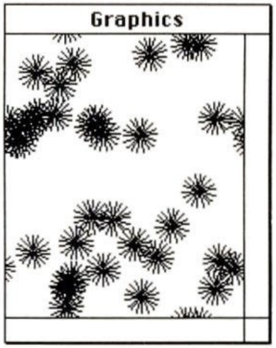
You may want to add another input to Sky – the number of stars to draw:

```
TO SKY :amount :size
repeat :amount [move star :size]
END
```

Remember that Sky now takes two inputs when you run it:

```
cg
sky 35 10
```

35 is the Amount.
10 is the Size.



Reflection

Use inputs as variables

The idea of variables is a powerful one. Variables allow you to make your procedures, whether they manipulate graphics, text, or numbers, more flexible. A variable in a graphics procedure may allow you to vary its size with the same procedure. To help you remember what a variable does, use a meaningful name, like Size.

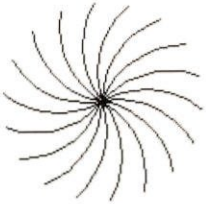
When you defined Square, you wanted to give Forward an input of some variable number, so you named it Size. Naming an input lets you refer to it in general terms. In Logo, :Size refers to whatever happens to have the value of Size. How do you give Size a value? When you type Square 10 or Square 25, Size takes the value of 10, 25, or whatever number you type as input.

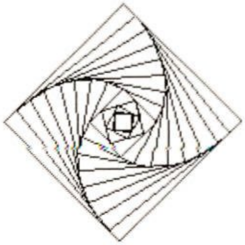
The value of a variable can be passed from a superprocedure to a subprocedure. For example, the Sky procedure passes the value of :Size to Star.

Exploring Further

Try other ways of using variables. For example:

- Circle with input for Size
- Arcs of Circles with inputs for the size and degree of the arc





Note The procedures which create these graphics and the graphics in the other "Exploring Further" sections are on the Master Logo disk, in a file named "Exploring Further".

Logo Vocabulary

Special characters

: (colon) for "the thing that is called"

6 Drawing Polygons and Spirals

Just as you can vary the number of steps the turtle takes, you can also vary how much it turns. In fact, you can produce some beautiful and surprising designs by varying both these components. A procedure for drawing polygons which takes these components as inputs will be defined in this chapter. This procedure is *recursive;* it runs itself as a subprocedure. The chapter will also suggest ways of experimenting with and exploring polygons.

Action

Drawing Polygons

The Poly procedure takes two inputs: one for the number of turtle steps; the other, the amount to turn:

```
to poly :step :angle
fd :step
rt :angle
poly :step :angle
end
```

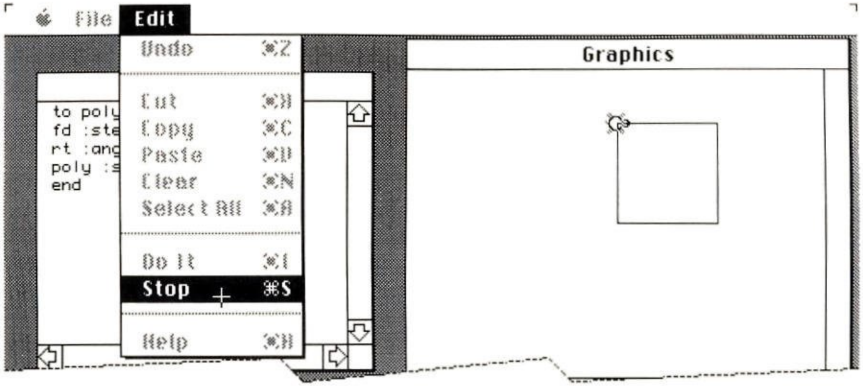
The recursive line.

The last line of Poly before End is the *recursive line.* This is an instruction to run Poly as a subprocedure.

What is the recursive line?

Now try it! (Show the turtle before running Poly so you can see the process.)

```
poly 60 90
```



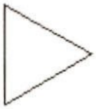
```
to poly
fd :ste
rt :ang
poly :s
end
```

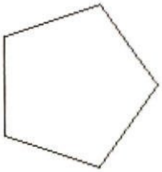
Stop a recursive procedure

A square! Choose Stop (from the Edit menu) to stop Poly. Stop signals Logo to stop what it is doing.

Try Poly with other inputs. For example:

```
poly 60 120
poly 60 72
poly 75 160
```





As you change the Angle input, notice that the shape of the polygon changes. Experiment with other inputs for Poly. How many different kinds of shapes can you produce? Try to predict the kind of polygon a particular angle will produce.

Defining a Sun

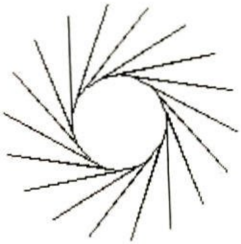
The Sun procedure resembles Poly except that it moves the turtle back before turning. Its designs look like sun rays because of the inputs for the forward step, back step and turn. Here is Sun's definition:

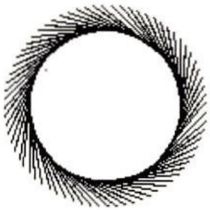
```
to sun :fdstep :bkstep :turn
fd :fdstep
bk :bkstep
rt :turn
sun :fdstep :bkstep :turn    The recursive line.
end
```

The recursive line instructs Sun to run itself as a subprocedure.

Try:

```
sun 70 60 20
sun 30 50 25
```





Choose Stop to stop the procedure.

Try other inputs. What will happen if Sun's second input is zero? What will happen if FdStep and BkStep are equal?

Drawing Spirals

Both Poly and Sun instruct the turtle to draw closed figures. The turtle goes forward and rotates to get back to where it started.

To draw a spiral, the turtle should not go back to where it started. Instead, the turtle should increase its forward step on each round so that it moves further and further away from its starting point.

Do this by slightly increasing the value of Step on the recursive line. Edit Poly to define Spi by changing the title and recursive lines.

Use comments in a procedure

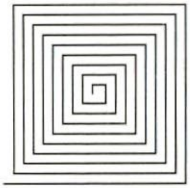
Add a *comment* in the procedure definition to help you (or someone else) understand what the procedure does. A comment is a line following a semicolon (;). The semicolon signals Logo to ignore the rest of the line.

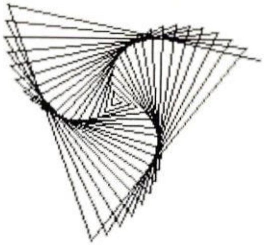
```
to spi :step :angle                  The title line.
fd :step
rt :angle
;step increases on each round        The comment.
spi :step + 3 :angle                 The recursive line.
end
```

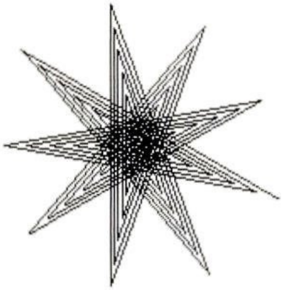
Notice the difference in the recursive lines of Spi and Poly. Poly's recursive line is an exact copy of its title line. This means that each round of recursion is exactly like the previous one. Spi's recursive line is not an exact copy: 3 is added to the value of Step. When each Spi subprocedure runs, it draws a longer side.

Now, experiment with Spi. (Choose Stop to stop.)

```
spi 5 90
spi 0 122
spi 5 160
```





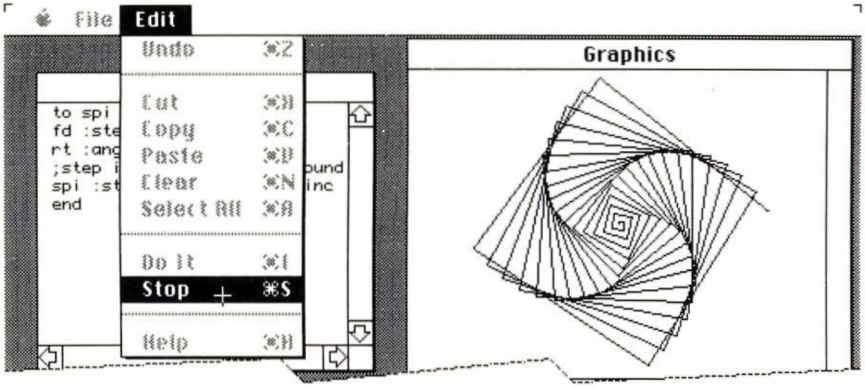


You can make Spi more interesting by making the increment *+3* variable. This will become a third input named Inc. Spi will add :Inc to :Step, instead of 3. :Inc will allow you to vary the amount added to the number of turtle steps by choosing different numbers for its input.

```
to spi :step :angle :inc
fd :step
rt :angle
;step increases on each round
spi :step + :inc :angle :inc
end
```

For example:

```
spi 1 92 2
```



```
to spi
fd :ste
rt :ang
;step i          ound
spi :st          inc
end
```

Try varying the third input to produce different effects.

Reflection

Experimenting

Throughout this book, you are encouraged to experiment with ideas other than those presented here. However, it is not always obvious how to explore the primitive procedures and concepts presented. This chapter provides you with three procedures that produce exciting effects. You can explore Poly, Sun, and Spi graphically to create exciting designs, or write your own procedures to produce other recursive designs.

Total Turtle Trip Revised

According to the Total Turtle Trip, the turtle will turn a total of 360 degrees to complete the trip around any closed figure when the turtle starts and ends with the same position and heading.

However, if you follow the turtle's trip around a star polygon, you'll notice an aberration. For example, Poly 50 144, a five-pointed star, makes the turtle turn a total of 720 (144 * 5) degrees. The turtle completes a full rotation twice. When turning around the third point of the star, the turtle rotates through its initial heading. Verify this phenomenon for different stars. The Total Turtle Trip must now be revised as follows:

The turtle will turn a total of 360 degrees *or a multiple of 360 degrees* to complete the trip around any closed figure when the turtle starts and ends with the same position and heading.

Recursion

You have seen a few examples of recursion: Poly runs Poly as part of its definition, Sun runs Sun, and Spi runs Spi. What is recursion all about?

Consider this recursive riddle:
If you had two wishes, what would your second wish be?
Answer: Two more wishes.

Nested Russian dolls is another example which works much like Spi. A painting inside another painting, a movie within a movie, a story within a story like "A Thousand and One Arabian Nights", are all examples of recursion.

The notion that recursion continues forever gives us a chance to play with infinity. The easiest way of making a recursive procedure stop is by choosing Stop from the Edit menu. The next chapter explores ways of embedding a "stop rule" in a recursive procedure, enabling you to specify the condition when the procedure will stop.

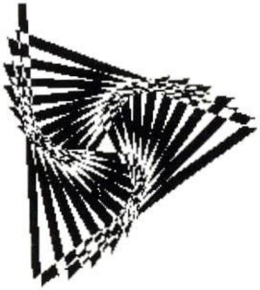
Exploring Further

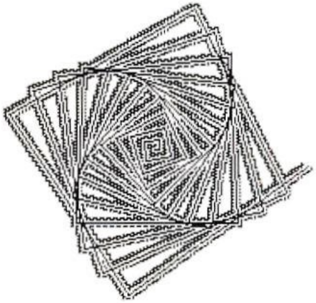
Modify Spi so it draws from the outside in rather than from the inside out.

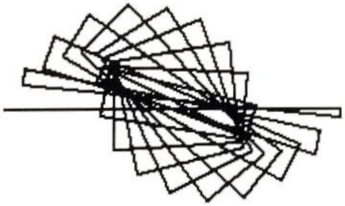
Try Spi and Poly with different pen patterns.

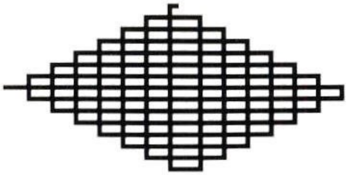
What will happen if Spi increases the angle *instead* of the forward step? Write a procedure to experiment.

Write a Shrink-Grow procedure that alternately decreases and increases the forward step while keeping the angle constant at 90 degrees.









Note The procedures which create these graphics and the graphics in the other "Exploring Further" sections are on the Master Logo disk, in a file named "Exploring Further".

Logo Vocabulary

Special Characters

; (semicolon) for a comment line

Menu Items

Stop

7 Exploring Recursive Procedures

This chapter discusses various kinds of recursive procedures and different ways of stopping them within the procedures themselves.

Action

Creating Stop Rules

Recursive procedures such as those illustrated in the previous chapter won't stop unless you choose Stop. For example:

```
to spi :step :angle :inc
fd :step rt :angle
spi :step + :inc :angle :inc
end
```

Try:

```
spi 10 123 5
```

You must choose Stop to end the spiral. However, you *can* modify this procedure to stop another way. In fact, creating appropriate stop rules is an essential part of writing recursive procedures.

Note If you encounter the *Not enough symbol space* message, use the Recycle command. Recycle clears all unneccessary symbols from your workspace. For more information, see Appendix D, "Memory Space", in the *Reference Manual.*

Writing a Stop Rule for Spi

Suppose you decide that Spi should stop if the length of a side (:Step) is greater than 175. Then, insert this line in the procedure:

The > (greater than) operation

```
if :step > 175 [stop]
```

Where should the stop rule be placed? Try putting it immediately after the title line:

Place the stop rule

```
to spi :step :angle :inc
if :step > 175 [stop]
fd :step rt :angle
spi :step + :inc :angle :inc
end
```

The stop rule

Run Spi 10 123 5. Experiment with placing the stop rule on different lines of the procedure. What happens if the stop rule is at the end of the procedure? Do you get different effects? Experiment also with changing the limit of :Step in the stop rule; for example, 250 instead of 175.

Writing a Stop Rule for Poly

Writing a stop rule for Poly is a little trickier. Poly looks like this:

```
to poly :step :angle
fd :step
rt :angle
poly :step :angle
end
```

Poly completes a figure when the turtle returns to its starting state – its original position and heading. This means that the turtle must turn 360 degrees or a multiple of 360 degrees.

You need to know what the turtle's heading is when it starts, and then compare that to the turtle's heading after each turn. The primitive procedure called Heading will help you do this. Heading is an operation that outputs the turtle's heading as a number between 0 and 360. So, before running Poly, make Logo remember the turtle's heading. Do this by naming the heading *Start* with the Name command:

```
name heading "start
```

Heading gives the turtle's current heading.

The Name command

Name's second input is the name we are giving to the information produced by Heading. Since *Start* is a name, it is preceded by a quotation mark. To see the information Start contains, put a : (colon) in front of Start.

```
print :start
```

If the graphics window has just been cleared, Logo responds:

```
0
```

The turtle is pointing straight up.

The following stop rule for Poly checks that the turtle's current heading (the direction the turtle is facing at that moment) is the same as :Start.

```
if heading = :start [stop]
```

When you put this stop rule into the procedure, make sure it's placed after the Rt command. If you put the stop rule before the Rt command, Poly stops immediately, before the turtle starts drawing!

Place the stop rule

```
to poly :step :angle
fd :step
rt :angle
if heading = :start [stop]
poly :step :angle
end
```

The stop rule.

Now, try Poly.

There is a problem here. You must remember to name the starting heading before you run Poly, or the stop rule will not work.

It is best to put that action into a procedure. Write a superprocedure called SuperPoly which gives Start a value and then runs Poly.

```
to superpoly :step :angle
name heading "start
poly :step :angle
end
```

Now SuperPoly does the whole job.

Another way to give Start a value is to add the input :Start to the title and recursive lines:

Give Poly a third input

```
to poly :step :angle :start     :Start on the title line.
fd :step rt :angle
if heading = :start [stop]
poly :step :angle :start        :Start on the recursive line.
end
```

To run Poly, you must also give it a third input which refers to the starting heading. This input can be the operation Heading. This way, Logo calculates the starting heading, which becomes the value of Start. For instance:

Use Heading as an input

```
rt 90
poly 90 144 heading
```

If you don't want to type Heading each time you run Poly, make SuperPoly its superprocedure.

```
to superpoly :step :angle
poly :step :angle heading
end
```

Writing a Stop Rule for Words and Lists

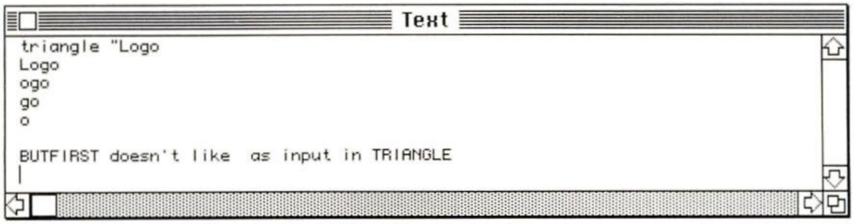
This simple recursive procedure removes one letter at a time from a word (or one word at a time from a list). It creates a kind of triangle.

Create a text triangle

```
to triangle :object
print :object
triangle butfirst :object
end
```

The ButFirst operation

ButFirst (or BF) outputs all but the first element of its input. With ButFirst on the recursive line, :Object loses one element each time Triangle is called.



```
triangle "Logo
Logo
ogo
go
o

BUTFIRST doesn't like  as input in TRIANGLE
```

Triangle has a bug. Logo complains because :Object becomes an *empty word* – a word with no characters. ButFirst tries to take this empty word as its input.

This bug can be fixed by making Triangle stop when its input is empty. The stop rule to do this is:

```
if emptyp :object [stop]
```

The EmptyP operation

EmptyP (P stands for Predicate) outputs True if its input, a word or a list, is empty (contains no elements).

Where should the stop rule be placed?

Place the stop rule

This example makes the stop rule the first line after the title line in Triangle:

```
to triangle :object
if emptyp :object [stop]   The stop rule.
print :object
triangle butfirst :object
end
```

Experiment with placing the stop rule after the Print line.

Now, try:

```
triangle "Logo
```

```
triangle [going going going gone]
```

Adding Instructions After the Recursive Line

Procedures ending with a recursive line are not the only kind of recursive procedures. In fact, instructions after the recursive line produce powerful and sometimes unexpected effects.

This different kind of spiral stops at a specified heading:

```
to curl :step :angle :heading
fd :step rt :angle
if heading = :heading [stop]
curl :step + .5 :angle :heading
end
```

For example, try:

```
curl 5 15 270
```

```
curl 5 15 0
```

Add graphics instructions after the recursive line

For variation, add a few more turtle actions after the recursive line. For example, edit Curl to add:

```
fd :step lt :angle
```

and change Curl's name to Surprise on the title and recursive lines:

```
to surprise :step :angle :heading
fd :step rt :angle
if heading = :heading [stop]
surprise :step + .5 :angle :heading
fd :step lt :angle                          The new line.
end
```

What do you expect will happen?

```
surprise 5 15 270
```

```
surprise 5 20 0
```

Experiment with different inputs.

Add a Print instruction after the recursive line

Now add Print :Object after the recursive line in Triangle to see its effect. Change Triangle's name to Tri2:

```
to tri2 :object
if emptyp :object [stop]
print :object
tri2 butfirst :object
print :object                    The new line.
end
```

Predict what the new line will print if you run Tri2 "Logo.

Now try it.

Reflection

Conditions, Actions, and Predicates

What is a predicate?

The If command needs two inputs: a condition and an action that is carried out if the condition is True. An *action* is a list of Logo instructions. Like other lists, it's enclosed in brackets [ ]. The *condition* is expressed with a special kind of operation called a *predicate,* a word that asks whether something is True or False. The P in EmptyP reminds you it's a predicate. Some other Logo predicates are > (greater than) and = (equals).

Recursion With Words and Graphics

Compare the effect of a recursive procedure like Triangle with a graphics procedure that spirals inward.

```
to triangle :object
if emptyp :object [stop]
print :object
triangle butfirst :object
end
```

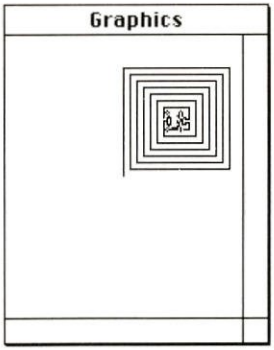
Prints a text triangle.

```
to spiralin :step :angle
if :step < 1 [stop]
fd :step rt :angle
spiralin :step - 2 :angle
end
```

Draws a spiral.

For example:

```
spiralin 90 90
```

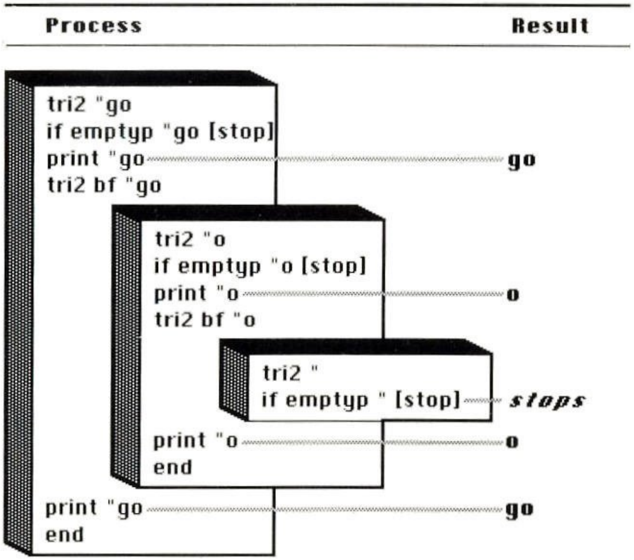


In Triangle, ButFirst on the recursive line removes the first element of its input each time Triangle is called. In SpiralIn, :Step – 2 on the recursive line makes the line drawn by the Fd command shorter each time SpiralIn is called.

The stop rule in Triangle uses the EmptyP operation to check if the word or list is empty of elements. Another way to determine if a word or list is empty is to check whether the number of elements is 0. In SpiralIn, the stop rule checks if :Step is less than 1.

Thinking About Recursion

Run Tri2 "go and look at the result. The following telescoping model mirrors that result. The "Process" column shows the process, or flow of control for Tri2 "go. The "Result" column shows the results that are printed on the window.

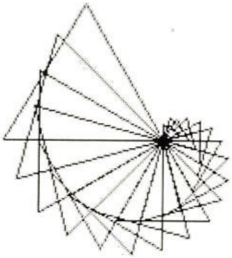


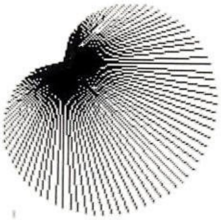
```
tri2 "go
if emptyp "go [stop]
print "go                          go
tri2 bf "go
    tri2 "o
    if emptyp "o [stop]
    print "o                       o
    tri2 bf "o
        tri2 "
        if emptyp " [stop]         stops
    print "o                       o
    end
print "go                          go
end
```

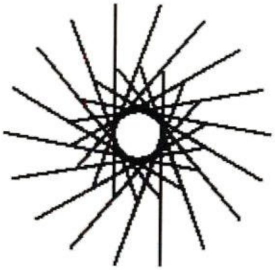
Remember: the recursive line is where Tri2 runs itself as a subprocedure. When Tri2 " stops, each subprocedure (Tri2 "o and Tri2 "go) must finish. This means running the remaining lines of the procedure definition.

Exploring Further

Write your own recursive procedure. Add some actions after the recursive line and check the results.







Note The procedures which create these graphics and the graphics in the other "Exploring Further" sections are on the Master Logo disk, in a file named "Exploring Further".

Logo Vocabulary

Commands

If
Name
Stop
Recycle

Operations

= (equals)
> (greater than)
< (less than)
ButFirst BF
EmptyP (P for Predicate)
Heading

8 Creating a Bar Graph Project

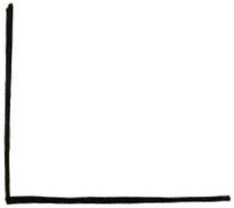
In this chapter, you will develop a project to draw a bar graph. You'll learn several new programming ideas: the interactive program, the technique for printing text on the graphics screen, and manipulating windows under program control.

The interactive program creates a dialog between the computer and the person at the keyboard. An interactive Logo program can be written so everyday English words and sentences are used for questions and answers.

The Scenario

Here is an example of a bar graph program which uses interaction to draw the bars.

The turtle draws the axes of the graph.



Then Logo asks:

```
How many computers were sold in 81?
```

You type:

```
1000
```

Logo calculates a distance to represent 1000, and the turtle draws the first bar of that height.

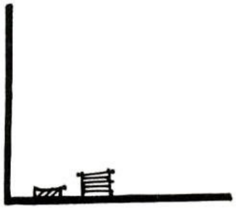
Logo asks:

```
How many computers were sold in 82?
```

You type:

```
2000
```

Logo calculates the distance and the turtle draws the second bar.



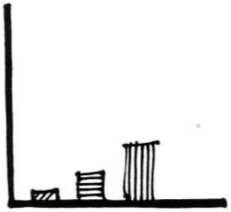
Logo asks:

```
How many computers were sold in 83?
```

You type:

```
5000
```

Logo calculates the distance and the turtle draws the third bar.



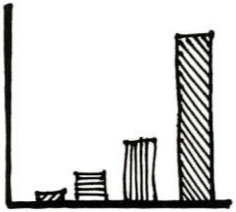
Logo asks:

```
How many computers were sold in 84?
```

You type:

```
10000
```

The turtle draws the fourth bar.



The Plan

Planning is an important part of a programming project. Before starting a project like this one, divide the task into its logical steps:

Step 1: Set up the windows.

Step 2: Draw the picture:
- of the axes, with a marked scale on the y-axis.
- of the bars, side by side on the graph.

Step 3: Make the program interactive by using data from the keyboard to determine the height of each bar.

Step 4: Label the graph and each bar.

Step 5: Write a superprocedure, putting everything together in an easy-to-use way.

Step 6: Add the finishing touches to the program by controlling the initial window set-up.

Note A listing of all the procedures making up the bar graph program can be found at the end of this chapter. There is also a diagram showing the structure of the program.

Action

Step 1: Setting Up the Windows

In any project, it is necessary to set up the initial conditions for running the program. At this point, this means clearing the text and graphics windows and hiding the turtle before drawing the graph. Here is a simple SetUp procedure:

```
to setup
recycle                          Recycle the memory.
ct                               Clear the text.
cg ht                            Clear the graphics.
end
```

Step 2: Drawing the Axes and the Bars

The y-axis will have a scale marked along it. It's useful to have a general procedure that draws the marks. The interval for the scale marks is variable:

```
to drawmarks :int
repeat 120 / :int [fd :int rt 90 fd 5 bk 5 lt 90]
end
```

Now write a procedure to draw the y-axis, giving it a scaling interval as input. (DrawMarks will be a subprocedure of the YAxis procedure.)

The SetHeading command

This procedure uses SetHeading (SetH for short) to change the turtle's heading. SetHeading sets the turtle's heading in absolute terms like a compass – 0 is straight up.

Draw the y-axis

```
to yaxis :scale
setheading 0                Sets the turtle's heading.
drawmarks :scale            Draws the marks.
bk 120
end
```

Try:

```
yaxis 20
cg yaxis 10
```

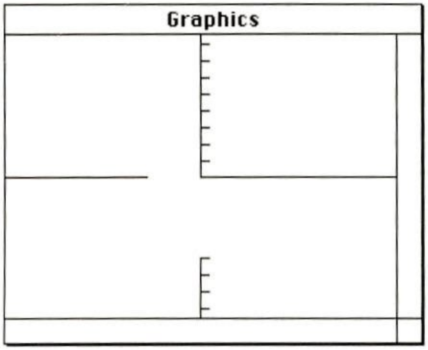
XAxis does not have a subprocedure, since it doesn't need marked intervals:

Draw the x-axis

```
to xaxis
setheading 90                       Sets the turtle's heading.
fd 200 bk 200
lt 90
end
```

Try it:

```
xaxis
```



The SetPos command

SetPos is a command that sets the turtle's position in terms of x and y coordinates. [0 0] is the center of the graphics window:

```
pu setpos [-100 -55] pd
```

Use the Pos operation to check the turtle's position:

```
print pos
```

Logo responds

```
-100 -55
```

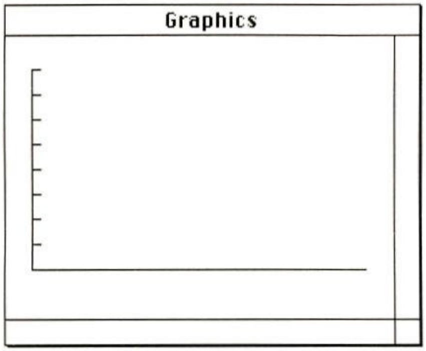
DrawAxes is the superprocedure. It sets the starting position for the axes, and runs YAxis and Xaxis.

Here is the definition for DrawAxes:

```
to drawaxes :startpos :scale
;x and y axes
;starting position is bottom left
pu setpos :startpos pd                    Sets the starting position
yaxis :scale
xaxis
end
```

Try out:

```
drawaxes [-100 -55] 15
```



Use SetPWidth to set the width of the bars. (The PWidth operation outputs the current pen width.)

Draw one bar

The procedure to draw one bar has :Height as its input. After the turtle draws a bar, the procedure returns the turtle's line to its normal width.

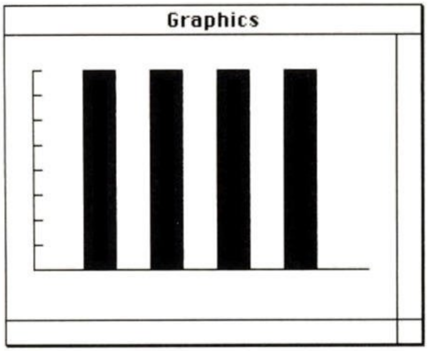
```
to bar :height
;draws a wide line for a bar
setpwidth 20                 Sets pen width to 20.
pu fd 10 pd                  Centers the wide pen above the axis.
fd :height
bk :height
pu bk 10 pd                  Centers the narrow pen on the axis.
setpwidth 1                  Sets pen width back to normal.
end
```

To place the turtle to draw each bar side by side, use the Position procedure.

```
to position :distance
;moves turtle to draw next bar
rt 90
pu fd :distance pd
lt 90
end
```

Now, type:

```
setup
drawaxes [-100 -55] 15
repeat 4 [position 40 bar 100]
```



Step 3: Determining the Bar Height

The ReadWord operation

ReadWord (RW for short) reads a word or a line of words typed at the keyboard and outputs the information to another procedure. For example, if you type:

```
print readword
```

Press Enter.

the insertion point waits at the beginning of the next line for you to type something. You may type:

```
echo
```

Press Enter.

Logo responds:

```
echo
```

Since ReadWord is an operation, it is used as an input to another procedure. In this case, the word you typed at the keyboard was given to Print. Print then printed the word.

You can name the output of ReadWord so that Logo will store it for future use. To do this, use Name:

```
name readword "message
```

You type:

```
Hello
```

This time, the word isn't printed again. When you type:

```
print :message
```

Logo responds:

```
Hello
```

The line typed at the keyboard was picked up by ReadWord and stored under the name Message. When you asked Logo to Print :Message, it printed the line.

In the bar graph project, ReadWord is used to pick up a number so this information can be converted into an input for Bar.

Now, write a procedure that uses ReadWord to get the number of computers that the company sold in a year. The year can be an input which will be passed from the superprocedure.

```
to baramount :year
;gets a number and draws a bar
print se [How many computers were sold in] :year
name readword / 100 "height       Gets an answer.
position 40                        Positions the turtle.
bar :height                        Draws the bar.
end
```

The Sentence operation

In BarAmount, Sentence (or Se) combines its inputs into a list.

To run BarAmount, give the year of your choice as input for now.

```
baramount 82
```

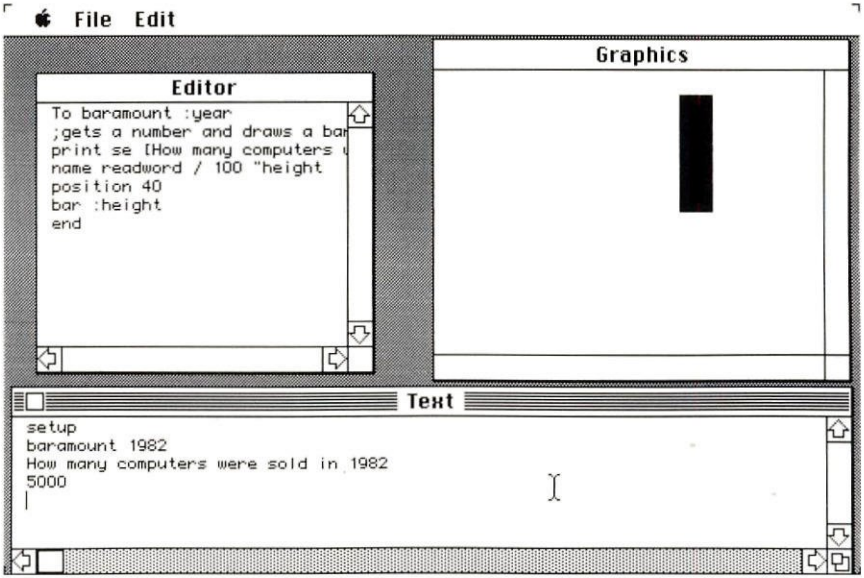
This question appears:

```
How many computers were sold in 82
```

If you type:

```
5000
```

Logo draws a bar 50 steps high (5000 / 100 = 50).



```
To baramount :year
;gets a number and draws a bar
print se [How many computers
name readword / 100 "height
position 40
bar :height
end
```

```
setup
baramount 1982
How many computers were sold in 1982
5000
```

Step 4: Labelling the Graph and the Bars

In any bar graph, it's a good idea to label the graph and the bars.

Print on the graphics window

Printing text on the graphics window is almost the same as printing on a text window. Since Logo normally prints on the current text window, a special command is needed to direct printing somewhere else. Use the SetWrite command:

The SetWrite command

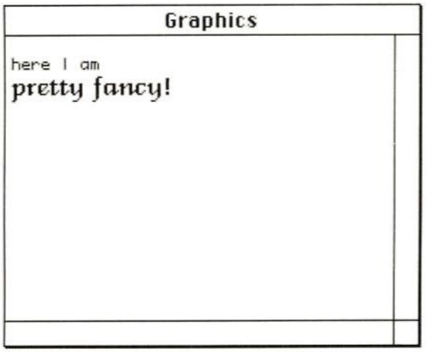
```
setwrite "graphics
print [here I am]
```

Graphics is the name of the window. After being given this instruction, the Print command prints its input on the graphics window. (If your graphics window is named something other than "Graphics", give its actual name as input to SetWrite.)

Change the printing font

On a graphics window, printing in any available font is possible:

```
setfont "Venice              Sets the printing  font.
setstyle [0 14]              Changes the printing style.
print [pretty fancy!]
```



here I am
pretty fancy!

```
setfont "Monaco              Restores the original font
setstyle [0 9]               and style.
```

For more information on fonts and printing styles, see Chapter 10, "Graphics", of the *Reference Manual*.

To return printing to the text window, type:

```
setwrite "text
```

The SetCursor command

Printing on the graphics window starts from the cursor position. The SetCursor command is used to position the cursor on the graphics window, just as SetPos positions the turtle. The GrPrint procedure takes two inputs: the word or list to print as a label on a graphics window, and the cursor's position.

Write a general graphics printing procedure

```
to grprint :position :label
setwrite "graphics           Sets the graphics window for printing.
setcursor :position          Sets the cursor's position.
print :label                 Prints the label.
setwrite "text               Restores the text window for printing.
end
```

If you want to place the title "Computer Sales" just above the bar chart, type:

Give a title to the graph

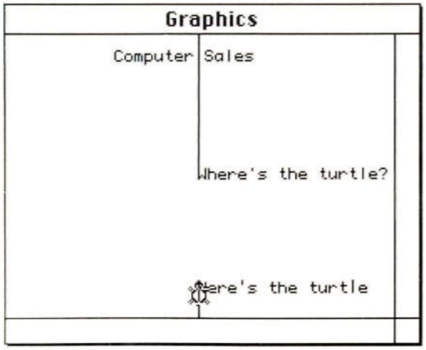
```
grprint [-50 70] [Computer Sales]
```

Now use the GrPrint procedure to put the year labels on the bars of the graph. GrPrint should be added as a subprocedure to BarAmount. The reason for this is that it's easy to calculate the position for printing. After each bar has been drawn, the turtle is on the x-axis of the graph, at the position of the bar. The Pos operation outputs the turtle's position:

```
print pos
```

The printing cursor can be moved to wherever the turtle is by using the output from Pos, as the input to SetCursor. Try it:

```
setwrite "graphics
setcursor pos                Sets the cursor to the turtle's position.
print [Where's the turtle?]
forward 100
setcursor pos                Sets the cursor to the turtle's position.
print [Here's the turtle]
```



Computer Sales

Where's the turtle?

Here's the turtle

On the graph, you want to print the label *below* each bar. That can be done by backing the turtle up 15 steps before setting the cursor, then moving it forward again.

Label the bars

```
to baramount :year
;gets a number and draws a bar
print se [How many computers were sold in] :year
name readword / 100 "height
position 40
bar :height
;labels the bar
pu bk 15 pd                Backs the turtle up.
grprint pos :year          Labels the bar.
pu fd 15 pd                Restores the turtle's position.
end
```

Step 5: Writing the Superprocedure

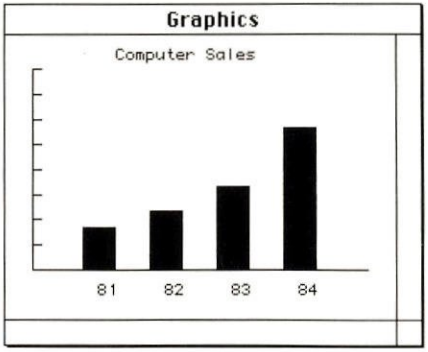
Finally, you need a superprocedure to put all the subprocedures together. Write this procedure using the years of your choice as inputs to BarAmount:

```
to bargraph
setup
drawaxes [-100 -55] 15
grprint [-50 70] [Computer Sales]
baramount 81
baramount 82
baramount 83
baramount 84
end
```

Now run it by typing:

```
bargraph
```

Enter any four numbers to draw the four bars. The resulting graph will depend on the inputs you give.



Step 6: Setting Up the Initial Windows

Once your program works, you may want to add some finishing touches. To make your new program a little more "user-friendly", it's a good idea to expand the SetUp procedure. It's possible that the graphics window has been changed in size or hidden behind other windows. Therefore, SetUp should:

1. Set the size and position of the graphics window, and clear it off.
2. Set the size and position of the text window, and clear it off.

The SetWSize command

The commands which set the size and position of windows are SetWSize (for Set Window Size) and SetWPos (for Set Window Position). SetWSize takes two inputs: a name, so it knows which window to move, and a list of two numbers, the width and the height of the window. If the graphics window named Graphics is still on the screen, try:

```
setwsize "graphics [200 200]
```

The window just became a 200 by 200 square. Notice that you set the size of the "document portion" of the window; that is, the part of the window that you can use, excluding the title bar and the scroll bars.

The SetWPos command

SetWPos also needs two inputs: the name of the window to be moved and its new location in x-y coordinates. However, the coordinates are not *graphics window* coordinates, they are *screen* coordinates. In screen coordinates, [0 0] is the top left corner of the screen. Try:

```
setwpos "graphics [80 40]
```

The window moved to the top of the screen (a y-coordinate of 40), near the left (an x-coordinate of 80). Again you set the position of the document portion of the window.

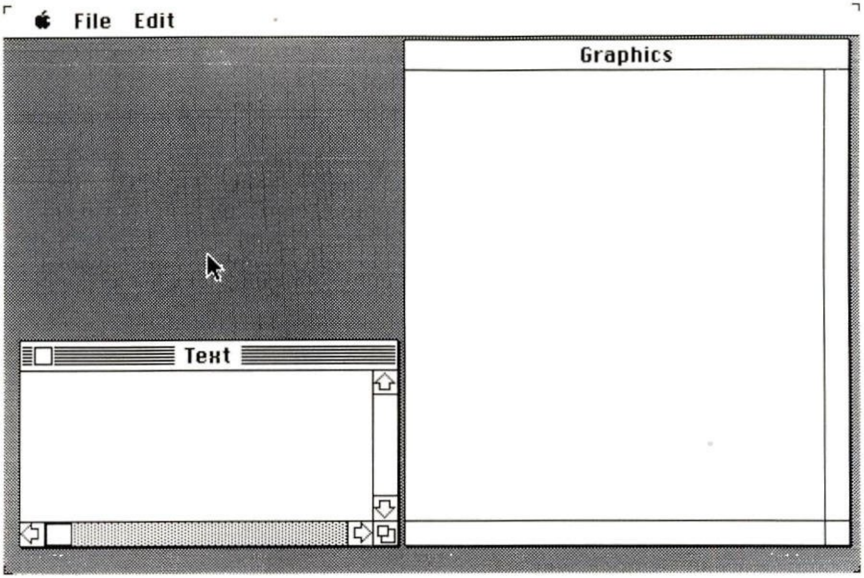
Edit SetUp to set the size and position of the graphics window:

```
to setup
recycle
ct
cg ht
setwpos "graphics [240 40]     Sets the location of Graphics.
setwsize "graphics [250 270]   Sets the dimensions of
end                            Graphics.
```

Try it out. To find out if Setup really works, move the window or make it smaller with the mouse first.

Now set the position and size of the text window. The name of the text window when Logo starts up is Text. If you're using a text window with another name, use that name as input. Edit SetUp again:

```
to setup
;clears and positions the windows
recycle
ct
cg ht
setwpos "graphics [240 40]
setwsize "graphics [250 270]
setwpos "text [10 220]    Sets the location of Text.
setwsize "text [210 90]   Sets the dimensions of Text.
end
```

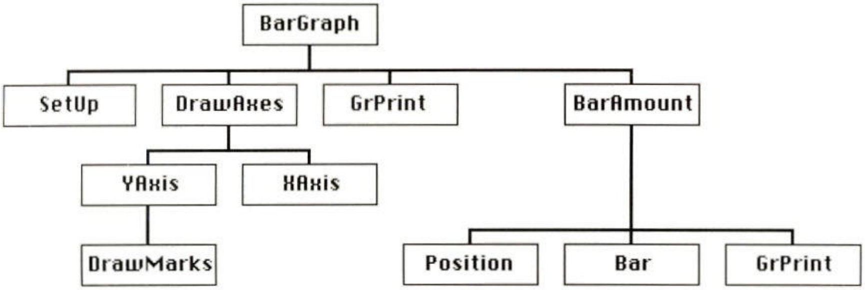


Program Listing

```
to bargraph
setup
drawaxes [-100 -55] 15
grprint [-50 70] [Computer Sales]
baramount 81
baramount 82
baramount 83
baramount 84
end

to setup
;clears and positions the windows
recycle
ct
cg ht
setwpos "grahics [240 40]
setwsize "graphics [250 270]
setwpos "text [10 220]
setwsize "text [210 90]
end

to drawaxes :startpos :scale
;x and y axes
;starting position is bottom left
pu setpos :startpos pd
yaxis :scale
xaxis
end

to yaxis :scale
setheading 0
drawmarks :scale
bk 120
end

to xaxis
setheading 90
fd 200 bk 200
lt 90
end

to drawmarks :int
repeat 120/:int [fd :int rt 90 fd 5 bk 5 lt 90]
end
```

```
to baramount :year
;gets a number and draws a bar
pr se [How many computers were sold in] :year
name readword / 100 "height
position 40
bar :height
;labels the bar
pu bk 15 pd
grprint pos :year
pu fd 15 pd
end

to position :distance
;moves turtle to draw next bar
rt 90
pu fd :distance pd
lt 90
end

to grprint :position :label
setwrite "graphics
setcursor :position
print :label
setwrite "text
end

to bar :height
;draws a wide line for a bar
setpwidth 20
pu fd 10 pd
fd :height
bk :height
pu bk 10 pd
setpwidth 1
end
```

Program Structure of BarGraph



Reflection

Operations

ReadWord and Sentence are operations, as are PWidth and Pos. An operation always produces an output that becomes the input to another procedure. Pos is used as an input to SetCursor. Unlike commands, operations can't be the sole instruction on a line.

The power of operations such as Pos is evident when they are used as inputs. Finding an exact position becomes unnecessary. In these cases, the computer does the work for you.

Some Notes on ReadWord

When using Readword, it is important to understand that Logo reads the line as a word, even if you type a series of words. For example, type:

```
name readword "anything
```

Press Enter.

then type:

```
this is a line with spaces
```

Now use the WordP operation to check if :Anything is a word:

```
print wordp :anything
```

Logo responds:

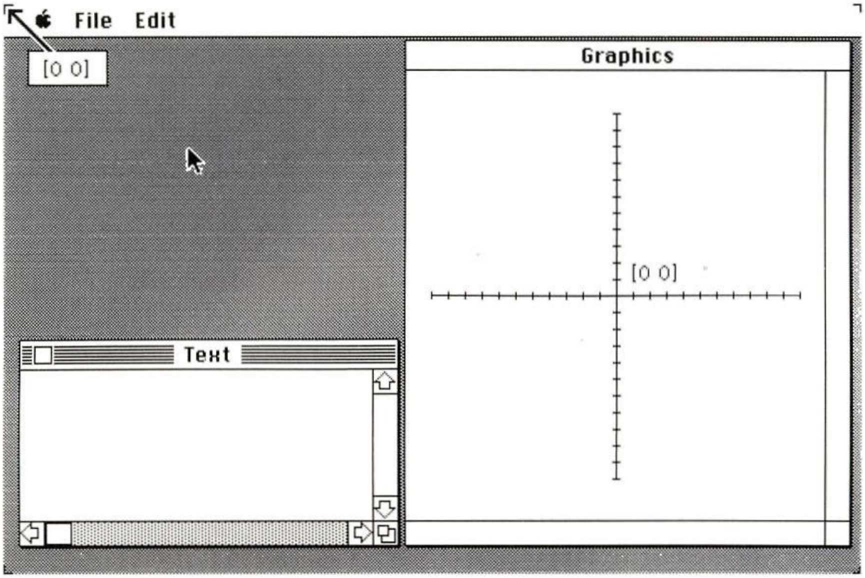
```
TRUE
```

Window Coordinates and Screen Coordinates

The graphics window always has the coordinates of [0 0] as its center, no matter what its size or location.

The screen has the coordinates of [0 0] on the top left corner. This means that the screen coordinate system looks quite different from the window coordinate system.

The screen coordinates are always the same, but since a graphics window may be large or small, anywhere on the screen, the window coordinates only have meaning relative to the center position.



Exploring Further

Write a bar graph procedure that is flexible enough to process a large amount of data. (Your program would have to divide the x-axis line into equal parts.)

Plot a line graph from peak to peak on the bars.

Modify BarGraph so the bars aren't drawn until all the data is collected, then:

1. Scale the y-axis to the height of the largest amount.
2. Calculate the data in terms of percentages.

Logo Vocabulary

Commands

SetCursor
SetFont
SetHeading   SetH
SetPos (for Set Position)
SetStyle
SetWPos (for Set Window Position)
SetWrite
SetWSize (for Set Window Size)

Operations

Pos (for Position)
PWidth (for Pen Width)
ReadWord   RW
Sentence   Se
WordP (P for Predicate)

Special Words

Venice
Monaco

9 Manipulating Text

Programs that build, analyze, and restructure words and sentences can be used as the basis for other projects such as questionnaires and quizzes. This chapter develops two "text manipulation" projects. In this context, new primitive procedures that act on words and lists are introduced, and you are shown how to write your own operations.

The first project is a random sentence generator, that generates sentences in the following form:

Dogs dance
Computers laugh
People bark
People beep

The second project analyzes and restructures words. Its function is whimsical: to change words ending with "ght" (for example, light) to end with "te" (lite). A phrase such as "light beer" becomes "lite beer".

Action

Generating Random Sentences

The random sentence generator produces sentences by combining words that are chosen randomly. This can produce interesting and amusing results.

This program can be written by following these steps:

Plan the steps

Step 1: Create two lists. Write a procedure that will pick words at random from a list.

Step 2: Write the procedure to generate sentences and then write the superprocedure.

Step 3: Extend the sentence generator by adding adjectives and adverbs.

Step 1: Creating Lists and Picking a Random Word

Before beginning, increase the size of the text window so you can see the results of your work. Then create a list of nouns and a list of verbs. Since Logo has to store them in memory, use Name to name them. Any words will do in these lists. These are just examples:

```
name [children dogs computers people] "nouns
name [laugh bark beep dance] "verbs
```

The Item operation

To select a word from a list, use Item:

```
print item 3 :nouns
```

Logo responds:

```
computers
```

```
print item 1 :verbs
```

Logo responds:

```
laugh
```

If you use Random as the first input to Item, you can select a random word:

```
print item random 4 :nouns
```

Try this line a few times. What happens if Random 4 outputs 0? Logo prints:

```
ITEM doesn't like 0 as input
```

To prevent Random from producing 0 as its output, add 1 to Random as in 1 + Random 4. The instruction line selecting a random word from a list can then be generalized into a procedure:

Define an operation to pick a word

```
to pick :list
output item (1 + random count :list) :list
end
```

The Output command makes Pick an operation. Output outputs an element of the input list extracted by Item.

The Pick procedure accepts a list of any length as its input by using the Count operation.

Count counts the number of elements in its input and outputs that number.

Pick is a useful tool. For example:

```
print pick :nouns
```

may print:

```
dogs
```

Try Pick again. You'll probably get a different result:

```
print pick :nouns
```

may print:

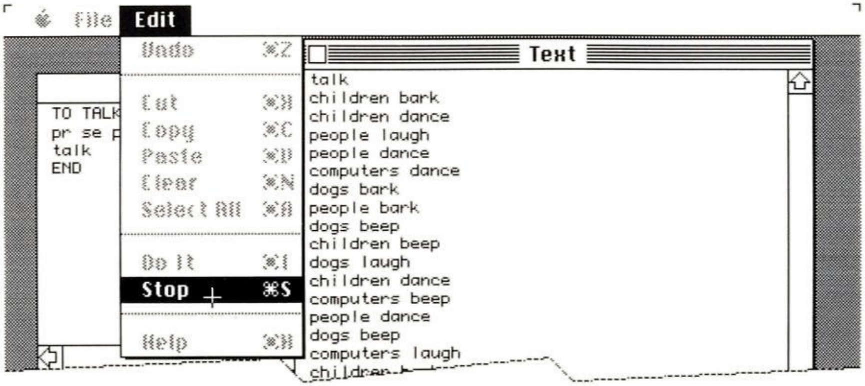
```
children
```

Step 2: Writing the Sentence Generator

This procedure uses Pick to randomly pick nouns and verbs and combine them in a sentence:

```
to talk
print se pick :nouns pick :verbs
talk
end
```

Since the procedure is recursive, it will print many random sentences until you choose Stop.



```
TO TALK
pr se p
talk
END
```

```
talk
children bark
children dance
people laugh
people dance
computers dance
dogs bark
people bark
dogs beep
children beep
dogs laugh
children dance
computers beep
people dance
dogs beep
computers laugh
```

Although the sentence generator appears complete, one problem remains: the noun and verb lists were not named in a procedure. This means that every time you run Talk, you have to check if :Nouns and :Verbs have values. It's more convenient to write a superprocedure that names the noun and verb lists, and runs Talk.

```
to randomsengen
name [children dogs computers people] "nouns
name [laugh bark beep dance] "verbs
talk
end
```

Step 3: Extending the Sentence Generator

At this point, you could make the sentences more interesting by adding adjectives and adverbs. In RandomSenGen, add adjectives and adverbs to the names. Put your choice of words in their lists.

```
to randomsengen
name [children dogs computers people] "nouns
name [laugh bark beep dance] "verbs
name [red blue green yellow] "adj
name [loudly quietly happily sadly] "adv
talk
end
```

The Wait Command

Talk must be edited to add adjectives and adverbs to the sentence printed. To slow down the sentences so they can be read, insert Wait before the recursive line. Wait makes Logo pause for the length of time given by its input in 60ths of a second. *Wait 60* makes Logo pause for 1 second.

```
to talk
pr (se pick :adj pick :nouns pick :verbs pick :adv)
wait 60
talk
end
```

Notice that when Se (Sentence) has more than two inputs, you must put parentheses around Sentence and its inputs.

Trying RandomSenGen may print funny combinations of words:

```
red dogs laugh quietly
blue computers beep loudly
green children dance happily
yellow people bark sadly
```

Generating a "Dialect"

The dialect generator takes a phrase or a sentence and changes all words ending with "ght" to end with "te".

To write this program, follow these steps:

Plan the steps

Step 1: Examine a word for "ght". Replace "ght" in a word with "te".

Step 2: Write a superprocedure to replace all the "ght" words in a list.

Step 1: Examining and Replacing Part of a Word

The MemberP operation

The simplest way to examine a word for the presence of a letter or group of letters is MemberP. MemberP is a predicate like EmptyP. MemberP (P for Predicate) checks if its first input (a word or list) is an element of its second input (a word or list). Try:

```
print memberp "k "macintosh
FALSE
```

The instruction tells Logo to check if the character *k* is in the word *macintosh*. It isn't there, so MemberP outputs False.

The ButLast operation

The last three letters will be removed from a word ending in "ght" by ButLast (or BL for short). ButLast outputs all but the last element of a word or list. ButLast will be used three times in a row; three ButLast's leave a word with the last three letters missing.

```
print bl bl bl "night
ni
```

The Word operation

Use Word to "glue on" the new ending "te". Word creates a new word made up of its inputs:

```
print word "ni "te
nite
```

Combining Word and ButLast in one instruction does the "cutting" and "pasting" in one step:

```
print word bl bl bl "night "te
nite
```

Define an operation to examine and replace

Examining and replacing part of a word can be the job of one procedure. The ChangeTag procedure is an operation that uses MemberP to check if a word ends with "ght". If "ght" is found, these letters are chopped off the word, and the letters "te" are added. If the word doesn't contain "ght", the input word is output without any change.

IfElse: a conditional

IfElse is a conditional like If, except it can run one of two instruction lists: the first instruction is run when the predicate or condition is True; the second, when it is False.

```
to changetag :wd
ifelse memberp "ght :wd→         Press Tab and Space bar.
 [op word bl bl bl :wd "te]→     Press Tab and Space bar.
 [op :wd]
end
```

Note To "format" lines so they carry across more than one screen line, press the Tab key instead of the Return key. A continuation arrow appears and you can put spaces at the beginning of the next line. Using Tab and spaces will increase the readability of long lines.

Test ChangeTag:

```
print changetag "alright
alrite                         The word with "ght" is changed.
print changetag "lettuce
lettuce                        A word without "ght" is left alone.
```

Step 2: Writing a Superprocedure to Replace Words in a List

You now can change a part of one word. The next step is to take a list and change only the relevant words. The superprocedure should take the list:

```
star light star bright
```

and change it to:

```
star lite star brite
```

Define a recursive operation

The Dialect superprocedure is a recursive operation that accomplishes this task. Dialect takes a list, passes one word at a time to ChangeTag, and outputs the updated list.

Dialect uses the primitive procedure FPut (for First Put) to create a list by putting its first input at the beginning of its second input, a list.

```
to dialect :list
;changes "ght words in list to "te    The comment line.
if emptyp :list [op [ ]]              The [ ] is an empty list.
op fput changetag first :list dialect bf :list
end
```

Now try Dialect:

```
print dialect [Star light star bright]
Star lite star brite
print dialect [I see the light]
I see the lite
```

Reflection

Global Variables

Did you notice that Talk doesn't have inputs on the title line, but still uses the lists named Nouns and Verbs? The lists were named with Name outside the procedure, but are accessible inside the procedure. This is because variables created with Name are *global*; that is, they are accessible to every procedure in the workspace.

Operations Written in Logo

Pick, ChangeTag, and Dialect are all operations. They are the first operations introduced that are not *primitive* procedures. Output is the command that makes these procedures operations. Output takes its input and sends it to another procedure.

Pick is very useful because it is used as an input. This is only possible because it is an operation. What would happen if Pick was a command? Replace Output with Print and experiment.

```
to picker :list
pr item (1 + random count :list) :list
end
```

```
picker [a b c d e f g]
c
```

Picker is static: we can't do anything else with it except look at the result.

Compare it with this:

```
print se pick [A B C] pick [a b c]
B c
```

Recursive Operations

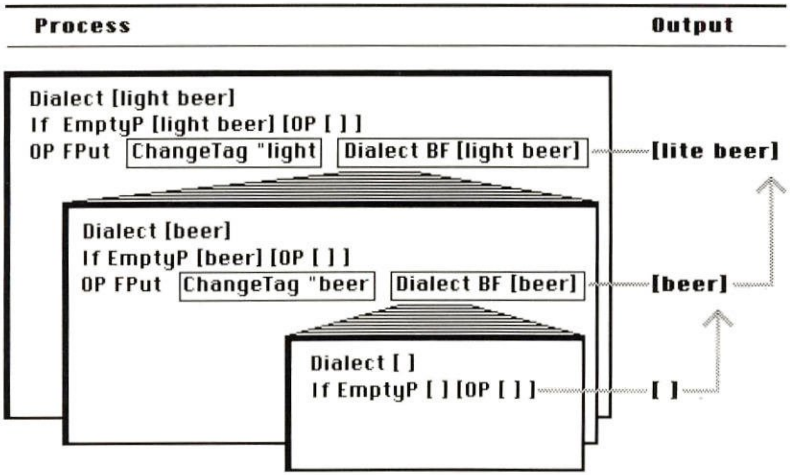
Unlike Pick and ChangeTag, Dialect is recursive. When a recursive procedure acts as an operation, the passing of information occurs not only between the recursive procedure and its superprocedure, but also between each round of recursive subprocedures.

In the Dialect procedure, the line:

```
op fput changetag first :list dialect bf: list
```

is difficult to understand. The telescoping model provides a visual representation of the process of running the Dialect procedure in the instruction:

```
print dialect [light beer]
```



Dialect [light beer] runs two subprocedures, ChangeTag "light and Dialect [beer]. Although ChangeTag outputs something immediately, Dialect [beer], in turn, runs two subprocedures, ChangeTag "beer and Dialect [ ]. When Dialect [ ] runs, it outputs the empty list. This is passed to FPut in Dialect [beer].

Now dialect [beer] can output the result of [beer] to FPut in Dialect [light beer]. Dialect [light beer] then outputs the result of [lite beer] to Print so the instruction can be printed.

Remember that a recursive call is a subprocedure call. Each subprocedure must finish running before the result of the instruction line is output.

Exploring Further

Write a program that generates rhyming poetry, using RandomSenGen as a model.

Using Dialect as a model, write a program changing all words ending in "es" to "ing". Write a program that changes past tense verbs to present tense.

Logo Vocabulary

Commands

Output OP
Wait

Special Keys

Tab (for formatting)

Operations

ButLast BL
Count
FPut (for First Put)
IfElse
Item
MemberP (P for Predicate)
Word

10 Building a Phone Directory

Use a property list

One of the uses of a programming language is to keep records. A large set of records, stored in files, is often called a *data base.* Building a data base in Logo can be accomplished in different ways. One way, to be shown in this chapter, is through the use of property lists. A *property list* is a list of attributes and values, associated with a name. It takes this form:

name [property1 value1 property2 value2 property3 value3...]

For instance, look at the table or desk you're working at. It has a name (desk) and a number of attributes or "properties". It has the property of color. The color may be brown. In that case:

name [property1 value1]
desk [color brown]

The desk also has the properties of height, depth and width. With values for these properties added, a property list for a desk might look like:

desk [color brown height 28″ depth 30″ width 62″]

Property lists are very useful for storing data by name and property. The following program will use peoples' names as the properties and phone numbers as values, to create a phone directory that is a large property list which can be printed out, saved in a file, and easily updated or added to.

You can modify the phone directory later to include anything you wish; for example, addresses, birthdays, and favorite colors.

The project will be developed in the following steps:

Plan the steps

Step 1: Entering the data in the form of a property list. This section will be interactive.

Step 2: Printing out the phone directory clearly on the graphics window.

Step 3: Updating the phone directory so that numbers can be changed or added.

Action

Step 1: Entering the Data

The PProp command

Suppose you want the phone book identified by your name ("Laura's Phone Book"). The properties (your friends' names) and values (their phone numbers) can then be stored under your name. PProp (which stands for Put Property) gives a name a property and a value. To give your friend Eric a phone number:

```
pprop "Laura "Eric [373-5655] Laura is your name.
                              Eric is your friend.
                              373-5655 is Eric's phone
                              number.
```

You could go on and enter your other friends' phone numbers in this manner, but it's much more convenient to write a procedure to put the data in the form of a property list for you.

PhoneList asks for your name and your friends' names, then runs a subprocedure to add the phone numbers:

Use a procedure to enter data

```
to phonelist
;builds a phone book using a property list
pr [What's your name?]
name readword "myname      Picks up your name.
pr se [List your friends' names please,] :myname
name readlist "namelist  Picks up a list of friends.
addnumbers :namelist     Adds the phone numbers.
end
```

AddNumbers is a recursive procedure that asks for each friend's phone number in turn, and puts the name and phone number into the property list form of a property and value:

```
to addnumbers :namelist
if emptyp :namelist [stop]
pr se word first :namelist "'s [number is: ]
pprop :myname first :namelist readlist
addnumbers butfirst :namelist
end
```

Try out:

```
phonelist
```

Logo responds:

```
What's your name?
```

You may type:

```
Laura
```

Remember to press Enter.

Logo says:

```
List your friends' names please, Laura
```

You may type:

```
Eric Judy Lorraine Alain
```

Type them all before pressing Enter.

Then Logo says:

```
Eric's number is:
```

You may type:

```
373-5655
```

And so on, until all the names in the list of friends have numbers.

Step 2: Printing Out the Phone List

The PList operation

You can view the property list you just created by using PList (for Property List).

```
print plist "Laura
```

This prints the property list associated with Laura.

A list similar to this will be printed:

```
Eric [373 - 5655] Judy [738 - 1212]
Lorraine [212 - 8888] Alain [767 - 9999]
```

That's fine but hard to read. The list could be printed in columns:

```
Eric           373 - 5655
Judy           738 - 1212
Lorraine       212 - 8888
Alain          767 - 9999
```

You need to take the property list apart in order to print it. This can be the job of the recursive procedure, ColumnPrint.

Print the phone list in columns

ColumnPrint uses the PadRight operation to print words or numbers in a fixed number of spaces. This is how columns of information are printed. No matter how many characters there are in a number or word, PadRight will output it with a fixed number of characters, "padding" the spaces to the right of the next word will be printed in a certain place. PadRight's first input is the amount of spaces allotted to the column; the second input is a word or list.

```
to columnprint :props
;prints names and phone numbers in columns
if emptyp :props [stop]
type padright 15 first :props
pr first butfirst :props
columnprint butfirst butfirst :props
end
```

```
columnprint plist "Laura
```

You should see something like this:

```
Eric           373 - 5655
Judy           738 - 1212
Lorraine       212 - 8888
Alain          767 - 9999
```

Display the phone list on the graphics window

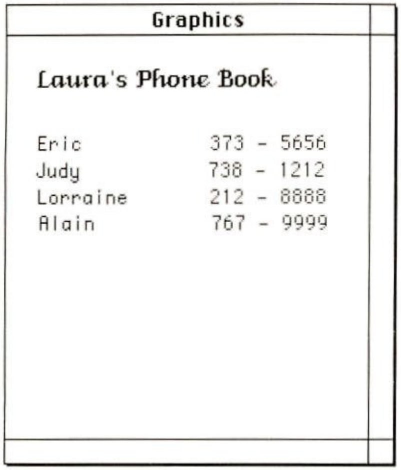
A really elegant phone directory program would print the directory listing on a graphics window, in a fancy font, perhaps with the title in a different font from the listing.

ShowList is the procedure that prints the listing on the graphics window (ColumnPrint becomes its subprocedure). At the same time, give the display a title (for example, Laura's Phone Book):

```
to showlist :phonelist
;displays the phone list on the graphics window
cg
setwrite "graphics          Sets printing to graphics window.
setcursor [-90 60]          Positions the cursor at top left.
setfont "Venice             Sets the letter font.
setstyle [0 14]             Set the printing style.
(pr word :myname "'s [Phone Book])
pr [ ]
setfont "Monaco
setstyle [0 12]
columnprint :phonelist      Phone listing.
setstyle [0 9]              Restores original style.
setwrite "text              Restores printing to text window.
end
```

Try:

```
showlist plist "Laura
```



```
Laura's Phone Book

Eric          373 - 5656
Judy          738 - 1212
Lorraine      212 - 8888
Alain         767 - 9999
```

At this point, add ShowList to the original PhoneList superprocedure. After entering the names and phone numbers, you will see everything listed.

```
to phonelist
;builds a phone book using a property list
pr [What's your name?]
name readword "myname
pr se [List your friends' names please,] :myname
name readlist "namelist
addnumbers :namelist
showlist plist :myname
end
```

Step 3: Adding and Changing Listings in the Phone Directory

The GProp operation

This program must have the ability to change a phone number, or add a friend's name and number. Changing an existing phone number can be easily done using GProp (for Get Property) and PProp.

For instance:

```
print gprop "Laura "Judy
```

prints Judy's phone number:

```
738 - 1212
```

Then:

```
pprop "Laura "Judy [654 - 1111]
```

changes Judy's phone number. You can check whether Judy's number has been changed by:

```
print gprop "Laura "Judy
```

or

```
showlist plist "Laura
```

Note To find Judy's phone number with GProp, Judy must be typed exactly as it was originally entered, in capital and lower case letters.

Write a procedure to update the phone list

Now, write a procedure to change or add a phone number interactively. Update displays the current phone list, asks whose number you want to add or change, makes the change, then displays the updated list:

```
to update
;updates the existing phone list
showlist plist :myname
pr [Whose number do you want to change or add?]
name readword "name
(pr word :name "'s [old number is]→
 gprop :myname :name)
pr [What's the new number?]
pprop :myname :name readlist
showlist plist :myname
end
```

To try out the program, type:

```
update
```

Erase the data

To erase the complete phone book and start fresh, use the command ErasePList (for Erase Property List), as in:

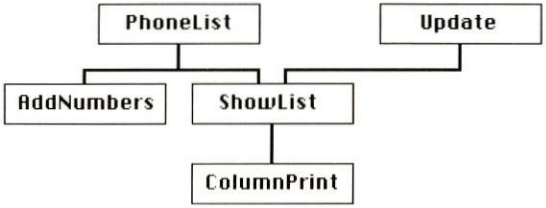
```
eraseplist plist "Laura
```

Program Listing

```
to phonelist
;builds a phone book using a property list
pr [What's your name?]
name readword "myname
pr se [List your friends' names please,] :myname
name readlist "namelist
addnumbers :namelist
showlist plist :myname
end

to addnumbers :namelist
if emptyp :namelist [stop]
pr se word first :namelist "'s [number is: ]
pprop :myname first :namelist readlist
addnumbers butfirst :namelist
end
```

```
to showlist :phonelist
;displays the phone list on the graphics window
cg
setwrite "graphics
setcursor [-90 60]
setfont "Venice
setstyle [0 14]
(pr word :myname "’s [Phone Book])
pr [ ]
setfont "Monaco
setstyle [0 12]
columnprint :phonelist
setstyle [0 9]
setwrite "text
end

to columnprint :props
;prints names and phone numbers in columns
if emptyp :props [stop]
type padright 15 first :props
pr first butfirst :props
columnprint butfirst butfirst :props
end

to update
;updates the existing phone list
showlist plist :myname
pr [Whose number do you want to change or add?]
name readword "name
(pr word :name "’s [old number is]→
 gprop :myname :name)
pr [What’s the new number?]
pprop :myname :name readlist
showlist plist :myname
end
```

Program Structure of PhoneList



Reflection

The Elements of a List

It may be confusing to distinguish the elements of a property list. The property list

[Eric [373 - 5655] Judy [738 - 1212] Lorraine [212 - 8888]]

has 6 elements. The first element of this list is Eric, and the sixth is [212-8888].

The organization of this list makes writing a recursive procedure to print each element fairly simple. ColumnPrint prints two elements of its input at each round of the recursive call. Since there are two elements printed, the recursive line (ColumnPrint ButFirst ButFirst :PList) uses two ButFirst's to drop off two elements each time.

Replacing an Element in a Property List

An interesting aspect of property lists is the way elements in the list can be accessed and replaced. The way to access elements in standard Logo lists is by using word and list operations (First, ButFirst, Item, etc.). Instead, with property lists, you can access the value of a property directly using GProp, and replace it using PProp. This makes property lists easier to update, since an element is replaced or added by name, not by its placement within a list.

Exploring Further

Add addresses and birthdays to the phone list program.

Examine other ways of creating a data base in the sample programs (look at the Samples Menu file).

Logo Vocabulary

Commands

ErasePList ErPL (for Erase Property List)
PProp (for Put Property)
Type

Operations

GProp (for Get Property)
PadRight
PList (for Property List)
ReadList RL

A Concluding Note by Seymour Papert

When I have learned something new, I am full of questions. Some about the subject matter: what have I learned, where do I go from here? Some about learning: what kind of a learning experience was this, what did I learn about learning? Some about myself and other people: have I learned something new about myself and my relationship with other people?

I hope Logo has left your head buzzing with such questions. Of course I can't answer them for you but a few guidelines might help.

The only way to find out what you have learned is to use it. You have seen examples of Logo programs. Try your hand at inventing some of your own. If you are a cautious person, start by making first small, and then larger, modifications to our programs. If you are a risk taker, try something very different. Both routes can take you a long way.

Whichever route you take, you must not expect, or want, everything you try to work out. You will be experimenting with your knowledge of Logo, testing and extending its limits and finding out what style fits you best.

When your projects don't work out, take a hard look at the reasons. If you get a large number of inexplicable error messages, you are probably missing a fundamental concept. Perhaps you should go through this guide again trying to write some simple programs, quite similar to our examples. If your procedures run but don't do what you hoped they would, you can take two tacks. One is to stand back from the project, rethink your goal and start again with a more carefully structured plan. Or you can stick with your partially working program and develop it through understanding its strengths and weaknesses.

You are at a particularly exciting point when your procedures run, but you may suspect there are better ways to get the same results. You are ready to go on to learn more Logo than we have shown in this guide. I have four pieces of advice about how to do this.

The first is to dip into the *Reference Manual*. Logo has many more primitive procedures than you have seen in this guide and there are ways to understand the language more deeply than those shown so far. Read Chapter 2, "Logo Grammar", of the *Reference Manual* carefully. You can browse through the rest. Treat the manual like a dictionary. Whenever you use a Logo primitive procedure, look it up. Read its description and skim through the procedure descriptions in the same section.

Another place for browsing is the collection of sample programs on your Logo disk or Logo programs in the growing literature on Logo. You should learn a computer language like you would learn a natural language: first and foremost by expressing yourself in it but also by reading it. Read programs as well as writing them.

My third piece of advice for how to get a deeper understanding of Logo has already been stated several times: write lots of programs, learn by doing.

And finally the most important advice of all is THINK ABOUT your program. Best of all find someone to talk to, to think with. One learns best by doing... and by thinking about what one has done.

Seymour Papert

Seymour Papert

Other Books About Logo

Here are some other books that have been written about Logo. They can provide ideas for projects and additional information on the concepts and philosophy of Logo. Check your bookstore for more books.

Apple® Logo, by Harold A. Abelson. Published by Byte Books, McGraw-Hill, 1982.

Turtle Geometry: The Computer as a Medium for Exploring Mathematics, by Harold A. Abelson and Andrea diSessa. Published by MIT Press, 1981.

Logo for Apple® Computers, by Roger W. Haigh and Loren E. Radford. Published by John Wiley and Sons, Inc., 1984.

Mindstorms: Children, Computers, and Powerful Ideas, by Seymour Papert. Published by Basic Books, 1980.

Introducing Logo, by Peter Ross. Published by Addison-Wesley, 1983.

Discovering Apple® Logo, An Invitation to the Art and Pattern of Nature, by David Thornburg. Published by Addison-Wesley, 1983.

Index

( ), 93
<, 64
>, 63
=, 63
;, 52
:, 42
[ ], 11
", 11
+, 16
/, 16

addition +, 16
AddNumbers, 102

Back Bk, 8
Backspace key, 10
Bar, 73
BarAmount, 75, 79
BarGraph, 80
brackets [ ], 11
bugs, 17
ButFirst BF, 60
ButLast BL, 94

Cancel, 35
CG (Clear Graphics), 9
ChangeTag, 95
Clear, 10
close box, 16
closing windows, 16
colon :, 42
ColumnPrint, 104
commands, 30
Comment ;, 52
condition, 63
coordinates, 72
Copy, 26
copying the master disk, xi
Count, 91
CT (Clear Text), 10
Curl, 62
cursor position, 78
Cut, 26

defining
  operations, 91
  procedures, 21
  procedures with inputs, 42
  recursive operations, 98
  recursive procedures, 49, 57
Demo Menu, x
demonstration programs, x, 6
Dialect, 96
division /, 16
Do It, 3
DrawAxes, 73
DrawMarks, 71

Edit, 25
editing a procedure, 25
Editor, 23
editor window, 23
empty list, 61
EmptyP (P for Predicate), 61
empty word, 61
End, 23
Enter key, 3
equals =, 63
ErAll (Erase All), 36
EraseFile ErF, 37
ErasePList ErPL (Erase Property List), 107
EraseProc ErP (Erase Procedure), 34
erasing
  files, 37
  graphics, 9
  procedures, 34
  property lists, 107
  text, 10

False, 63
FatSquares, 32
files
  erasing, 37
  listing, 35
  loading, 37
  saving, 35
file space, 39
FillSh (Fill Shape), 12
fonts
  Monaco, 77
  Venice, 77
formatting, 95
Forward Fd, 8
FPut (First Put), 96

global variables, 97
GProp (Get Property), 106
Graphics, 2
graphics window, 2
greater than >, 63
GrPrint, 78

Heading, 58
Help, 6
HideTurtle HT, 7
Home, 9
House, 29

If, 63
IfElse, 95
inputs, 8, 17, 41
insertion point, 3
interactive program, 67
Item, 90

Left Lt, 8
less than <, 64
List, 11
Load, 35
loading
  files, 37
  windows, 37

Many, 44
MemberP (P for Predicate), 94
Monaco, 77
Move, 27, 28

Name, 59
naming a procedure, 29

Open Editor, 22
Open Window, 13
opening windows, 13
operations, 30, 85
Output Op, 91
outputs, 30

PadRight, 104
parentheses ( ), 93
Paste, 26
PenDown PD, 9
PenErase PE, 9
PenReverse PX, 32
PenUp PU, 12
PhoneList, 102, 106
Pick, 91
Picker, 97
PList (Property List), 103
Poly, 49, 59, 60
POProc POP (Print Out Procedure), 34
Pos (Position), 72
Position, 74
POTS (Print Out Titles), 33
PProp (Put Property), 102
predicate, 63
primitive procedures, ix, 29
Print Pr, 11
printing
  on graphics window, 77
  pictures, 29
printing out
  procedures, 34
  titles, 33
procedures
  defining, 21
  editing, 25
  erasing, 34
  naming, 29
  primitive, ix, 29
  recursive, 49, 57
  subprocedures, 28, 30
  superprocedures, 28, 30
Procedures, 34
property lists, 101
PWidth (Pen Width), 73

Quit, 36
quitting Logo, 36
quotation mark “, 11

Random, 27, 91
RandomSenGen, 92, 93
ReadList RL, 102
ReadWord RW, 74, 85
recursion, 55, 65
recursive line, 49
recursive procedures, 49, 57
Recycle, 57
Repeat, 11
Return key, 3
Right Rt, 8

Save, 35
saving
  windows, 37
  workspace, 35
screen coordinates, 81, 86
scroll bar, 34
Sentence Se, 76
SetCurrent, 15
SetCursor, 78
SetFont, 77
SetHeading SetH, 71
SetPos (Set Position), 72
SetPPattern (Set Pen Pattern), 12
SetPWidth (Set Pen Width), 32
SetStyle, 77
SetUp, 70, 81, 82
SetWPos (Set Window Position), 81
SetWrite, 77
SetWSize (Set Window Size), 81
ShowList, 105
ShowTurtle ST, 7
Sky, 28, 45, 46
Spi, 52, 53, 58
SpinSquare, 25
SpiralIn, 64
Square, 23, 42
Star, 26, 44
starting Logo, 1
Stop, 50
stop rules, 57
subprocedures, 28, 30
Sun, 51
SuperPoly, 59, 60
superprocedures, 28, 30
Surprise, 62

Tab key (formatting), 95
Talk, 92, 93
Text, 2
text window, 2
title line, 23
To, 23
Total Turtle Trip, 30, 55
Tri2, 63
Triangle, 60, 61
True, 63
turtle, 7
Type, 104

Update, 107

variables
  global, 97
  inputs, 41
Venice, 77

Wait, 93
window coordinates, 86
windows
  closing, 16
  editor, 23
  graphics, 2
  opening, 13
  text, 2
word, 11
Word, 95
WordP, 86
workspace, 33
writing procedures, 21

XAxis, 72

YAxis, 71

Microsoft® MacLibrary™

Product Problem Report

10700 Northup Way, Box 97200, Bellevue, WA, 98009

Use this form to report software problems, documentation errors, or suggested enhancements. Please send the form to Microsoft MacLibrary.

Name ____

Street ____

City ____________________ State __________ Zip __________

Phone Number (   ) _______________ Date _______________

MacLibrary Product Name ______________________________

Version Number ____________ Registration Number ____________

Category of Problem

____ Software Problem

____ Documentation Error
(Document and page number)

____ Suggested Enhancement

____ Other

Description of Hardware

____ 128K Macintosh

____ 512K Macintosh

____ Lisa 2/10

____ Other

____ 2nd Disk Drive

____ Other Peripherals

____ Hard Disk Manufacturer__________________ Size_____

Problem Description

Please describe or attach a listing of the problem.

Can the problem be duplicated? _____ Yes _____ No

Microsoft License Agreement

CAREFULLY READ ALL THE TERMS AND CONDITIONS OF THIS AGREEMENT PRIOR TO OPENING THIS DISK PACKET. OPENING THIS DISK PACKET INDICATES YOUR ACCEPTANCE OF THESE TERMS AND CONDITIONS.

If you do not agree to these terms and conditions, return the unopened disk packet and the other components of this product to the place of purchase and your money will be refunded. No refunds will be given for products that have an opened disk packet or missing components.

1. LICENSE: You have the non-exclusive right to use the enclosed program. This program can only be used on a single computer. You may physically transfer the program from one computer to another provided that the program is used on only one computer at a time. You may not electronically transfer the program from one computer to another over a network. You may not distribute copies of the program or documentation to others. You may not modify or translate the program or related documentation without the prior written consent of Microsoft.

YOU MAY NOT USE, COPY, MODIFY, OR TRANSFER THE PROGRAM OR DOCUMENTATION, OR ANY COPY, EXCEPT AS EXPRESSLY PROVIDED IN THIS AGREEMENT.

2. BACK UP AND TRANSFER: You may make one (1) copy of the program solely for backup purposes. You must reproduce and include the copyright notice on the backup copy. You may transfer and license the product to another party if the other party agrees to the terms and conditions of this Agreement and completes and returns a Registration Card to Microsoft. If you transfer the program, you must at the same time transfer the documentation and backup copy or transfer the documentation and destroy the backup copy.

3. COPYRIGHT: The program and its related documentation are copyrighted. You may not copy the program or its documentation except for backup purposes and to load the program into the computer as part of executing the program. All other copies of the program and its documentation are in violation of this Agreement.

4. TERM: This license is effective until terminated. You may terminate it by destroying the program and documentation and all copies thereof. This license will also terminate if you fail to comply with any term or condition of this Agreement. You agree, upon such termination, to destroy all copies of the program and documentation.

5. HARDWARE COMPONENTS: Microsoft product hardware components include only Microsoft circuit cards, power supplies, product housings, and electrical cords.

6. LIMITED WARRANTY: THE PROGRAM IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE PROGRAM IS ASSUMED BY YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU (AND NOT MICROSOFT OR ITS DEALERS) ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR, OR CORRECTION. FURTHER, MICROSOFT DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE OF, OR THE RESULTS OF THE USE OF, THE PROGRAM IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS, OR OTHERWISE; AND YOU RELY ON THE PROGRAM AND RESULTS SOLELY AT YOUR OWN RISK.

Microsoft does warrant to the original licensee that the disk(s) on which the program is recorded be free from defects in materials and workmanship under normal use and service for a period of ninety (90) days from the date of delivery as evidenced by a copy of your receipt. Microsoft warrants to the original licensee that the hardware components included in this package are free from defects in materials and workmanship for a period of one (1) year from the date of delivery to you as evidenced by a copy of your receipt. Microsoft's entire liability and your exclusive remedy shall be replacement of the disk or hardware component not meeting Microsoft's Limited Warranty and which is returned to Microsoft with a copy of your receipt. If failure of the disk or hardware component has resulted from accident, abuse, or misapplication of the product, then Microsoft shall have no responsibility to replace the disk or hardware component under this Limited Warranty. In the event of replacement of the hardware component, the replacement will be warranted for the remainder of the original one (1) year period or thirty (30) days, whichever is longer.

THE ABOVE IS THE ONLY WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE THAT IS MADE BY MICROSOFT ON THIS MICROSOFT PRODUCT. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS AND YOU MAY ALSO HAVE OTHER RIGHTS THAT VARY FROM STATE TO STATE.

NEITHER MICROSOFT NOR ANYONE ELSE WHO HAS BEEN INVOLVED IN THE CREATION, PRODUCTION, OR DELIVERY OF THIS PROGRAM SHALL BE LIABLE FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, OR INCIDENTAL DAMAGES ARISING OUT OF THE USE, THE RESULTS OF USE, OR INABILITY TO USE SUCH PRODUCT EVEN IF MICROSOFT HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES OR CLAIM. SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES, SO THE ABOVE LIMITATION MAY NOT APPLY TO YOU.

7. UPDATE POLICY: In order to be able to obtain updates of the program, the licensee and persons to whom the program is transferred in accordance with this Agreement must complete and return the attached Registration Card to Microsoft. IF THIS REGISTRATION CARD HAS NOT BEEN RECEIVED BY MICROSOFT, MICROSOFT IS UNDER NO OBLIGATION TO MAKE AVAILABLE TO YOU ANY UPDATES EVEN THOUGH YOU HAVE MADE PAYMENT OF THE APPLICABLE UPDATE FEE.

8. MISC.: This License Agreement shall be governed by the laws of the State of Washington and shall inure to the benefit of Microsoft Corporation, its successors, administrators, heirs, and assigns.

9. ACKNOWLEDGEMENT: YOU ACKNOWLEDGE THAT YOU HAVE READ THIS AGREEMENT, UNDERSTAND IT, AND AGREE TO BE BOUND BY ITS TERMS AND CONDITIONS. YOU ALSO AGREE THAT THIS AGREEMENT IS THE COMPLETE AND EXCLUSIVE STATEMENT OF AGREEMENT BETWEEN THE PARTIES AND SUPERCEDES ALL PROPOSALS OR PRIOR AGREEMENTS, VERBAL OR WRITTEN, AND ANY OTHER COMMUNICATIONS BETWEEN THE PARTIES RELATING TO THE SUBJECT MATTER OF THIS AGREEMENT.

Should you have any questions concerning this Agreement, please contact in writing Microsoft Corporation, Customer Sales and Service, 10700 Northup Way, Box 97200, Bellevue, WA 98009.

CAREFULLY READ THE MICROSOFT LICENSE AGREEMENT
ON THE FRONT OF THIS PACKET BEFORE OPENING!

The program on the enclosed disk(s) is licensed to the user.

By opening this packet, you indicate your acceptance of the Microsoft License Agreement.

Part No. 999-999-870

Three Important Reasons to Register Your Product Now

1 Microsoft Help Hotline

The help you need, whatever your application. We want you to get the maximum performance from your Microsoft software. If you have any technical problem, we'll be glad to help. However, most of the time you'll find the answer right in your product documentation, so please take a look at that first. You might also give your Microsoft dealer a call. If you're still puzzled, gather all the information pertinent to the problem and call our Product Support staff at (206)828-8089. They'll be ready to give you the support you need to get the most from your Microsoft software.

2 Microsoft Product Replacement Plan

If you need it, when you need it. In spite of rigorous testing and the highest quality-control standards, even Microsoft products sometimes need replacement. If your product proves defective, it will be replaced at no charge during the warranty period, and for a reduced price thereafter. However, you must provide us with proof-of-purchase and return the defective component to us.

If you think you may have a defective product, you'll probably want to call our help hotline at (206)828-8089 before mailing the product to us. When you have confirmed that a problem exists, follow the instructions outlined in the attached product replacement card. Mail the card, the defective component, your proof-of-purchase, and full details about the problem you are experiencing to:

Customer Service Department
Microsoft Manufacturing
13221 S.E. 26th Street
Bellevue, WA 98005

Or call Microsoft Customer Service at (206)828-8088 for more information.

3 Microsoft Product Upgrade Plan

It keeps your program up to date. Your Microsoft software product uses the most advanced technology available today. But we continually improve our software, making it even more powerful and easy to use. You can take advantage of our ongoing research—if you send in your registration card today!

As a registered Microsoft user, you receive announcements about major improvements in your program. These announcements give you the cost of the update and ordering procedures. In most cases the enhanced version is available to you at a reduced price. Only registered owners receive these special update notices. (Microsoft offers updates only for its productivity tools and languages. Recreational software is not eligible for updates. Owners of recreational products do not receive update announcements.)

Just Register Now.

Microsoft Software Limited Warranty

- The disk on which your Microsoft program is recorded is warranted to be free of defects in materials and workmanship under normal use for a period of 90 days from date of product purchase.

- This limited warranty applies to the original purchaser only and to the recording medium (disk) only, not to the information encoded on it. This warranty covers disks included in Microsoft hardware/software packages, such as the Microsoft® SoftCard® system products and the Microsoft® RAMCard® memory board for the IBM® PC.

Microsoft Hardware Limited Warranty

- Microsoft hardware components include only circuit cards and the mechanical mouse.

- If a hardware component is included with your Microsoft product, the component is warranted to be free of defects in materials and workmanship under normal use for a period of one year from date of product purchase.

- This limited warranty applies to the original product purchaser only and to the hardware component only, not to the application for which it is used.

Disclaimer of Liability for Use and the Results of Use

The Microsoft programs are licensed solely on an ''as is'' basis. The entire risk as to their quality and performance is assumed by the purchaser. MICROSOFT CORPORATION DOES NOT GUARANTEE, WARRANT, NOR MAKE ANY REPRESENTATION REGARDING THE USE OF, OR THE RESULTS OF THE USE OF, THE PROGRAMS IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS, OR OTHERWISE; AND THE PURCHASER RELIES ON THE PROGRAMS AND THE RESULTS SOLELY AT HIS OR HER OWN RISK. Microsoft Corporation assumes no liability for any direct, indirect, incidental or consequential, special or exemplary damages, regardless of its having been advised of the possibility of such damages.

A full description of the limited warranty for hardware and software and the terms of the above disclaimer of liability are in the License Agreement that accompanies this booklet.

New Answers from the Oldest Name in Microcomputer Software

We wrote the first BASIC for the very first personal computer. Today it's the world's most widely installed computer language, running on more than two million machines. The same care and attention we put into Microsoft® BASIC, we put into every product we sell today. The result?

Microsoft consistently delivers powerful, reliable, easy-to-use solutions for business, industry, and education.

Register It Now!

Solutions like MS-DOS™, the most popular operating system for 16-bit computers. Like Microsoft Multiplan®, the flexible, plain-English electronic worksheet. Or like Microsoft Word, the writing system that's revolutionized word processing.

Every Microsoft product is designed to be easy to learn and use, and to take full advantage of your computer's capabilities. Your new Microsoft program incorporates the most advanced concepts in software today, to give you peak performance and to unlock the power of your machine.

MICROSOFT®
The High Performance Software™

Product Replacement Order Card

Please use this card when ordering a replacement for a defective Microsoft product. Mail it with the defective component(s) to the address below. To validate a replacement request for a product under limited warranty, include proof-of-purchase. A product returned without proof-of-purchase is not eligible for warranty service.

If the product warranty has expired, or if the product does not qualify for warranty service, you will be charged a service fee. No out-of-warranty service will be performed prior to receipt of payment. You may include credit card information if you would like to charge the service. You may call Microsoft Customer Service at (206)828-8088 to inquire about the current charge for the service required.

1-800-321-5646

882-8080

8:30-6:00 Eastern

Name

(Please include all information required for delivery including company name, mailstop, and apartment or suite number, if applicable.)

Address

Street

City State Zip Country

Phone ( ) Telex

Registration number on disk

Name of product as it appears on package

Date of product purchase / /

Month Day Year

Reason for return

If the warranty has expired, I authorize you to charge my credit card. Charges vary. The minimum service charge is $25.00. ☐ American Express ☐ Visa ☐ MasterCard

Credit card number ☐☐☐☐ ☐☐☐☐ ☐☐☐☐ Expiration date

Authorized signature

Mail to: **Customer Service Department**
Microsoft Manufacturing
13221 S.E. 26th Street
Bellevue, WA 98005

Microsoft Corporation
10700 Northup Way
Box 97200
Bellevue, WA 98009

0984 Part No. 999-999-886

MICROSOFT®

Microsoft Corporation
10700 Northup Way
Box 97200
Bellevue, WA 98009

0485 Part No. 080-096-001