

# LOGO PROFESSOR™ USER GUIDE

Developed by  
UNISON WORLD, INC.

## ACKNOWLEDGEMENTS

Jay Fenlason  
Unix Logo Specialist  
Unison World, INC.

Fred Fisher  
Senior Systems Programmer  
Unison World, INC.

Nathaniel Stitt  
Senior Systems Programmer  
Unison World, INC.

Special Thanks To  
Hong Liang Lu  
*President* Unison World, Incorporated  
COMPAI

## CONTENTS OF THE PACKAGE

The LOGO PROFESSOR package contains the following items:

- The LOGO PROFESSOR User Guide
- Program diskette
- Data diskette
- Owner registration card

This User Guide is an instructional and reference manual for The LOGO PROFESSOR.

It will guide you through the basic steps for operating the program.

The return of the owner registration card ensures your eligibility for any updates, replacement diskettes, or announcements.

**NOTE:** The LOGO Professor is protected under copyright law. Copying of any portion of the disks is against federal regulations. The user cannot legally make a back-up disk of the program. If there are any problems with the purchased disk, return it to the dealer or distributor within 90 days for a free replacement disk. After 90 days, contact:

**Acorn Publishing Company**  
1335 West 134th Street  
Gardena, CA 90247

Acorn Publishing will replace the disk for a nominal charge.

## **PREFACE**

Welcome to LOGO PROFESSOR, the computer language designed to allow you to take full advantage of your Epson QX-10. LOGO PROFESSOR can teach you graphics, and programming, painlessly. Above all you will learn how to interface with your computer.

LOGO PROFESSOR is an up-to-date enhancement of LOGO, a language that was developed (primarily as an educational tool) at MIT in the sixties. All levels of users—from children to experienced computer professionals—will enjoy and find immediate use for LOGO PROFESSOR.

LOGO PROFESSOR is extensively documented on two media: the on-line help screen, which will walk you through all functions of LOGO; and the hard copy documentation found in this manual. The manual contains:

- An introduction to LOGO PROFESSOR (Chapter 1).
- A detailed chapter on Using LOGO, which may be read as a supplement to the Help Screens (Chapter 2).
- A collection of sample programs, which will enable you to begin producing interesting graphics immediately (Chapter 3).
- A glossary of key words and commands (Chapter 4).
- A series of appendices that include:

Helpful Hints  
Recursion  
Error Messages

Whether you are new to computers or a seasoned programmer, LOGO PROFESSOR will provide you with a fascinating addition to your current body of knowledge.

# TABLE OF CONTENTS

Chapter	Page
1	INTRODUCTION . . . . . 1
	Logo Professor Overview . . . . . 1
	Getting Started . . . . . 3
	Handling Diskettes . . . . . 4
2	USING LOGO PROFESSOR . . . . . 5
	Introduction to LOGO PROFESSOR . . . . . 6
	Turtle Graphics . . . . . 9
	Procedures and Names . . . . . 17
	Text and Lists . . . . . 24
	Conditionals, Logic and Tests . . . . . 34
	Disk File and Memory Management . . . . . 40
	Advanced Procedures and Concepts . . . . . 44
	Categorization of Primitives . . . . . 55
	SAMPLE PROCEDURES . . . . . 59
	GLOSSARY . . . . . 66
	Function Keys . . . . . 66
	Control and Arrow Keys . . . . . 68
	Primitives Glossary . . . . . 70
	Glossary of Terms . . . . . 91
	Bibliography & References . . . . . 96
APPENDIX	
A	HELPFUL HINTS . . . . . A-1
B	EXAMPLE OF RECURSION . . . . . B-1
C	ERROR MESSAGES . . . . . C-1
D	TURTLE COMPASS . . . . . D-1
E	HELP SCREEN INDEX . . . . . E-1
USER GUIDE INDEX	

# Chapter 1

## INTRODUCTION

### **LOGO PROFESSOR Overview**

Welcome to LOGO PROFESSOR, the computer language that is both simple and powerful. As a new user of LOGO PROFESSOR, you are about to enter a computer environment that holds new and surprising possibilities for you. Consider the following features of LOGO PROFESSOR:

#### **Graphics**

With very little training, you will learn from LOGO PROFESSOR to produce a variety of stunning, intricate graphic displays on your terminal screen. By using LOGO PROFESSOR, you will take full advantage of the impressive graphics capabilities of your Epson QX-10. As you watch the Turtle move, you will be observing your own parallel development as a computer graphics artist.

#### **A Programming Language**

Although LOGO PROFESSOR is ideal for beginners, it contains features that make it suitable for advanced applications as well. You will notice that LOGO PROFESSOR is a programming language that lends itself to the easy-to-use structured approach, in which large programs are broken down into smaller, more manageable blocks. And, you will quickly adapt to LOGO PROFESSOR's sophisticated math and graphic manipulations. Experienced programmers will appreciate these features, while the beginner will quickly become a proficient user of them.

#### **An Educational Tool**

LOGO PROFESSOR is a valuable asset for students at all levels. It gives the elementary-level pupil the chance to understand—through exploration and play—the fundamentals of geometry. As he creates increasingly complex designs, the younger student is introduced to mathematical equations. The high school or college student will find in LOGO PROFESSOR a valuable tool for plotting and illustrating equations and functions throughout all of his upper-level mathematics. LOGO PROFESSOR also gives all students practical exposure to computer operations and logic. With LOGO PROFESSOR, the student learns by doing.

## **Background**

LOGO was developed in the late sixties by Seymour Papert and a team of researchers at MIT. It was originally created as an educational language, one that would allow children and beginners to interact comfortably and confidently with computers. Since its creation in 1968, the original LOGO program has been under continual development. Throughout this development it has become more powerful and easier to use. Acorn Publishing is proud to present what it considers to be the state-of-the art LOGO in LOGO PROFESSOR.

## GETTING STARTED

Before you begin using LOGO PROFESSOR check to make sure that the contents of your LOGO PROFESSOR package are complete.

### Contents of the Package

The LOGO PROFESSOR package consists of two diskettes and the LOGO PROFESSOR User Guide. The User Guide is of course the manual that you are presently reading. The diskettes are described below:

- **LOGO PROFESSOR Left** — Program Disk—This diskette contains all LOGO PROFESSOR software and on-screen (HELP) documentation. The program disk is protected so that you cannot change or delete any information on it.
- **LOGO PROFESSOR Right** — Data Disk—You will use this diskette to store all LOGO procedures that you create.

### Loading LOGO PROFESSOR

To load LOGO PROFESSOR onto your Epson QX-10 processor, simply proceed as follows:

- Insert the Left — Program Disk into the left disk drive.
- Insert the Right — Data Disk into the right disk drive.
- Press the button marked PUSH for both drives.
- The screen will display a select screen which give you the option of displaying the help screens, or proceeding directly to LOGO PROFESSOR (see Chapter 2, Using LOGO PROFESSOR).

**NOTE:** The system should respond as described above in approximately 30 seconds. If there is no response, you must restart the system by pressing the RESET button located below the right disk drive.



## **HANDLING DISKETTES**

The floppy disks that you use on your QX-10 are very sensitive magnetic media, and it is important that you handle them with caution and care. Follow the rules below when handling diskettes.

- DO NOT touch the exposed magnetic surface of a diskette.
- ALWAYS store the diskette in its protective sleeve and stand it on edge to prevent damage.
- DO NOT store a diskette in sunlight, heat, or cold.
- DO NOT bend, fold, or staple.
- DO NOT attach paper clips or rubber bands to a diskette.
- DO NOT open the disk drive when the red light is on.
- DO NOT write on the diskette with a pen or pencil. If marking is necessary, use a felt tip pen, and press lightly.
- DO NOT expose the diskette to any magnetic surface, such as telephones, your printer, or a typewriter. Such exposure can destroy data!
- ALWAYS insert the diskette into the disk drive with care.

As a general rule, simply make sure that you always handle the diskettes with extreme care. Remember that if a disk is damaged or destroyed, so is any data that is stored on it.

## Chapter 2

# USING LOGO PROFESSOR

When you are first introduced to LOGO PROFESSOR, you will be asked if you would like help or if you would like to work with LOGO on your own. If you wish to access the help screens, simply press the HELP function key, and the following screen will display:

**Choose by number one of the following**

1. **INTRODUCTION to LOGO**
2. **TURTLE GRAPHICS**
3. **PROCEDURES and NAMES**
4. **TEST and LISTS**
5. **CONDITIONALS, LOGIC and TESTS**
6. **DISK FILES and MEMORY MANAGEMENT**
7. **ADVANCED TECHNIQUES**
8. **INDEXES to HELP SCREENS**

**Push the number of your choice for more HELP**  
**Push STOP key to go to LOGO**

Within Help you have two options as described below:

The first option is full help instruction (options 1 through 7 on the menu) which will walk you through a lesson on working with LOGO. LOGO PROFESSOR will explain each function within that topic. After each lesson, a sample procedure will show you how the newly introduced primitives can be used.

The second option (option 8 on the menu) will list a series of terms and commands and allows you to access them individually. You need only select the option from the index and LOGO PROFESSOR will go directly to that screen. When you are working with LOGO PROFESSOR and find that you need help, simply press the HELP function key on the keyboard and both full and indexed help will be available.

This chapter of the User Guide is designed to be used in conjunction with the LOGO PROFESSOR help instructions that appear on your screen. The following sections of the chapter correspond to the selections on the Main help menu. Please note that all primitives are specified in lower case, just as you will key them in (in the text, they are highlighted in bold face—e.g., **forward**). All function keys are in upper case (e.g., STORE key) or the actual key is depicted.

# INTRODUCTION TO LOGO

LOGO PROFESSOR is one of the most powerful and user-friendly languages implemented on a personal computer. Simple and easy-to-understand commands are used to build user-defined procedures, which pave the way to creating powerful programs. In this section you will be introduced to the basic LOGO PROFESSOR terms and concepts.

## Turtle Graphics

Creating graphics images with LOGO is referred to as Turtle Graphics. When you first begin to work with LOGO PROFESSOR, you will learn to instruct the LOGO Turtle to draw graphics images. This is accomplished with simple commands such as **forward**, **back**, **right** and **left**.

## Primitives

The commands that we mentioned (**forward** and **back**) are just two of those offered by LOGO PROFESSOR. Commands that are a part of LOGO PROFESSOR's vocabulary are called primitives. The LOGO PROFESSOR vocabulary is not limited to graphics-oriented commands. Some primitives are also available which enable you to work with words and lists.

## Procedure

You can create your own commands in LOGO Professor by defining a procedure, which is simply a set of commands or primitives that are grouped together to perform a specific function. There is virtually no limit to the flexibility of LOGO PROFESSOR and the procedures that you can define.

## Workspace

LOGO PROFESSOR sets aside an area in the computer's memory for you to work in. This area is called workspace. Workspace is used to store and perform your instructions. It is also used by LOGO to perform internal system functions. Workspace is erased when you exit LOGO or turn off your system. For this reason, you must **store** to disk all procedures that you have created. When you want to save the work that you have done, you can save everything in (or a portion of) workspace so that you can retrieve the information later. To save your procedures, simply press the STORE key on the keyboard (the STORE key executes the **store** primitive). After pressing STORE, you must specify a filename to identify the group of procedures that are currently in workspace. These procedures will be stored on disk under this name. The STORE function will be discussed in more detail in Section 6 of this chapter, Disk Files and Memory Management.

When you wish to access any of the procedures that have been saved on disk, simply press the RETRIEVE key on the keyboard and specify the same filename where the procedures were stored. The RETRIEVE key initiates the **retrieve** primitive, which loads the file (see Chapter 4 Glossary, **retrieve**). LOGO PROFESSOR will then load the procedures into your workspace. If you are not sure of the name that you assigned to the file when you were saving the workspace, simply press the INDEX key on the keyboard. All LOGO PROFESSOR files will be displayed on the screen. The INDEX key executes the **index** primitive (see Chapter 4).

The help screens that are available to you through the HELP key on the keyboard can be printed on your printer by turning the printer on. To do this, press the PRINT key on the keyboard. As the help screens display on the screen, they will simultaneously print out to the printer. The PRINT key on the keyboard acts as a switch. When you are through printing, turn the printer off by pressing the PRINT key again. The PRINT key executes the primitive **pon** (printer on) the first time you press the key and **poff** (printer off) when you press the PRINT key a second time (see Chapter 4, Glossary, for definitions of these primitives).

Other function keys are available under LOGO. These are discussed in Section 6, of this chapter, Disk Files and Memory Management and in Chapter 4, Glossary.

Let's recap the terms and concepts introduced in this section.

- **Command** — An instruction that LOGO PROFESSOR understands. This term can be used to refer to a primitive or a user-defined procedure.
- **Filename** — A name of eight or fewer characters given to a group of procedures stored on a floppy disk.
- **Primitive** — A pre-defined command in LOGO PROFESSOR. A complete list of LOGO Professor primitives are outlined in Chapter 4, Glossary.
- **Procedure** — A user-defined command containing a series of primitives.
- **Workspace** — The temporary working area available to the user to perform and create procedures.



- Lists all LOGO files stored on the diskette in the right (or B) drive. This key executes the **index** primitive.



- Turns the printer on and off. This key executes the **pon** and **poff** primitives.



- Loads procedures from disk. Use the same name that you entered when you saved the workspace using **store**. LOGO PROFESSOR will prompt you to enter the filename. This key executes the **retrieve** primitive. The format is:

> **retrieve** "filename



- The key that is pressed to save workspace (or selected procedures) on disk. LOGO PROFESSOR will prompt you to enter a file name to save the workspace under. This name should be no more than 8 characters long, and should not contain any periods. The STORE key executes the **store** primitive. The format is:

> **store** "filename "procedure name

## TURTLE GRAPHICS

The LOGO Turtle is the triangular shape that appears in the center of your screen (shown below):



The name, Turtle, refers back to the days when the designers of the LOGO language used an actual three-dimensional motorized “turtle” on a piece of paper to develop graphics.

To draw graphics with LOGO PROFESSOR, all you have to do is instruct the LOGO Turtle to move. For example, entering **forward** 100 tells the turtle to move forward 100 steps. **Right** 90 tells the turtle to turn right 90 degrees. If this instruction is repeated four times, the turtle will create a square. It is that easy to create graphics with LOGO PROFESSOR.

### Screen Modes

Before we begin drawing, we should briefly discuss screen modes. LOGO PROFESSOR has three screen modes, described below:

- **textscreen** — **Textscreen** is the first screen that you see when you enter LOGO. This screen is blank except for the > symbol (the LOGO PROFESSOR prompt), which appears in the upper left corner of the screen. You will use **textscreen** for entering text (see Section 4, of this chapter, Procedures and Names). Key:

> **textscreen**

- **splitscreen** — **Splitscreen** is a composite screen that enables you to display the turtle (in the body of the screen) while you enter text (in the bottom five lines of the screen). To get to **splitscreen** mode, Key:

> **splitscreen**

- **fullscreen** — **Fullscreen** mode devotes the entire screen to the graphics display, with no lines reserved for the display of text. This screen is used primarily when you are running your predefined procedures, (see Section 4, Procedures and Names), which require no entry of text. Key:

> **fullscreen**

Pressing the STYLE key on your keyboard enables you to switch to any of the three screen modes. The style key switches from **fullscreen** to **splitscreen** to **textscreen**.

## Turtle Movement Primitives

Turtle Graphics, simply put, requires you to instruct the turtle to move. You do this by entering primitives. When you first begin working with turtle graphics, work in splitscreen mode so that you can view the commands that you enter, as well as the movements of the turtle (as the turtle moves, it will draw a line). We briefly discussed the command **forward** (you may abbreviate this to **fd** when you key it in). Remember that when you instruct the turtle to move forward, you must also tell it the number of steps to move (e.g., **fd 100**). You may also use the primitive called **back** (or **bk**) to move the turtle back. Again, you must specify the number of steps that the turtle is to move (e.g., **bk 50**).

The turtle can also turn **right** (**rt**) and **left** (**lt**). For example, **right 90** tells the turtle to turn 90 degrees to the right. **left 180** will turn the turtle in the exact opposite direction. It is a good idea to practice using these commands before you move on to other primitives. Remember that all LOGO PROFESSOR primitives must be entered in lower case, or they will not be understood. Let's review the commands we have introduced so far:

- **forward (fd)** — Moves the turtle forward the number of steps specified.
- **back (bk)** — Moves the turtle back the number of steps specified.
- **right (rt)** — Turns the turtle to the right the number of degrees specified.
- **left (lt)** — Turns the turtle to the left the number of degrees specified.

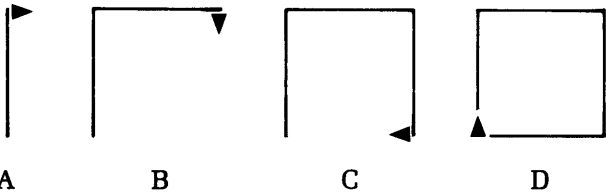
Now that you know how to move the turtle, you may draw a simple image, a square, by entering the following:

A > **fd 25 rt 90**

B > **fd 25 rt 90**

C > **fd 25 rt 90**

D > **fd 25 rt 90**



While these commands are simple, it is a bit tedious to type the same command four times. The following section explains how to avoid this problem with the **repeat** Primitive.

## The Repeat Primitive

The **repeat** primitive enables you to instruct LOGO PROFESSOR to repeat a statement a specified number of times, rather than rekeying it. The command is used as follows:

```
>repeat 4 [fd 100 rt 90]
```

As shown above, **repeat** is followed by the number of times you want the instruction repeated, then within square brackets the list of commands that you want repeated. The first time that you use **repeat**, you may not get the exact graphic display that you had planned. In this case, try rekeying the commands separately, so that you can check the turtle's movements.

As an exercise, try the **repeat** statements shown below to see how easy it is to produce graphics.

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

```
>repeat 180 [repeat 4 [fd 50 rt 90] rt 2]
```

The first **repeat** statement shown above produces a circle. The second example produces an image that is an enhancement of a square (notice the **repeat** statement within a **repeat** statement).

## Pen Control Primitives

LOGO PROFESSOR'S pen control primitives enable you to control the visibility of the turtle and the line that is drawn in Turtle Graphics. For example, if you want to move forward 20 steps, but, you don't want to draw a line, you may use the **penup** primitive (**pu**). Just key **penup**, then enter your **forward** command. To put the pen back down, use the **pendown** command (**pd**). If you do not wish to see the turtle while it is drawing, you may hide it by entering the **hideturtle** (or **ht**) command. When you want to see it again, enter the **showturtle** (or **st**) command.

The LOGO PROFESSOR turtle also has an eraser that you can attach to it by entering the command, **penerase** (or **pe**). Let's say that you have an image on the screen, and you want to erase one part of the image. You would simply enter **penerase**, then instruct the turtle to move back over the line that you want to remove. Remember to reset the pen when you are finished erasing by entering **pendown** (or **pd**).

**Penreverse** (or **px**) command instructs the turtle to reverse whatever image is on the screen as the turtle moves over it; if the screen is blank, the turtle will draw a line; if the turtle crosses over a line, it will erase it. Again, remember to reset the pen when you are finished by using **pendown** (or **pd**).



The pen control primitives are summarized below:

- **hideturtle** (**ht**) — Instructs the LOGO PROFESSOR turtle to disappear. You can request the Turtle to appear again using the **showturtle** (or **st**) command.
- **pendown** (**pd**) — Instructs the LOGO PROFESSOR turtle to put the drawing pen down.
- **penerase** (**pe**) — Erases any line that the turtle crosses over.
- **penreverse** (**pr**) — Instructs the turtle to reverse the pen so that, if there is an image on the screen, wherever the turtle crosses a line, that point will be erased, if there is no image, a line will be drawn.
- **penup** (**pu**) — Instructs the LOGO PROFESSOR turtle to pick up the pen and not draw a line until the **pendown** command is given.
- **showturtle** (**st**) — Instructs LOGO PROFESSOR to show a previously hidden turtle.

## Clearing the Graphics Screen

Now that you have had a chance to experiment with Turtle Graphics, your screen is probably filled with lines. How do you clear the graphics screen? This is done by entering the **clearscreen** (or **cs**) primitive. This command will clear the screen and return the turtle to the home position in the center of the screen. If you want to clear the screen but leave the turtle where it is, use the primitive **clean**. If you want to leave the contents of the screen intact, but move the turtle back to the home position, enter **home**. The primitives to clear the graphics screen and move the turtle home are summarized below:

- **clean** — This command will clear the screen without moving the turtle.
- **clearscreen** (**cs**) — This command will clear the screen and return the turtle to the home position.
- **home** — This command will not clear the screen, but will return the turtle to the home position. If the pen is down, a line will be drawn.

## Screen Border Primitives

There are some screen border primitives that will enable you to put new variations and controls into the graphics that you create. These are: **fence**, **window** and **wrap**.


The **fence** primitive places an invisible border on the screen, causing an error condition if the turtle runs into it (LOGO PROFESSOR will display the error message, "turtle out of bounds"). As its name implies, using this primitive is like putting a fence on the screen.

The second screen border primitive is **window**, which allows the turtle to travel beyond the boundaries of the graphics screen. Note that **window** allows the turtle to move around on a virtually infinite screen. You will be able to see the turtle only when it passes by the window, and to do this you will have to instruct it to move back into the screen boundaries.

The third screen border primitive is **wrap**, and it does just what its name suggests. That is, if you give the turtle an instruction to go outside of the screen boundaries, this primitive will cause it to wrap around to the opposite side of the screen. You will be in **wrap** mode when you first enter the LOGO graphics screen.

As an example of how the turtle reacts to these three screen border primitives, consider the command "**fd 250.**" From the home position, this command will cause the turtle to go out of the screen boundaries. In **fence** screen mode, LOGO PROFESSOR will display the error message "turtle out of bounds". In **window** screen mode, the turtle will disappear past the screen border. (You may move the turtle back within the screen boundaries to view it.) In **wrap** mode, the turtle will move off of the top of the screen and wrap around to the bottom of the screen.

The **SIZE** key is provided to allow you to switch to any of the three screen border modes. Let's review the commands that have been introduced.

- **fence** — Prevents the turtle from traveling beyond the display screen boundaries.
- **window** — Allows the turtle to move outside of the display screen. Displays only what was drawn within the display screen.
- **wrap** — Wraps around any lines that are drawn over the display screen limitations.
-  — This key switches to the three screen borders **window**, **fence**, or **wrap** (in that order).

## Draw to the Printer

The DRAW key on the keyboard has been enabled to allow you to print graphics images on the printer. This function key is like a switch: when you press it once, it turns the graphics print capabilities on; when you press it a second time, it turns the printer off.

## Basic Math Functions

The last topic to be covered in this section is the mathematical capabilities of LOGO PROFESSOR. LOGO understands all of the basic math commands. For example, if you enter:

```
>12 + 8
```

and press the RETURN key, LOGO PROFESSOR will print "Result: 20" on the next line. The following list covers all of the basic math commands that LOGO PROFESSOR understands.

**+** Plus. When numbers are entered before and after this sign, LOGO PROFESSOR adds the numbers together.

```
>2 + 2  
Result: 4
```

**-** Minus. If a number is entered before and after this sign, LOGO PROFESSOR subtracts the second number from the first number.

```
>2 - 2  
Result: 0
```

**\*** Multiplication. If a number is entered before and after this sign, LOGO PROFESSOR multiplies the first number by the second.

```
>2 * 2  
Result: 4
```

**/** Division. If a number is entered before and after this sign, LOGO PROFESSOR will divide the first number by the second number.

```
>2 / 2  
Result: 1
```

=

Equals. Checks to see if two values are equal.

>2 + 2 = 5

Result: false

>2 + 2 = 4

Result: true

This symbol can also be used with words.

>"orange = "orange

Result: true

>"orange = "apple

Result: false

You will learn more about how LOGO works with words in Section 4 of this chapter, Procedures and Names.

<

Less than. If you enter 2 < 1, LOGO PROFESSOR displays:

>2 < 1

Result: false

>

Greater than. If you entered 2 > 1, LOGO PROFESSOR would display:

>2 > 1

Result: true

()

Parentheses. Used to change or clarify the sequence of the operations. LOGO PROFESSOR first computes the items within parentheses, then completes the rest of the equation. You may also use parentheses to specify the sequence of operations of statements containing primitives. This feature enables you to group a series of instructions together. The use of parentheses is illustrated by the following examples.

>1 + 2 \* 6

Result: 13

>(1 + 2) \* 6

Result: 18

The results of the two equations are different because, as explained, when no parentheses are used, multiplication and division are performed first, followed by addition and subtraction.

The math keys shown are enabled on the math key pad on the right side of your keyboard, as well as on the regular character keyboard.

The sequence of operations of the math operators, indicating when they will be performed in an equation, is as follows:

1. Multiplication, division.
2. Addition, subtraction.
3. Greater than, less than, equals.

This is the order in which these operations will be performed in an equation.

## PROCEDURES AND NAMES

### Creating Procedures

So far, you have learned to maneuver the turtle, and to create images by entering series of instructions. The next step is to learn to store these instructions so that you may create your image by entering only a single word. This is called creating a procedure. The following commands allow you to create a procedure:

```
to
edit
```

At this point, you should only use the **to** primitive (Refer to Chapter 4 Glossary for a full explanation of the command **edit**). To create a procedure, key **to** followed by a name that is meaningful to you. For example, if you are creating a square, assign the procedure name to be **square**. LOGO PROFESSOR will then display the following screen:

```
to square
■
end
```

The name that you assign to the procedure should be as descriptive as possible. However, we recommend that the name be eight characters or fewer. Please remember also that all procedure names should be unique.

After you enter **to** followed by your procedure name, the cursor will be positioned on the second line. This is called edit mode, distinct from command mode where every command that you enter is executed at the time that you enter it, edit mode saves the commands and executes them when the procedure name is entered. Every procedure must begin with the word **to** and end with the word **end**. You will notice that in edit the **>** LOGO prompt is no longer displayed. At this point you should type in the commands necessary to create a square, as shown below:

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



When you are ready to save a procedure, press the STORE key on the keyboard. The STORE key has two independent functions. When pressed in edit mode, the STORE key will save (or define) the procedure in the user workspace that LOGO provides. LOGO will print a message:

Defining square

And return to command mode in textscreen.

When pressed in command mode, the STORE key will cause LOGO PROFESSOR to wait for a filename to be entered, and will then save all of the workspace to the diskette in the right disk drive.

If for some reason you wanted to leave the procedure without storing it, you could do that by pressing the STOP key. (The STOP key may also be used in command mode to pause or stop a procedure that is running. This function of the STOP key will be discussed again later.) Let's review the concepts discussed so far:










- **edit (ed)** — Same as **to**. This command is used to create a procedure in LOGO PROFESSOR.
-  — When this function key is pressed in edit mode, LOGO PROFESSOR will exit the procedure, ignoring any changes made. If the procedure was new, it will not be defined.
-  — Pressing STORE in edit mode will save a procedure in workspace. In command mode it will save the workspace to disk.
- **to** — Key this command, followed by a procedure name, to enter edit mode and create a procedure.

## Entering Text in Edit Mode


Now that you understand the commands that are used to create a procedure let's discuss how you enter the commands in edit mode. First we will tell you how to move the cursor in edit mode. Then we will define some of the keys on your keyboard that you will be using in edit mode (control keys).

The arrow keys move the cursor one character in the direction (both horizontal and vertical) that they are pointing. If you hold the shift key down at the same time as the up or down arrow, the cursor will page forward or back an entire screen. Holding the shift key down at the same time as the left arrow will move the cursor to the beginning of the line. Holding the shift key down at the same time as the right arrow will move the cursor after the last character on the line.

Following is a list of some of the keys that you will use when in edit. This list includes control keys, so called because you must hold the CTRL key down at the same time as the key specified.

-  — Moves to the right one character. In edit mode, if the SHIFT key is held down at the same time as the right arrow key, the cursor will move to the end of the current line.
-  — Moves to the left one character. In edit mode, if the SHIFT key is held down at the same time as the right arrow key, the cursor will move to the beginning of the current line.
-  — Moves up one line. In edit mode, if the SHIFT key is held down at the same time as the up arrow key, the cursor will page back an entire screen.
-  — Moves down one line. In edit mode, if the SHIFT key is held down at the same time as the down arrow key, the cursor will page forward an entire screen.
-  — The backspace key. It deletes one character to the left of the cursor.
-  — The right delete key. Deletes the character that the cursor is on.
-  — The line delete key. Deletes everything on a line to the right of the cursor. (The UNDO key will return the deleted line.)
-  — Erases any changes that have been made and returns to command mode.
-  — Defines a procedure and returns to command mode.



-  — Returns the last line that was deleted with the LINE key.
- **CTRL O** — Inserts a blank line.
- **CTRL Q** — Moves the cursor to the left one word.
- **CTRL W** — Moves the cursor to the right one word.

Refer to Chapter 4, Glossary for a complete listing of primitives, function keys, and control keys.

In edit mode of LOGO PROFESSOR, you have the ability to create more than one procedure at a time. this capability is very useful when you have several procedures that interface with one another.

To create multiple procedures simply enter edit mode with the first procedure name, make sure you enter **end** on the last line of the procedure.

```
to square
repeat 4 [fd 100 rt 90]
end
```

Then on the next line type **to** and the next procedure name, the contents of the second procedure, with **end** as the last line.

```
to square
repeat 4 [fd 100 rt 90]
end
to circle
repeat 90 [fd 4 rt 4]
end
```

Keep in mind that you can move freely, between the procedures using the control keys. (See Chapter 4, Glossary, Control Keys.)

When you press the STORE key, all procedures will be defined.

At this point in your education, you should practice creating procedures, experimenting with commands, and familiarizing yourself with the function and control keys. If you need some ideas for procedures, you can try some of the sample procedures in Chapter 3, Sample Procedures

## Variables

Once you have become comfortable with creating and using procedures, you are ready to learn about a new concept, variables. A variable is a name whose value (or contents) can change. For example, if we have a pile of oranges, but the exact number of oranges in the pile can change (i.e., is variable), we can create a variable name (oranges, for example). This variable name will always refer to the number of oranges, although the number can be different at different times. Therefore, at one time in your procedure, oranges may equal 50, whereas later, after you perform calculations, oranges may equal 25.

In LOGO, we create variables and assign values to them with the command **make**. From that point on, every procedure that contains the variable name will use the value assigned to the variable. As shown in the example below, when the value is assigned to oranges with **make**, double quotes precede the variable name. This is how we tell LOGO PROFESSOR that we want to assign a value to a variable. If we later want to access the contents of the variable, we enter a colon before the variable name.

```
> make "oranges 50
> "oranges
Result: oranges
> :oranges
Result: 50
```

Now every procedure that uses the variable oranges, will use the value assigned to it (initially the number 50). This is an example of a global variable.

Let's apply the global variable concept to the procedure that we have created, square. First we will create a variable called steps. We can assign a value to steps right before we run the procedure; we can also change that value at any time.

```
> make "steps 25
```

Now let's modify the procedure square so that it uses the variable steps. You tell LOGO that you are entering a variable name in a procedure by putting a colon (:) in front of it.

```
to square
  repeat 4 [fd :steps rt 90]
end
```

Since the value has already been assigned to the variable `steps`, we can now run the square procedure. Any other procedures that contain the variable name `steps` will also have the value of 25. If you want to change the value, simply execute the **make** primitive again.

```
>make "steps 50
```

Remember, the variable name is preceded by quotes when we are assigning a value to it. When we actually want to use the contents of the variable, we use the colon.

The command **thing** will display the value assigned to a variable. It uses the same format as the **make** command.

```
>thing "steps  
Result: 50
```

Variable names give us flexibility. However, every time that we change the value of a variable, it changes for all procedures using that variable name. We can set up a variable as private instead of global. To do this, we make the variable `steps` a part of the square procedure name when we define the procedure. Then, every time that we run the procedure, we assign the value to the variable. Notice in the example below that the variable name `steps` is a part of the defined procedure name. Now every time that we run square, we must specify a number for `steps`.

```
to square :steps  
repeat 4 [fd :steps rt 90]
```

Now every time that LOGO PROFESSOR sees square, it will expect it to be followed by a number indicating the number of steps. Type in the procedure shown above. Then run the square procedure with the following numbers:

```
square 100
```

```
or
```

```
square 50
```

```
or
```

```
square 75
```

If you enter square without entering the number of steps, LOGO will display an error message:

```
not enough inputs to square
```

This means that LOGO can not create the square unless we specify the number of steps.

So far we have only used variables with numbers. We may also assign words to variables.

```
> make "apples "red  
> :apples  
Result: red
```

Now every time that we use the variable of apple, LOGO will output red. Also note that the word red is preceded by double quotes. We will discuss words and lists in more detail in the next section of this chapter.

This section is just an introduction to variables. You will find as you work with LOGO PROFESSOR that variables names are very helpful and powerful tools. Let's review the commands and concepts associated with variables:

- **make** — Assigns a value to a variable. The format is as follows:  
  

```
make "variable value (numbers or words)
```

If the value being assigned is a word or letter, enter double quotes before the value.
- **thing** — Outputs the value assigned to a variable.
- **Value** — The contents that may be contained within a name such as a variable.
- **Variable** — A name whose value can change.

Refer to Chapter 3, Sample Procedures, for some fun and creative examples of procedures and variables.

## TEXT AND LISTS

Up to this point we have discussed procedures and graphics. In this section, we explain how LOGO creates and manipulates words and lists. We are introducing lists in simple terms in order to increase your understanding of how LOGO PROFESSOR works.

**NOTE:** To make it easier to work with text, you may want to enter **textscreen** (remember the **STYLE** key switches to the different screen modes), so that you can take advantage of the whole screen.

### Displaying Words and Lists

In this section, we demonstrate how primitives treat words and lists in the command mode. Keep in mind that words and lists can be assigned variable names, as explained in the previous section. They can also generate a result that can be used within a procedure. This process is referred to as **output**. The command, **output**, will be discussed later in Section 6, Conditionals, Logic, and Tests, and Section 7, Advanced Techniques.

The first lesson to learn is the difference between words and lists. Words and names (as discussed in the previous section), are preceded by double quotes. Lists are enclosed in square brackets. LOGO PROFESSOR also needs double quotes and square brackets to distinguish between commands and words. For example:

```
> print "hello  
hello
```

If we had entered:

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

LOGO PROFESSOR printed the error message because it read hello as a command or procedure name. We could also have entered:

```
> print [hello]  
hello
```

and it would have been accepted as a list (rather than a **command**). If we want to print more than one word, we must create a list and use the brackets, or put the double quotes in front of each word. For example:

```
> print "hello there  
I don't know how to there
```

“there” is read as a command. But if we entered:

```
> print “hello “there  
hello there
```

or

```
> print [hello there]  
hello there
```

“there” is accepted as a word by LOGO PROFESSOR.

Now that you have seen how brackets and quotes are used with the command **print**, let’s see how they are used with some other primitives. The command **type** is similar to the **print** command when working with lists. For example:

```
> print [hello there]  
hello there  
>
```

```
> type [hello there]  
hello there>
```

The difference between **print** and **type** is that **type** puts the LOGO prompt on the same line as the text, while **print** puts the prompt on the next line.

Now let’s compare the **print** and **type** commands using words:

```
> print “abcd “efgh  
abcd efgh  
>
```

```
> type “abcd “efgh  
abcdefgh>
```

Notice that the **type** command combines the two words into one, and the **print** command keeps the words separate.

The next command we will introduce is the **show** command. Let’s compare it to **print** and **type**:

```
> print “What’s “up “doc  
What’s up doc  
>
```

```
> type “What’s “up “doc  
What’supdoc>
```

```
> show "What's "up "doc
What's up doc
>
```

With words, **show** works exactly the same as **print**. Notice how it works with lists.

```
> print [What's up doc]
What's up doc
>
```

```
> type [What's up doc]
What's up doc>
```

```
> show [What's up doc]
[What's up doc]
>
```

Notice that **show** displays the brackets and also the list. The purpose of **show** is to display lists in their original format. The **show** command will become very useful when you begin creating procedures that contain lists, and you need to make sure that the lists were properly created. This is part of debugging a procedure. (Debugging is simply the process that you go through to ensure that a procedure works properly.) The **print**, **type**, and **show** commands all display the result on the screen. There are summarized below:

- **print** — Used to print a result on the screen. It can be used with text, numbers and commands to print their output.
- **show** — Similar to the **print** command, except that it shows the brackets around lists.
- **type** — Prints a list in the same way as the **print** command. If the **type** command is used with words, it combines all letters together to create a single word. With words and lists, it also leaves the cursor on the same line as the text, instead of the next line (as does the **print** command).

We can now move on to the primitives **sentence**, **word**, and **list**. The command, **word**, will generate a single word out of all words given as inputs, as shown below:

```
> word "tom "cat
Result: tomcat
```

The **sentence** (or **se**) command keeps words separate. When working with lists, the **sentence** command combines words and lists together. See the examples below:

```
> sentence "The "tom "cat "was "making "noise  
Result: [The tom cat was making noise]
```

```
> se "The "tom "cat "was "making [lots of] "noise  
Result: [The tom cat was making lots of noise]
```

The **list** command keeps a list separate from words, as shown below:

```
> list "The "tom "cat "was "making [lots of] "noise  
Result: [The tom cat was making [lots of] noise]
```

Notice the difference between **list** and **se** and the treatment of [lots of]. The list, [lots of], is a sublist within the first list of words. The way that we create a sublist is to surround it by brackets. Sublists are useful when you want to keep one list independent from the rest of the list or list of words.

The **list**, **sentence**, and **word** commands are used to output a result from lists or words. Let's review the commands that have been introduced so far in this section:

- **list** — This command is used to output single or multiple lists (lists and sublists).
- **sentence** (or **se**) — This command will output a group of words or items.
- **word** — This command will output a word. If more than one word or item is given as input, these will be combined into one output.

## Manipulating Words and Lists

The following commands are used to manipulate words and lists: **first**, **last**, **butfirst**, **butlast**, **fput**, **lput**, **count**, and **item**. These commands can be used to find a specific letter in a word or an item in a list. If the input is a word, LOGO PROFESSOR assumes that you wish to extract a character. If the input is a list, then LOGO PROFESSOR assumes that you wish to extract an item from the list. For example:

```
> first "start  
Result: s
```

```
> last [the man jumped [in the air] ]  
Result: [in the air]
```



In the example using the command, **first**, the word, start, was used and only the first character was output. In the example using the command, **last**, the last item in the list (which consisted of 3 words) was output. If we wanted to output everything but the first or last item, we could use the **butfirst (bf)** and **butlast (bl)** commands.

```
> bf "start  
Result: tart
```

```
> bl [the man jumped [in the air] ]  
Result: [the man jumped]
```

Use the **fput** and **lput** commands to put new items in a list.

```
> fput "carefully [the man jumped in the air]  
Result: [carefully the man jumped in the air]
```

```
> lput "carefully [the man jumped in the air]  
Result: [the man jumped in the air carefully]
```

As shown, **fput** and **lput** take two inputs (the second of which must be a list). If the first input is a list, (i.e. [character]) the brackets will print in the output along with the items contained in this first input. Thus these commands can also be used to manipulate lists inside of lists.

```
> fput [the man jumped] [in the air]  
Result: [ [ the man jumped] in the air]
```

```
> lput [the man jumped] [in the air]  
Result: [in the air [the man jumped] ]
```

These commands will also combine lists together.

Two commands that you will find useful when working with lists are **count** and **item**. The easiest way to explain the two commands is through examples:

```
> count [girl dog street car]  
Result: 4
```

As you can see, **count** actually counts the items in the list. Notice the difference when we use **print** with the **count** command.

```
> print count [girl dog street car]  
4
```

In this case the word, Result:, does not print. You will find that using the **print** command with other word, list and math commands has this same effect.

The **item** command selects and outputs a single item from a list. This item may be a sublist. See the examples below:

```
> item 3 [girl dog street car]
Result: street
```

```
> print item 3 [girl dog [busy street] car]
[busy street]
```

Again, note that the use of the **print** command stops Result: from printing.

Let's review the word and list manipulation primitives discussed so far:

- **butfirst (bf)** — Prints (or outputs) all items or letters but the first in the list.
- **butlast (bl)** — Prints (or outputs) all items or letters but the last in the list.
- **count** — Prints (or outputs) the total number of items in a list.
- **first** — Prints (or outputs) only the first character of a word or item of a list.
- **fput** — This command requires two inputs. The second input must be a list. The command will combine the first and second inputs so that the first input becomes the first item in the list.
- **item** — Takes a number and a list as input and prints out the specified item in the list.
- **last** — Prints (or outputs) only the last character of a word or item of a list.
- **lput** — This command requires two inputs. The second input must be a list. The command will take the first input (words or lists) and append it to the second input.

## Readlist, Readchar, and Other Text Commands

You will find it useful when running procedures in LOGO to take input from your terminal keyboard. There are two commands that enable you to do this: **readchar** (or **rc**) and **readlist** (or **rl**). These primitives will output the last entry that is made on the keyboard so that you can use it in subsequent steps. The command **readlist** will output the last entry from the keyboard. The example below displays a very simple use of the **readlist** primitive. This command is used in conjunction with other commands:

```
to question
  print [What is your favorite sport?]
  make "sport readlist
  print [Wow] :sport [sounds like fun!]
end
```

First, the question (What is your favorite sport) prints on the screen. Then, the second statement (make "sport readlist) executes. This statement does two things: first, **readlist** tells LOGO to wait for input from your keyboard; then, the **make** command assigns the last item that you input to be the value of "sport.

We can then use the variable **sport** in the second print statement. When we actually execute this procedure, this is what it looks like:

```
> question
What is your favorite sport?
skiing
Wow skiing sounds like fun!
```

This procedure used the **print** command, the **make** command, and the **readlist** (or **rl**) command. The reason that we use the **make** command is that we can not use **readlist** itself as a variable. Therefore, we transfer the contents of **readlist** to the variable, **sport**.

The command **readchar** (or **rc**) is a bit more advanced than **readlist**. It stores only the first character entered. Notice the effect that **readchar** has on the word **yes**.

```
> readchar
yes
Result: y
> es
```

The word **yes** will not display when you type it in. This simple example displays what the **readchar** primitive does. In most of the procedures that you create initially, you will use the **readlist** command.

To clear the screen of text and move to the top line of the screen, use the command **cleartext**. You can also use the **cleartext** command inside of a procedure to clear the textscreen.

The **text** command allows you to output the contents of a procedure as a list. See the example below:

```
> text "square
Result: [ [ ] [repeat 4 [fd 20 rt 90] ] ]
```

With the **text** command, the entire contents of the specified procedure will print (or output).

Let's review the commands discussed in this section:

- **cleartext** — Clears all text from the screen and puts the LOGO prompt on the first line of the screen.
- **readchar (rc)** — Waits until a key or keys are entered, followed by the RETURN key. Outputs the first letter only.
- **readlist (rl)** — Waits until a line is typed in at the keyboard, followed by the RETURN key. Outputs the line entered as a list.
- **text** — Outputs the contents of a procedure as a list.

## Some Procedures for Manipulating Text

The following procedures give examples of how some of the primitives that have been introduced in this section can be used:

```
> to funlist
make "a [the rain in spain]
make "b [falls mainly in the plain]
print :a :b
make "c ( butfirst butfirst :a )
make "d ( butlast butlast :a )
print :c :d :b
print :c
print :d
print :d :c :b
end
```

The result of this procedure is shown below:

```
> funlist
the rain in spain falls mainly in the plain
in spain the rain falls mainly in the plain
in spain
the rain
the rain in spain falls mainly in the plain
```

Notice that we used the **make** command to assign the value to the variables, and the **butfirst** and **butlast** commands to manipulate the text. Now we will write a procedure which uses variables, and the commands **readlist**, **count**, **cleartext**, and **print**:

```
to questions
print [PLEASE ENTER YOUR NAME:]
make "name readlist
print [THANK - YOU] :name [NOW ENTER YOUR AGE:]
make "age readlist
print [PLEASE TELL ME THE FIRST NAMES OF YOUR BROTHERS
      AND SISTERS]
make "bro readlist
make "num (count :bro)
print [WHAT IS YOUR FAVORITE COLOR?]
make "clr readlist
cleartext
print [HERE IS WHAT I KNOW ABOUT YOU]
print [YOUR NAME IS] :name
print [YOU ARE] :age [YEARS OLD]
print [YOU HAVE] :num [BROTHERS AND SISTERS AND THEIR
      NAMES ARE] :bro
print [YOUR FAVORITE COLOR IS] :clr
end
```

Here is the result:

```
questions
PLEASE ENTER YOUR NAME:
Mary Jones
THANK - YOU Mary Jones NOW ENTER YOUR AGE:
30
PLEASE TELL ME THE FIRST NAMES OF YOUR BROTHERS AND
SISTERS
Jim Sue Sally
WHAT IS YOUR FAVORITE COLOR?
blue

HERE IS WHAT I KNOW ABOUT YOU
YOUR NAME IS Mary Jones
YOU ARE 30 YEARS OLD
YOU HAVE 3 BROTHERS AND SISTERS AND THEIR NAMES ARE
Jim Sue Sally
YOUR FAVORITE COLOR IS blue
```

Notice the use of **count**, and how the number of brothers and sisters was assigned to the variable, **num**.

## Printer Commands

LOGO has commands that assist you when you want to print out information, such as the names of all procedures and variables. The first thing to understand about printing with LOGO PROFESSOR is that, when you want to print something out to the printer, you must first turn the printer on. Then you must press the PRINT key in order to have LOGO PROFESSOR print to the screen and the printer simultaneously. The PRINT key acts as a switch. When you press it the first time, it turns the printer on. When you press it the second time, it turns the printer off.

Note that when you press the PRINT key the first time, the terminal screen will display **pon**. When you press the PRINT key a second time, the screen will display **pooff**. These are LOGO PROFESSOR primitives that also turn the printer on and off. However, we have enabled the PRINT key for your convenience.

To print out the contents of a single procedure, use the **po** primitive. For example:

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

To print out the contents of all procedures, use the **pops** (print out procedures) command. This command will print out the contents of all procedures in the workspace. If a procedure has not been **retrieved** from the disk, then it will not be listed.

The **pots** command prints out all procedure titles, as shown:

```
> pots  
square  
question  
funlist  
questions
```

The **popr** command prints out all primitives. Remember that LOGO's commands are primitives, while any commands that you create are procedures.

To print out all variable names, use the **pons** primitive. To print out all procedures, titles, and variable names, use **poall**.

## CONDITIONALS, LOGIC, AND TESTS

In the previous section we introduced many commands that manipulated text and lists. In this section, we will create procedures applying many of the commands that you have already learned.

### Conditionals

A conditional statement tests a condition to determine what the next action in the program will be. Logic is simply the act of reasoning, a process that we all take part in everyday. Tests check to see if something is present (e.g., a word, a number etc.). In this section, we will discuss each of these concepts and its application to LOGO.

The primary conditional command is **if**. This primitive is very simple and flexible. You simply enter the command in the following format:

```
> if (condition) [action1] [action2]
```

The statement tells LOGO, if the condition is true, perform action 1. Else, if the condition is false, perform action 2. The action 2 entry is optional. If only one action (action 1) is given in an **if** statement, action 1 will be performed if the condition is true. If it is false, LOGO will move on to the next line in the procedure.

Let's fill in the blanks with an example.

```
> if :a > :b [print "hello] [stop]
```

This command is comparing the value of the two variables (:a and :b). If :a is greater than :b the word hello will print; if :a is not greater than :b, LOGO will execute **stop**, exiting the current procedure. Note that **stop** is a LOGO PROFESSOR command that will cause the current procedure to end.

In LOGO PROFESSOR, you can create a procedure and have it execute or call another procedure. This capability is useful when a procedure begins to get too large. Once a procedure is created, you can use it as if it were a command in other procedures. One powerful use of the **if** statement is to enable you to direct the flow to logic from one procedure to another. Review the following procedures to see how this can work.

```
to one
  make "number 5
  print [This is the beginning]
two
  print [this is the end]
end
```

```

to two
  if :number = 0 [stop]
  print [middle]
  make "number :number - 1
  two
end

```

When the first procedure (one) comes to the statement, "two", it will execute the procedure, two. As shown, two will use an **if** statement to determine when it should **stop** and return control to the calling procedure, one. (This will happen when number is equal to zero, which is after two is run five times).

If you wanted to return directly to the LOGO PROFESSOR prompt instead of to the original procedure, you could have used the **toplevel** command in place of **stop**. The primitive **toplevel** is a command that you should use only when you are sure that you don't want to return to the calling procedure. If there is no calling procedure, the **stop** command will return to the LOGO prompt.

## Logic

We will introduce you to the use of logic in LOGO PROFESSOR by discussing three logic primitives: **and**, **or**, and **not**. We will also demonstrate the logic of passing values from one procedure to another.

The **and** primitive requires at least two input statements. It will output true if both (or all) statements are true. Otherwise it will output false. For example:

```

> and (5 - 3 = 2) (2 + 8 = 10)
Result: true

```

```

> and (5 - 3 = 1) (2 + 8 = 10)
Result: false

```

The **not** command is unusual in that it uses reverse logic. **not** takes a statement as input. If the statement is true, the **not** command will output, false. If the statement is false, the **not** command will output, true.

```

> not (1 + 1 = 2)
Result: false

```

```

> not (1 + 1 = 15)
Result: true

```

```

> not (and (1 + 1 = 5) (2 + 2 = 4))
Result: true

```



Like **and**, the **or** command takes at least two inputs, but only one of the inputs must be true for **or** to output true. If all of the inputs are false, it will output false.

```
> or (1 + 1 = 2) (1 + 2 = 4)
Result: true
```

```
> or (1 + 1 = 3) (1 + 2 = 4)
Result: false
```

The examples we use make it easier to illustrate how the primitives work. You can also use **and**, **or**, and **not** with variable names to have them check the values assigned to a variable.

In the previous section of this chapter, Text and Lists, the concept of **output** was discussed. Let's review it for one moment. There will be cases when you want a value passed from one procedure, or variable or primitive to another. This process is referred to as **output**. Whenever you see a primitive print the word, Result:, this means that the **primitive** was ready to output that result to a waiting procedure or variable. However, there was none waiting so the output was printed to the screen.

The primitive, **output**, enables you to set up one procedure to output a value to another. This concept is illustrated by the two procedures shown below, warp and equate.

```
to warp :starunit
print [the star unit count is] :starunit
make 'a :starunit * .5 + 1 / 6
print [Warp speed attainable :] :a
end
```

```
to equate :atomweight
output :atomweight / 3 * 4
end
```

Warp figures your ship's potential warp speed. It needs, as input, the number of starunits available. To figure starunits, you must work through a mathematical equation. Instead of manually figuring and entering starunits, you may use as input another procedure called equate which takes atomweight as input.

Equate will figure your starunits and then outputs the answer to warp. Note that equate has its own numeric input, atomweight. To execute these procedures, type:

```
> warp equate 7
```

Let's read this statement from right to left. The 7 is the value that will be assigned to atomweight when equate executes. Equate will perform the equation shown, and the result value will be assigned to the variable starunit in the procedure warp. Thus, equate passes a value to warp.

## Test

Testing in LOGO is performed by executing one of a group of primitives that check the input to see if some condition is present. These primitives, each of which ends with a p (for present), are defined in this section.

- **equalp** — Checks to see if two (or more) values are equal. If they are, **equalp** outputs true. Otherwise, it outputs false.

```
> equalp 2 2  
Result: true
```

Remember that you can use **equalp** with variables as well.

- **keyp** — This primitive checks to see if there is a keyboard entry being made. If there is, **keyp** outputs true. Otherwise, it outputs false. This primitive is useful when you have a procedure containing an option of "press any key to continue."

- **namep** — Checks to see if the input is a currently defined variable name.

```
> namep "a  
Result: false  
> make "a "hello  
> namep "a  
Result: true
```

- **listp** — Checks to see if the input is a list. If it is, **listp** outputs true. Otherwise, it outputs false.

```
> listp "hello "there  
Result: false  
> listp [hello there]  
Result: true
```

Remember that LOGO PROFESSOR looks for the square brackets around lists.

- **numberp** — Checks to see if the input is a number. If it is, **numberp** outputs true. Otherwise it outputs false.

```
> make "a 50
> numberp :a
Result: true
> make "a "fifty
> numberp :a
Result: false
```

- **wordp** — Checks the input to see if it is a word. If the input is a word, **wordp** outputs true. Otherwise, it outputs false.

```
> wordp [hello]
Result: false
> wordp "hello
Result: true
```

- **definedp** — Checks workspace to see if the input is a defined procedure. If it is, **definedp** outputs true. Otherwise, it outputs false.

```
> pots
square
> definedp "square
Result: true
```

Remember, **definedp** checks the procedures in workspace. If a procedure has been created but has not been **retrieved**, **definedp** will respond false.

- **empty** — Checks to see if the input is empty. This test is helpful when you are working with lists. If the input is empty, **empty** will output true. Otherwise it will output false.

```
> empty [a]
Result: false
> empty butfirst [a]
Result: true
```

- **shownp** — Checks to see if the turtle is displayed. If it is, **shownp** will output true. If it is not, **shownp** will output false.

- **primitivep** — Outputs true if the input is a primitive. Otherwise it outputs false.

```
> primitivep "square  
Result: false  
> primitivep "forward  
Result: true
```

## DISK FILES AND MEMORY MANAGEMENT

In the first section of this chapter, we introduced you to saving and retrieving files from workspace. In this section, we will define these and other concepts in more detail in order to explain how you manage your files on disk and your memory storage.

### Saving Workspace and Procedures

When you begin to create procedures, you should identify those procedures that you want to save and those that you do not need. Before storing your workspace on disk, erase any unneeded procedures. This step should become an automatic housekeeping function that you perform regularly in LOGO. It doesn't do you any good to save unneeded procedures in workspace or on disk.

To erase a single procedure from workspace, use the primitive **erps** followed by the specific procedure name. Be sure to specify a procedure name after you enter **erps**. If you do not, you will erase all procedures from your workspace. See the example below:

```
> erps "square  
> erps
```




Once again, the first example erases a single procedure, while the second erases all procedures in workspace.

To erase any unneeded variable names from workspace, use the primitive **erns**. It works just as **erps** does. That is, if you specify a name, only that name will be erased: if you specify no names, **erns** will erase all names in workspace.

```
> make "a 50  
> pons
```

```
a is 50  
> erns "a
```

When you are ready to save your workspace on disk, simply press the STORE key on the keyboard (or key in the **store** primitive). LOGO will wait for you to enter the filename that the workspace should be stored under. This filename should be no more than 8 characters long and it should contain no periods. Make the name given to workspace something meaningful, so that you will be able to recognize it when you are ready to retrieve it. Also note that you should not assign the same filename as a previous file unless you want to write over the previously saved file.

-  — This function prints a directory of all LOGO data files on the data disk. (You may also enter the primitive, **index**.)
-  — Enables the **retrieve** primitive. LOGO PROFESSOR will wait for the entry of the filename of the file to be retrieved from the disk.
-  — In command mode, LOGO PROFESSOR will wait for you to enter a filename to have workspace stored under. In edit mode, the STORE key will define the procedure currently being worked on.

## Workspace

Throughout this manual, we have referred to workspace as your current working area when you are using LOGO PROFESSOR. Let's now consider what workspace is, and how it effects you.

When you are performing any functions in LOGO, even if they are temporary, they still take up workspace. When you create something temporary and then erase it, that used workspace is set aside until LOGO needs space. The process of reclaiming used space is called garbage collection. LOGO **recycles** the garbage workspace and returns it to available, contiguous workspace. You may enter the **recycle** primitive yourself, but there is no real need to do this because LOGO performs this function automatically when it needs the additional space.

In LOGO PROFESSOR, workspace is broken down into five-byte increments. These five-byte increments are called **nodes**. There are approximately 2000 nodes available for use. Keep this in mind, because it indicates that there is a limit to the number and size of procedures that you can have in workspace at any one time. If you type in **nodes**, LOGO PROFESSOR will display the number of free nodes available in workspace.

As mentioned, there are 2000 available nodes in LOGO PROFESSOR. Part of workspace is set aside for internal functions of LOGO PROFESSOR. If for some reason LOGO is so short on workspace that it must reclaim the portion of workspace assigned to internal functions, it will perform a function called **expand**; **expand** is also a primitive that you may enter. It is strongly recommended that you not use this function unless you are very experienced with LOGO PROFESSOR.

The primitive **mem** enables you to identify the amount of workspace that is available and the amount that is in garbage collection.

## ADVANCED PROCEDURES AND CONCEPTS

Now that you are the LOGO PROFESSOR expert, you can move on to the advanced LOGO concepts. These include: recursion, debugging procedures, whole numbers, advanced mathematics, random numbers, advanced graphics, and machine language interface.

### Recursion

In Section 6, Conditionals, Logic, and Tests, you were introduced to the concept of creating a procedure that calls another procedure. Now let's consider a concept called **recursion**, which is the process of a single procedure calling itself.

The term recursion comes from the word recur which means to occur again or in intervals. In relation to LOGO, a procedure which calls itself and thus occurs repeatedly is a recursive procedure. You can have a recursive procedure that is endless, which forces you to press the STOP key twice to exit. The example below illustrates this:

```
> to sketch
  make "command readchar
  if :command = "r [right 5]
  if :command = "l [left 5]
  if :command = "f [forward 10]
  if :command = "b [back 10]
  if :command = "c [clearscreen]
  sketch
end
```

In this procedure, the variable, **command**, is assigned the value of **readchar**. After we execute the procedure, every time the appropriate key is pressed, the turtle moves accordingly. After the last **if** statement, **sketch** calls itself and thus repeats.

You could also have entered a line to allow you to **stop** when you press a specific key, for example:

```
> to sketch
  make "command readchar
  if :command = "r [right 5]
  if :command = "l [left 5]
  if :command = "f [forward 10]
  if :command = "b [back 10]
  if :command = "c [clearscreen]
  if :command = "s [stop]
  sketch
end
```

## ADVANCED PROCEDURES AND CONCEPTS

Now that you are the LOGO PROFESSOR expert, you can move on to the advanced LOGO concepts. These include: recursion, debugging procedures, whole numbers, advanced mathematics, random numbers, advanced graphics, and machine language interface.

### Recursion

In Section 6, Conditionals, Logic, and Tests, you were introduced to the concept of creating a procedure that calls another procedure. Now let's consider a concept called **recursion**, which is the process of a single procedure calling itself.

The term recursion comes from the word recur which means to occur again or in intervals. In relation to LOGO, a procedure which calls itself and thus occurs repeatedly is a recursive procedure. You can have a recursive procedure that is endless, which forces you to press the STOP key twice to exit. The example below illustrates this:

```
> to sketch
  make "command readchar
  if :command = "r [right 5]
  if :command = "l [left 5]
  if :command = "f [forward 10]
  if :command = "b [back 10]
  if :command = "c [clearscreen]
  sketch
end
```

In this procedure, the variable, **command**, is assigned the value of **readchar**. After we execute the procedure, every time the appropriate key is pressed, the turtle moves accordingly. After the last **if** statement, **sketch** calls itself and thus repeats.

You could also have entered a line to allow you to **stop** when you press a specific key, for example:

```
> to sketch
  make "command readchar
  if :command = "r [right 5]
  if :command = "l [left 5]
  if :command = "f [forward 10]
  if :command = "b [back 10]
  if :command = "c [clearscreen]
  if :command = "s [stop]
  sketch
end
```



Recursion can be a very powerful programming function. The next example, the tree procedure, is a recursive procedure that calls itself in two separate statements (notice the fourth and sixth lines each call tree). This procedure begins with the variables of dist and num, which represent distance and number of branches.

```
> to tree :dist :num
  if :num = 0 [stop]
  fd :dist
  lt 25
  tree :dist / 1.1 :num - 1
  rt 50
  tree :dist / 1.1 :num - 1
  lt 25
  bk :dist
end
```

Each time LOGO encounters a line with tree, it causes a recursion. In this procedure, the fourth and sixth lines initiate recursion. The complete processing accomplished by this procedure is quite lengthy and dependent on the variables entered. It would be easier to understand if we could see the exact steps that LOGO is performing during this procedure. In Appendix B, An example of Recursion, you will find a step-by-step outline of the tree procedure.

## Advanced Logic

This section expands on several commands that are described elsewhere in this manual. These include: **define**, **text**, and **run**.

The **define** primitive is used to create a procedure from a series of lists without requiring you to go into edit mode. This makes it possible to create a procedure within a procedure. The format of this command is:

```
define “<procedure name > [ [variables used] [line 1] [line 2] ]
```

You may vary the number of lines that your procedure contains simply by adding additional sublists.

**text** is the reverse of **define**. It takes as input a procedure name and outputs a list in the same format:

```
[ [variables] [line 1] [line 2] [etc] ]
```

The **run** command allows you to enter a string of commands without creating a procedure. **Run** may be used to execute a list containing workable commands. For example:

```
run [repeat 4[fd 100 rt 90] ]  
  <will draw a square>
```

## Debugging Procedures

The outline that is in Appendix B, An Example of Recursion, was printed by initiating a function in LOGO called **trace**. The primitive **trace** allows us to have LOGO PROFESSOR display each step as it is being performed.

There are three functions which will allow us to **trace** a procedure as it runs. These three functions are:

- **trace 1** — Displays the titles of any user-defined procedures as they are called or stopped.
- **trace 2** — Displays primitives as they are called or stopped.
- **trace 3** — Displays both primitives and titles of user-defined procedures as they are called or stopped.

If you want to get a printout of **trace** as it executes, remember to turn the printer on by pressing the PRINT key (or typing **pon**). When **trace** is finished, you must remember to turn the trace function off with the **notrace** command. The example in Appendix B, uses the **trace 3** function.

There may be times when you need to use a character that is special to LOGO PROFESSOR (e.g. [ ] \* /). In order to tell LOGO that the special meaning of the character should be ignored, you may use the backslash (\) before the special character. When LOGO encounters the backslash, it will ignore the following characters special meaning.

## Whole Numbers

LOGO PROFESSOR has two primitives which take decimal numbers and make whole numbers out of them. These primitives use the concept of integers and rounding (an integer is a whole number). The primitive **int** will take a decimal number and truncate it to make it into a whole number. For example:

```
>int 10.3  
Result: 10
```

```
>int 10.9  
Result: 10
```

As you can see, the primitive **int** simply removes any numbers to the right of the decimal point. The second primitive that changes decimal numbers to whole numbers is **round**. The difference between **int** and **round** is that while **int** truncates any decimals, **round** rounds numbers off to the nearest whole number. Thus if the decimal is 5 or above, it will **round** up to the next number; if the decimal is 4 or below, it will **round** down. The following examples illustrate **round**:

```
> round 10.3  
Result:10
```

```
> round 10.9  
Result: 11
```

```
> round 10.5  
Result: 11
```

## Advanced Mathematics

This section describes the advanced mathematical functions that are available through LOGO. The mathematical concepts themselves are not explained in this document (for example, we presume that you understand the meaning of square root, cosine, etc.). But the applicaiton of these concepts to LOGO is described here.

In section 2 of this chapter, Turtle Graphics the basic math commands were introduced. Let's look at some additional math commands that you will find useful. These commands include **sum**, **quotient**, **remainder**, **product**, **sqrt**, **sin**, **cos**.

The primitive **sum** allows you to enter a string of numbers to be added without putting the plus sign (+) between the numbers.

```
> sum 1 2 5  
Result: 8
```

The primitive **quotient** takes two numbers as input, makes them integers and divides the second number into the first. If there is a remainder, it is discarded, as **quotient** only outputs whole numbers.

```
> quotient 10 5  
Result: 2
```

```
> quotient 10.5 5.2  
Result: 2
```

The primitive **remainder** takes two numbers as input, makes them integers and divides the second number into the first. The quotient is discarded and the result is the remainder.

```
> remainder 10 4  
Result: 2
```

```
> remainder 10 5  
Result: 0
```

```
> remainder 10.5 5.2  
Result: 0
```

The primitive **product** takes a string of numbers as input and multiplies them together without requiring the \* symbol between the numbers.

```
> product 5 10  
Result: 50
```

The primitive **sqrt** returns the square root of a number.

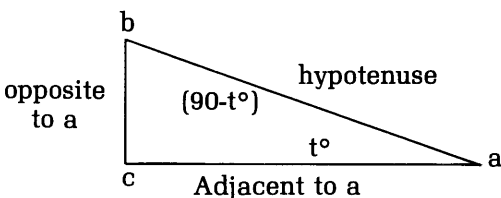
```
> sqrt 225  
Result: 15
```

```
> sqrt 300  
Result: 17.32051
```

LOGO PROFESSOR has the ability to generate the trigonometric sine and cosine. The definitions of the sine and cosine of an angle are given in terms of ratios between lengths of the sides of a right angle. If an angle (a) of a right triangle has a measure of t degrees, the trigonometric definitions of the sine and cosine of t degrees (abbreviated as sin t degrees and cos t degrees, respectively) are:

$$\sin t \text{ degrees} = \frac{\text{length of side opposite (a)}}{\text{length of hypotenuse}}$$

$$\cos t \text{ degrees} = \frac{\text{length of side adjacent to (a)}}{\text{length of hypotenuse}}$$



The primitive **sin** requires an input in degrees, and will output its trigonometric sine.

The primitive **cos** requires an input in degrees, and will output its trigonometric cosine.

```
> sin 30  
Result: .5
```

```
> cos 30  
Result .8660253
```

The **sin** and **cos** primitives can also be used to create the secant and cosecant of an angle.

```
> to secant :angle  
  output 1 / cos :angle  
end
```

```
> to cosecant :angle  
  output 1 / sin :angle  
end
```

You can also use the **sin** and **cos** primitives to create the tangent and cotangent of an angle.

```
> to tan :angle  
  output sin :angle / cos :angle  
end
```

```
> to cotangent :angle  
  output cos :angle / sin :angle  
end
```

## Random Numbers

LOGO PROFESSOR has the ability to generate random numbers through the **random** primitive. You simply enter the high number. For example, if we entered **random** with the number 10, LOGO would generate a number from 0 to 9. The random number generator allows you to create interesting effects with programs. An example of **random** is given below:

```
> to randomnum  
  repeat 20 [print random 100]  
end
```

This procedure will simply output a series of random numbers. If you refer to Chapter 3, Sample Procedures, you will find a procedure called Laser, which uses random in conjunction with graphics.

## Advanced Graphics

So far in this document, our only method of moving the turtle around has been by telling it how many steps to take. There are advanced graphics functions which allow us to move the turtle to a specific point on the screen by entering x and y coordinates. There are also primitives which will display the position of the turtle. If you are unfamiliar with the terms x and y coordinates, refer to Figure 2-1. This is the LOGO PROFESSOR screen with the x,y axis marked. Notice that the point where the two lines meet is the home position (or 0 0 coordinates). You should identify the point on the screen where you want to position the turtle and analyze the points where the x and y table meets. For example, the dot on the table is in the 100 100 x y coordinates. Please note that the numbers on the bottom and left side of the screen use negative numbers.

The primitive **pos** outputs a list containing two values, the first being the current x coordinate of the turtle and the second being the current y coordinate of the turtle. If we enter **pos** when the turtle is in the home position we will get the following result.

```
> pos
Result: [0 0]
```

The **xcor** and **ycor** simply outputs the x and y coordinates separately.

```
> xcor
Result: 0
```

```
> ycor
Result: 0
```

The primitive **heading** outputs the direction in which the turtle is pointing at that time. For example:

```
> heading
Result: 0
```

If we then tell the turtle to turn **right** 90 degrees, the heading will change:

```
> right 90
> heading
Result: 90
```

We then turn right 90 again to change the heading:

```
>right 90
>heading
Result: 180
```

And again:

```
>right 90
>heading
Result: 270
```

If we turn right 90 one more time, we are back at zero (which is also 360).

The primitive **setpos** will set the position of the turtle on the screen based on the x,y coordinates. If the pen is down, a line will be drawn to that spot. The **setpos** primitive requires two inputs, the first being the x coordinate and the second being the y coordinate. They must be entered surrounded by square brackets. For example:

```
>setpos [100 - 100]
```

**setpos** can also accept input from **list** or **sentence**. See the example below:

```
>setpos sentence random 320 random 199
```

or

```
>setpos list random 320 random 199
```

The most common use of **setpos** is with direct input. However, the **list** and **sentence** primitives are also compatible, as shown.

The primitive **dot** followed by the x,y coordinates, will make a dot appear on the screen at the specified point.

```
>dot 100 100
```

Here is a sample procedure that uses the **dot** and **random** primitives.

```
> to specs
  dot (random 400) - 200 (random 400) - 200
  specs
end
```

This procedure will continue to print dots on the screen until you press the STOP key twice.

You can also set the x and y coordinates separately with the **setx** and **sety** primitives.

```
> setx 100  
> sety 100
```

The **setheading** primitive allows you to tell the turtle what direction to point in. In the examples shown above with the **heading** command, we were moving the turtle with the **right** primitive. With **setheading**, you can tell the turtle to setheading 90 and the turtle will point to the right.

```
> setheading 90
```

The **pen** primitive outputs a list which contains the current status of the pen. The four possibilities are **penup**, **pendown**, **penerase** and **penreverse**. There is also a primitive which will allow you to set the pen status. This command is **setpen**, in most cases, you will simply enter the proper command to set the pen.

The primitive **setscrunch** allows you to change the axis of the screen. You must enter a number to tell LOGO how much you want to change the axis by. The standard axis setting is 1. If you change it to a number greater than 1, it will stretch the vertical axis. If you change it to a number less than 1, it will compress the vertical axis. If you change it to a negative number, the vertical axis will be inverted. When the axis is inverted, the normal coordinate positions are reversed. Instead of the positive x,y coordinates being on the top of the screen, they will be on the bottom.

The primitive **scrunch** displays the current setting of **setscrunch**. The standard screen setting is 1.

## Character/ASCII conversion

There are two primitives which will allow you to convert a character into its corresponding ASCII (American Standard for Computer Information Interchange table) value and vice versa. The primitive **char** requires input from 0 to 255. If the number input is between 0 and 255, **char** will output the character corresponding to that number in the Epson Expanded Character table.

```
> char 110  
Result: n
```



The **ascii** primitive outputs the corresponding character for the character entered. The character must be preceded by double quotes.

```
> ascii "n  
Result: 110
```

## Machine Language Primitives

LOGO PROFESSOR has the ability to interface with programs written in machine code. Machine Language refers to programming languages such as Assembly, C and Forth. Knowledge of machine language is necessary to utilize the commands described in this section. These primitives allow direct access to memory locations and machine language subroutines. The command **.deposit** requires two decimal (versus hexadecimal) inputs, the first being the memory location, the second being the value to deposit. The **.deposit** primitive is similar to the poke command in the BASIC programming language.

The command **.call** requires one decimal (versus hexadecimal) input, and runs the subroutine whose starting address is at that location in memory. When the subroutine is finished, LOGO will execute the next line in the calling procedure.

The command **.examine** requires one decimal (versus hexadecimal) input, and returns the decimal value stored in that memory location. The **.examine** primitive is similar to the peek command in BASIC.

The **in** and **out** primitives are used to control hardware devices. The command **in** requires a decimal (versus hexadecimal) address of an input port as input and returns the decimal value currently present at that input port.

The command **out** requires two decimal (versus hexadecimal) inputs, the first being an output port address, the second being the value to be sent to the device at that output address.

## Exiting LOGO PROFESSOR

When you are finished with LOGO PROFESSOR and are ready to exit, you should make sure that you have saved any procedures that you do not want to lose. You can then simply press the MENU key on the keyboard, and LOGO will display a message:

Remove your disks and press the RESET button to exit.

Make sure you keep both diskettes in their protective jackets.

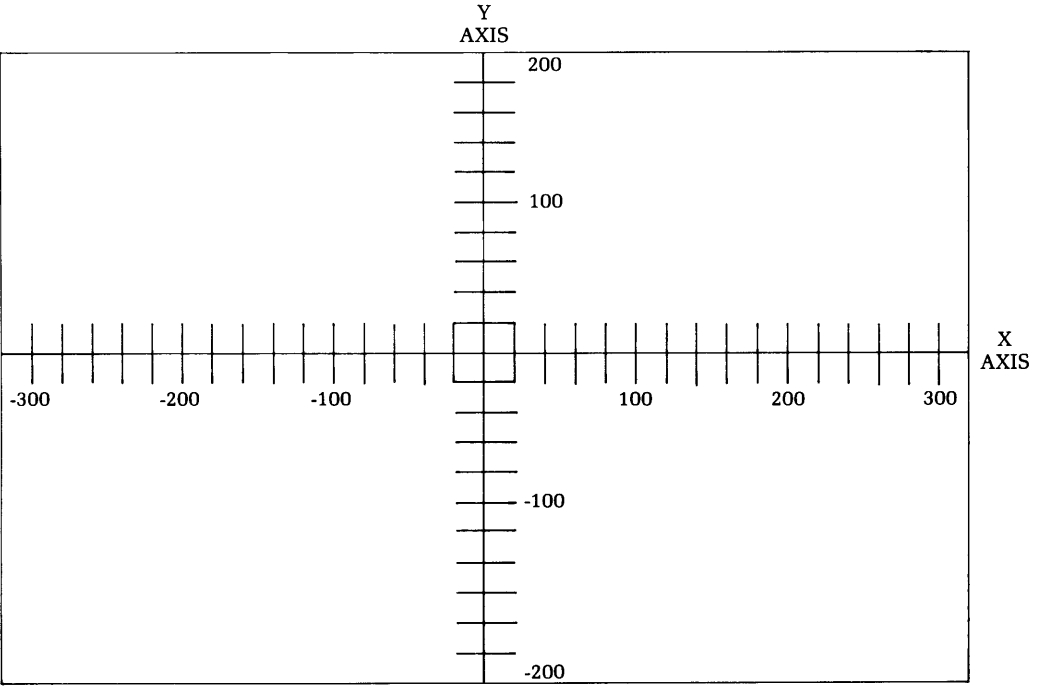


FIGURE 2-1

The following section simply categorizes all of the commands that have been introduced in this chapter.

## Section 2

### Turtle Movement Primitives:

forward	right
back	left
home	

### Pen Control Primitives:

penup	pendown
penreverse	penerase

### Turtle Display Primitives:

hideturtle	showturtle
------------	------------

### Clearing the Graphics Screen:

clearscreen	clean
-------------	-------

### Simple Logic with Graphics:

repeat

### Screen Mode Primitives:

fullscreen	splitscreen
textscreen	STYLE key

### Screen Type Primitives:

fence	wrap
window	SIZE key

### Mathematic and Logic Primitives:

-	+
/	*
=	<
>	{
}	

### Section 3

Primitives that create and Define Procedures:

to	edit
store	

Primitives that manipulate Variables:

make	thing
------	-------

Primitives that Print:

pon	poff
popr	pots
pops	pons
poall	

### Section 4

Primitives that Print and Output Text and Lists:

list	count
word	sentence
print	type
show	test

Primitives that Manipulate Text and Lists:

first	last
butfirst	butlast
lput	fput

Primitives that Remember Keyboard Entries:

readchar	readlist
----------	----------

Clearing the Text Screen:

cleartext

### Section 5

Conditional and Logic Commands:

repeat	if
stop	oplevel
output	and, or, not

Present tests

equalp	keyp
namep	listp
numberp	wordp
definedp	emptyp
shownp	primitivep

Section 6

Function Keys:

STORE	PRINT
RETRIEVE	

Primitives that Erase:

erasefile  
erps  
erns

Workspace Management:

recycle  
expand  
mem

Section 7

Advanced Math Commands:

sin	cos
sum	remainder
product	quotient
sqrt	int
round	random

Screen Manipulation Commands:

setscrunch    scrunch

Advanced Turtle Graphics Primitives:

setx	sety
setpos	setheading

**Primitives that Display the Turtles Position:**

<code>xcor</code>	<code>ycor</code>
<code>pos</code>	<code>heading</code>

**Pen Control Primitives:**

<code>pen</code>	<code>setpen</code>
------------------	---------------------

**Character and ASCII Conversion:**

<code>ascii</code>	<code>char</code>
--------------------	-------------------

**Debugging Procedure Primitives:**

<code>trace</code>	<code>notrace</code>
--------------------	----------------------

**Advanced Logic Primitives:**

<code>define</code>	<code>run</code>
---------------------	------------------

**Machine Language Primitives:**

<code>in</code>	<code>out</code>
<code>.deposit</code>	<code>.call</code>
<code>.examine</code>	

## CHAPTER 3

### SAMPLE PROCEDURES

In this chapter several sample procedures are outlined. These procedures support the concepts described in Chapter 2, Using LOGO PROFESSOR, and on the help screens. In working through these procedures, if you should come across a command you do not understand, review it in Chapter 2 or refer to Chapter 4, Glossary.

#### A SQUARE

This square uses only two commands: **forward** and **right**.

```
to square
forward 100
right 90
forward 100
right 90
forward 100
right 90
forward 100
end
```

#### A SIMPLE SQUARE

The square procedure is simplified by using the **repeat** command, which will simply repeat the “forward 100 right 90” sequence four times.

```
to rsquare
repeat 4 [forward 100 right 90]
end
```

#### A VARIABLE SQUARE

By adding the variable “:size,” the square procedure can be made more versatile. The procedure will now create a square with sides of variable length.

```
to vsquare :size
repeat 4 [forward :size right 90]
end
```

The images produced by square and rsquare will be identical, a simple square with sides 100 units long. Vsquare is also a simple square, but the sides can literally be any length that you wish. When you run vsquare, remember to follow it with the size that you desire (e.g., vsquare 100).

## A ROTATING SQUARE

The rotating square procedure is like the variable square procedure, but with an enhancement that will cause it to create an interesting design. It draws a square, rotates the turtle 10 degrees, and then draws another square. Rotating square will repeat this action 36 times.

```
to xsquare :size
  repeat 36 [repeat 4[forward :size rt 90]rt 10]
end
```

## A TRIANGLE

Vtriangle creates a triangle in the same way that vsquare creates a square. However, because a triangle has three sides rather than four, it is only necessary to repeat the forward/turn sequence three times. Also, the right turn increased to 120 degrees (360 divided by 3) as opposed to 90 degrees for the square (360 divided by 4).

```
to vtriangle :size
  repeat 3 [forward :size right 120]
end
```

## A POLYGON

Polygon is a procedure that takes two inputs—one for the “size” of the design, and one for the “angle” of the design’s turns and corners. By varying the inputs the user can create a vast array of designs. This procedure will require that STOP be pressed twice to halt execution, because the way it is written, the procedure will never end.

```
to poly : side :angle
  forward :side
  right :angle
  poly :side :angle
end
```

try these inputs for poly: poly 80 144  
poly 60 80

## A CIRCLE

In LOGO PROFESSOR, a circle is comprised of a large number of very small straight lines arranged in a circle. These straight lines are so minute that they are indistinguishable as straight lines.



```

to circle :size
repeat 180 [forward :size right 2]
end

```

The equation “**forward :size right 2**” is repeated 180 times. Each time it is run the line is shifted two degrees (**right 2**). 180 times 2 equals 360 degrees, thus forming a full circle. A faster, but less accurate circle could be drawn with the equation **repeat 90 [forward :size right 4]**.

## A DESIGN

This procedure will design a “cell” which will then be used in other procedures (createl and create2) to create more complex designs. This procedure is as follows:

```

to design
repeat 2[forward 60 right 90]
repeat 2[forward 30 right 90]
forward 60
right 90
repeat 2[forward 15 right 90]
forward 30
end

```

When stored and executed, this procedure will draw the figure shown below:



create1, shown below, executes design twice every time that it runs, and it will need to run four times to complete its creation. Because create1 calls itself at the end of the procedure, it sets up an endless loop and will run until the STOP key is pressed twice.

```

to create
design
design
left 90
create
end

```

The “creation” will look like this:



Create2 produces a different “creation” using the design procedure. This procedure also calls itself, and again it will be necessary to press the STOP key twice to stop it.

```
to create2
  design
  left 45
  forward 70
  create2
end
```

## SPIRAL

In this procedure you may dramatically alter your output design by using different numbers for the input.

```
to spiral :angle
  ht
  make "side 0
  repeat 100 [forward :side right :angle make "side :side + 2]
end
```

Remember, to execute spiral you will need to type spiral along with a number. Here are some suggestions:

```
spiral 89
spiral 72
spiral 118
```

## A POLYGON SPIRAL

The polygons procedure creates a polygon which spirals outward endlessly. It requires three inputs: angle, side, and increment of expansion. You must press the STOP key twice while this procedure is running to halt execution.

```
to polygons :side :angle :inc
  polydraw :side :angle
  polygons ( :side + :inc ) :angle :inc
end
```

```
to polydraw :side :angle
  forward :side
  right :angle
end
```

Here are some suggestions for “polygons”:

```
polygons 5 120 3
polygons 5 144 3
polygons 1 45 1
```

## PRONTO

Pronto allows you to manipulate the turtle with a single keystroke. In this example, **forward** will be equal to F, **right** will be equal to R, **left** will be equal to L, and **clear-screen** will be equal to C. To abbreviate additional commands, just add “if :ord” lines to the “order” procedure.

```
to pronto
order readchar
pronto
end
```

```
to order :ord
make “ord readchar
if :ord = “f [forward 10]
if :ord = “r [right 45]
if :ord = “l [left 45]
if :ord = “c [clearscreen]
end
```

## LASER

This procedure creates an action-filled design. Two procedures are required to execute this program, as given below. After defining the procedures, just type laser to execute them.

```
to laser
home
window
penreverse
hideturtle
lasermt
end
```

```
to lasermt
setx (random 600 * - 1)
setx (random 200)
sety (random 300) * - 1)
sety (random 200)
lasermt
end
```

## TEST

The following set of procedures are called test. This program was designed to drill the user with addition. Slight modifications can be made to create subtraction or multiplication drills.

This set of procedures utilizes both numeric and text logic functions.

```
to test
  make "num1 random 100
  make "num2 random 100
  make "answer :num1 + :num2
  print (sentence [HOW MUCH IS] :num1 [ + ] :num2)
  make "reply readnumber
  if :reply = :answer [print [good job
sport!]] [print sentence [no, the answer is]
:answer]
test
end

to readnumber
output first readlist
end
```

By typing in the numeric answers to test's questions, you trigger a right answer or wrong answer display which is automatically followed by another question.

## LACE

This procedure involves fairly complex mathematical manipulations. The procedure itself is short, but recursion (the act of a procedure calling itself) will produce results that are astounding.

```
to lace :size :limit
if :size < :limit [forward :size stop]
lace :size / 3 :limit
left 90
lace :size / 3 :limit
right 90
lace :size / 3 :limit
right 90
lace :size / 3 :limit
left 90
lace :size / 3 :limit
end
```

Any numbers may be used for the inputs, but here are some you should try:

```
lace 243 9
lace 243 3
```

## AN ARRAY

This set of procedures illustrates the use of an array (a stored list). File is the controlling procedure. It allows the creation of a file called array, which has 20 mailboxes. Upon completion of the 20 inputs, file will print the entire list. The procedure "setval" controls the "build" procedure. Together they create the file. "Display" controls the "find" procedure. Together they pull the values from the file and display them.

```
to build :name :num :val
  make ( word :name :num ) :val
end
```

```
to find :name :num
  output thing word :name :num
end
```

```
to display :value
  if :value = 0 [stop]
  print find "array :value
  make "value :value - 1
  display :value
end
```

```
to setval :n
  if :n = 0 [stop]
  type (make element #) :n [:]
  build "array :n readlist
  setval :n - 1
end
```

```
to file
  setval 20
  display 20
end
```

These sample programs and procedures are just a start, but they may suggest to you the awesome potential of LOGO PROFESSOR. We hope that we have planted within you the seeds of ideas to pursue and questions to investigate in this versatile language.








# CHAPTER 4






## GLOSSARY

This chapter contains definitions for all LOGO PROFESSOR function keys, primitives, control keys, and general terms. This chapter is intended primarily as a reference aid. To learn how and when to use the LOGO PROFESSOR keys and commands, refer to the on-screen help screens. See also Chapter 2, Using LOGO PROFESSOR, and Chapter 3, Sample Procedures. Then refer back to the appropriate section in this chapter when you need a reminder about the usage of a specific key or primitive.

### FUNCTION KEYS

This section lists and describes the function keys that have been enabled for use on the QX-10 with LOGO PROFESSOR.



-  — Prints graphics designs to the printer.
-  — Calls up the on-screen help screens. When you press this key, all information in workspace will be temporarily saved, and you will be presented with a choice between the help screen index (to locate a specific command function) and full help (to work through a specified subset of commands). On exiting from the help screens, you will be returned to workspace, where you may resume use of LOGO.
-  — Displays a directory of all files on the LOGO data disk. (See **index** in primitives section of this chapter.)
-  — Deletes all characters on the line to the right of the cursor. If pressed while the cursor is at the end of the line, it will cause the current line to merge with the line below it. The UNDO key may be used to recreate what has been deleted by the LINE key. (See UNDO key.)
-  — Used to exit LOGO. The system responds: “remove your disks and press reset to exit LOGO (or push any key to continue). When you exit LOGO, all procedures in workspace will be destroyed. If you would like to save the procedures in workspace, use the STORE function key. (See STORE key.)

-  — Turns the printer on and off. (See **pon** and **poff**.)
-  — Used to retrieve files from the data disk and load them into the workspace. The retrieve key will display “retrieve. (See **retrieve** in the glossary of primitives.)
-  — Press this key to switch from **window** to **fence** to **wrap**. (See these primitives in the glossary for more information.)
-  — Use the STOP key to pause or to stop execution, as described below:
  - Pressed once in command mode, the STOP key will cause execution to pause. Pressing any other key will cause execution to resume.
  - Pressed twice in command mode, the STOP key will stop execution and present the LOGO prompt.
  - If pressed in edit mode, STOP will exit from edit, ignoring any changes made.
-  — The STORE key has two functions:
  - In edit mode, the STORE key will exit the editor and process the edited text.
  - In command mode, the STORE key will display the prompt:

store”

To respond to this prompt, enter a filename to save a copy of the workspace on disk. This filename should be 8 characters or less and contain no periods. You may later retrieve this file from disk at any time by using the RETRIEVE key. (See **store** in the glossary of primitives.)









-  — Switches from **fullscreen** to **splitscreen** to **textscreen**. (See these primitives in the glossary.)
-  — At the current cursor position, inserts any text that was previously deleted by the LINE key. (See LINE key.)

## CONTROL KEYS AND ARROWS

This section contains a list of the control characters that function with the LOGO PROFESSOR procedure editor. A control function is executed by depressing the control key (CTRL) and the designated character key simultaneously.

- **CTRL A** — Moves the cursor to the beginning of the current line.
- **CTRL B** — Moves the cursor one character to the left (same as LEFT ARROW).
- **CTRL D** — Deletes the character at the current cursor position.
- **CTRL E** — Moves the cursor to the end of the current line.
- **CTRL F** — Moves the cursor one character to the right (same as RIGHT ARROW).
- **CTRL H** — Deletes the character to the left of the cursor.
- **CTRL J** — Same as RETURN.
- **CTRL N** — Moves the cursor down one line (same as DOWN ARROW).
- **CTRL O** — Opens a new line at the cursor position.
- **CTRL P** — Moves the cursor up one line (same as UP ARROW).
- **CTRL Q** — Moves the cursor to the end of the previous line.
- **CTRL V** — Pages the cursor down one screen (24 lines).
- **CTRL W** — Moves the cursor to the right one word.

-  — Moves the cursor one position to the right.
-  — Moves the cursor one position to the left.
-  — Moves the cursor to the position immediately above its present position.
-  — Moves the cursor to the position immediately below its present position.
- **THE SHIFT KEY** — Augments the effect of the arrow keys. If used with UP or DOWN ARROW, SHIFT ARROW will advance or retreat one full page. If used with RIGHT or LEFT ARROWS, SHIFT ARROW will set the cursor at the beginning or the end of the line.
-  — The backspace key deletes one character to the left of the cursor.
-  — In edit mode, deletes the character the cursor is on.

## PRIMITIVES

This section lists alphabetically all LOGO PROFESSOR primitives, along with their definitions. Simple abbreviations are also given for those primitives for which they are available, in order to help you reduce keying time.

- \* Multiply. Takes two numbers as input, multiplies them, and outputs their product. For example:

```
5 * 2
Result: 10
print 5 * 2
10
```

- = Equals. Compares two input values. If both inputs are numbers, compares them to see if they are equal. If both inputs are words, compares them to see if they are identical character strings. If both inputs are lists, compares them to see if their corresponding elements are equal. Outputs true or false accordingly. For example:

```
print 20 = word "2 "0
true
print "a = [a]
false
print [A B] = sentence "A "B
true
```

- < Less than. Compares two inputs. Outputs true if its first input is less than its second, false otherwise. (See **if**.) For example:

```
2 < 5
Result: true

make "a 4
make "b 7
:a < :b
Result: true
```

> Greater than. Compares two inputs. Outputs true if the first input is greater than the second, false otherwise. (see **if**.) For example:

```
12 > 45  
Result: false
```

```
make "x 10  
make "y 7  
if :x > :y [stop]
```

- Minus. With two numeric inputs, gives their difference. With one numeric input, gives it a negative value, if there is no space between the minus sign and its input. For example:

```
print 5 - 2  
3  
print 1 + (- 2)  
- 1
```

+ Plus. Adds two input numbers, and outputs their sum. For example:

```
print 5 + 2.5  
7.5
```

/ Divide. Outputs the first number divided by the second. Always outputs a decimal value. For example:

```
print 5 / 2  
2.5  
print 6 / 2  
3
```

**and** Takes a variable number of input statements (at least two) and evaluates whether each statement is true. Outputs true if all statements are true, otherwise outputs false. (Compare to **or**.)

```
print (and (1 + 1 = 2) (5 = 4) (1 = 1) )  
false
```

**ascii** Takes a character as input and outputs the number that is the ASCII code of that character. (Compare to **char**.)

```
ascii "R
82
```

**back** Abbreviated **bk**. Takes one number as input and moves the turtle backward. Draws a line when the pen is down. (See **forward**.) For example:

```
back 100
< Turtle moves backward
100 units >
```

**butfirst** Abbreviated **bf**. Prints all elements in a series but the first. If input is a list, outputs a list containing all but the first item. If input is a word, outputs a word containing all but the first character. Gives an error when called with empty word or empty list as input. (Compare to **butlast**.) For example:

```
print butfirst [THIS IS A LIST]
IS A LIST
```

```
print butfirst "ABRACADABRA
BRACADABRA
```

**butlast** Abbreviated **bl**. Prints all elements in a series but the last. If input is a list, outputs a list containing all but the last element. If input is a word, outputs a word containing all but the last character. Gives an error when called with empty word or empty list as input. (Compare to **butfirst**.) For example:

```
print butlast [this is a list]
this is a
print butlast "ABRACADABRA
ABRACADABR
```

**char** Takes an integer as input and outputs the character whose ASCII code is that integer. (Compare to **ascii**.) For example:

```
char 67
Result: C
```

**clean** Takes no inputs. Clears the graphics screen. Does not change the turtle's position. (Compare to **home** and **clearscreen**.)

**clearscreen** Abbreviated **cs**. Takes no inputs. Clears the graphics screen and centers the turtle on the screen. (Compare to **clean** and **home**.)

**cleartext** Takes no inputs. Clears the text screen and moves the cursor to the left edge of the screen on the first available text line.

**cos** Outputs the cosine of its input as an angle in degrees. For example:

```
cos 59.99999
Result: .5
cos 45
Result: .7071066
```

**count** Takes a list as input and outputs the number of items in the list. If the input is a word, outputs the number of letters. For example:

```
count [FEE FIE]
Result: 2
print count [FEE FIE FOE FUM]
4
```

**define** Defines a procedure. This primitive takes two inputs. The first input is the name that the procedure will be given. The second input is a list that consists of a series of "sub-lists." The first sub-list contains the inputs for the procedure. Each following sublist contains a line of the procedure. For example:

```
define "ptsum [ [x y] [print :x] [print :x + :y] ]
```

defines the procedure:

```
to ptsum :x :y
  print :x
  print :x + :y
end
```

Note that you normally use **to** rather than **define** in order to define procedures. The **define** primitive is useful for writing procedures that define other procedures.

**definedp**

Returns a true statement if the argument is a user-defined primitive. For example:

```
to foo
fd 10
rt 45
end
```

```
definedp "foo
Result: true
```

**.deposit**

An advanced function. Stores byte values at absolute RAM addresses. (Refer to Chapter 2, Using LOGO PROFESSOR Section 7, Advanced Procedures and Concepts.) For example:

```
.deposit 128 201
<byte 201 is stored at address 128>
```

**dot**

Takes as input two coordinates specifying a screen position and places a dot at that position. Does not move the turtle. For example:

```
dot 20 100
<a dot appears on the screen at position
(20,100)>
```

**edit**

Abbreviated **ed**. Takes as input a procedure name, and allows you to edit that procedure. (See **to**.) For example:

```
edit "square
<sets up procedure square for
editing>
```

**empty** Takes one input. Outputs true if the input is the empty word or the empty list, false otherwise.

```
empty [LOLLIPOP]
```

```
Result: false
```

```
print empty butfirst "X
```

```
Result: true
```

**end** Terminates a procedure definition that is typed into the editor. It is not necessary to type **end** after the final definition unless you are defining more than one procedure at a time. In that case, the procedure definitions must be separated by end statements.

**equalp** Same as =. For example:

```
print equalp "A [A]
```

```
false
```

**erase** Abbreviated **er**. Erases the designated procedure from workspace. Can also take as input a list of procedures to be erased. The procedure name or names must be preceded by a quotation mark ("). For example:

```
erase "square
```

```
< deletes the procedure SQUARE >
```

**erasefile** Takes a file name as input and removes that file from the disk. (Compare to **erps** and **erase**.)

```
erasefile "MYFILE
```

```
< deletes the file whose name  
is myfile >
```

**erns** With no input, erases all variable names from workspace. **erns** does not erase procedures.

**erps** With no input, erases all procedures from workspace. **erps** does not erase variable names.

**.examine** An advanced function. Outputs the value of the byte at the specified address. (Refer to Chapter 2, Using LOGO PROFESSOR, Section 7, Advanced Procedures and Concepts.)



**fence** Takes no inputs. Causes LOGO PROFESSOR to display the message “turtle out of bounds” in response to any attempt to move the turtle past the screen boundaries. The SIZE key can be used to switch to any of the three screen borders. (Compare **wrap** and **window**.)

**first** If input is a list, outputs the first element. If input is a word, outputs the first character. Signals an error when called with the empty word or the empty list as input. (Compare to **last**.)

```
print first [THIS IS A LIST]
THIS
print first “ABRACADABRA
A
```

**forward** Abbreviated **fd**. Takes one number as input, and moves the turtle forward for the number of units specified. Draws a line if the pen is down. (Compare to **back**.)

```
forward 100
< turtle moves forward 100 units >
```

**fput** Takes two inputs, the second of which must be a list. Outputs a list consisting of the first input followed by the elements of the second input. (Compare to **lput**.) For example:

```
print fput [A B] [C D]
[A B] C D
print fput “dark [in the cellar]
dark in the cellar
```

**fullscreen** In command mode, gives full graphics screen. Complementary to **splitscreen** and **textscreen**. No text will be visible. The STYLE key can be used to switch to any of the three screen modes.

**heading** Outputs the turtle’s heading as a decimal number between 0 and 360. (The heading is the turtle’s orientation on the screen, where 0 is straight up and 180 is straight down.) For example:

**heading**  
Result: 90

**setheading heading + 10**  
<rotates the turtle 10 degrees  
clockwise>

**hideturtle**      Abbreviated **ht**. Makes the turtle invisible. (See **showturtle**.)

**home**            Moves the turtle to the center of the screen, pointing straight up (heading is 0).

**if**                Enables you to set up conditional statements. LOGO's basic condition form is:

**if < condition > [ < action1 > ] [ < action2 > ].**

The < condition > is tested. If it is true, < action1 > is performed. If it is false, < action2 > is performed. The < action2 > entry is optional. For example:

```
> make "x 4
> make "y 7
if :x = 5 [stop] [print "hello]
hello

if :x < :y [print "yes] [print "no]
yes
if 10 < y + x [print "yes] [print
"no]
yes
```

**in**                An advanced function. Returns the value from the specified Z80 I/O port. Refer to Chapter 2, Using LOGO PROFESSOR, Section 7, Advanced Procedures and Concepts.

**index**            Displays a directory of all LOGO files on the currently mounted data disk. In drive "B", the INDEX key may be used to display an index.

**int**              Takes one numeric input and converts it to a whole number by truncating any fractional part. For example:

```
print int 5.6
5
print int - 5.6
- 5
```

**item** Takes a number *n* and a list (or word) as input, and outputs the *n*th item in the list (or letter in a word). Signals an error if *n* is greater than the **count** of the list. For example:

```
print item 3 [FEE FIE FOE FUM]
FOE
```

**keyp** Outputs true if a keyboard character is pending (i.e., the character input buffer is not empty), otherwise outputs false.

**last** If input is a list, outputs the last element. If input is a word, outputs the last character. Signals an error when called with the empty word or the empty list as input. (Compare to **first**.) For example:

```
print last [THIS IS A LIST]
LIST
```

```
print last "ABRACADABRAX
X
```

**left** Abbreviated **lt**. Takes one numeric input, and rotates the turtle that number of degrees counterclockwise. (Compare to **right**.)

```
left 90
< turtle rotates 90 degrees
counterclockwise >
```

**list** Takes a variable number of inputs and outputs a list of the inputs. It will separate the lists from the rest of the output with bracket symbols.

```
print list "A "B
A B
print list "A "B [1 2 3] "C
A B [1 2 3] C
```

**listp** Outputs true if its input is a list. Outputs false otherwise.

**lput** Takes two inputs, the second of which must be a list. Outputs a list consisting of the elements of the second input followed by the first input. (Compare to **fput**.) For example:

```
print lput "Z [ W X Y]
W X Y Z
print lput [A B] [C D]
C D [A B]
```

**make** Takes two inputs, the first of which must be a word (which is any character string). Assigns the second input to be the value associated with the first input. For example:

```
make "YELLOW 50
print :YELLOW
50
```

**mem** Displays the following information concerning memory: Nodes, Free nodes, Segment, and Garbage collections. (See **nodes**). For example:

```
> mem
Nodes:          1000
Free nodes:     535
Segments:       1
Garbage collections: 0
```

Node Refers to total available nodes. Free nodes refers to nodes not in use. The difference being, nodes that may be used by LOGO and procedures. Segments reflects the number of times expand has been performed. Garbage collections reflects the number of time recycle has been performed.

**memberp** Takes an item and a list as input. Outputs true if the item is a member of the list, false otherwise.

```
print memberp "A [ONCE UPON A TIME]
true
```

**menu** Takes no inputs. Exits LOGO. Will display the statement: "Remove your disks and press reset to exit (or press any key to continue)."

**namep** Outputs true if its input is a variable.

- nodes** Outputs the number of currently free nodes. This is a measure of how much storage is available in the workspace. LOGO PROFESSOR automatically expands its workspace as it fills up, so do not be alarmed if the node reading is low. A node is equal to 5 bytes.
- not** Equivalent to “does not equal.” Outputs true if its input is false, false if its input is true.
- ```
if not (1 = 2) [print "HELP]
help
```
- notrace** Turns off the trace function. (See **trace**.)
- numberp** Outputs true if the input is a number, false otherwise.
- or** Takes variable number of inputs (at least two) and outputs true if at least one is true, otherwise outputs false. (Compare to **and**.) For example:
- ```
print (or (1 + 1 = 3) (5 = 4) )
false
print (or (1 + 1 = 2) (5 = 4)
(1 = 1) )
true
```
- out** Advanced function. Outputs <VALUE> for the specified Z80 port. (See Chapter 2, Using LOGO PROFESSOR and Techniques, Section 7, Advanced Procedures and Concepts.)
- ```
out <PORT-NUMBER> <VALUE>
```
- output** Abbreviated **op**. Takes one input. Causes the current procedure to stop and output the result to the calling procedure.
- pen** Outputs the state of the pen (**penup**, **pendown**, or **penreverse**). For example:
- ```
pen
Result: [pendown]
```
- pendown** Abbreviated **pd**. Causes the turtle to leave a trail when it moves. (Compare to **penup**, **penerase**, and **penreverse**)

- penerase**      Abbreviated **pe**. Puts the turtle's eraser down, so that, when the turtle moves, any point that it passes over will be erased. (Compare to **pendown**, **penerase** and **penreverse**.)
- penreverse**    Abbreviated **px**. Changes the turtle's pen so that it "reverses" any point that it passes over. (Compare to **pendown**, **penup** and **penerase**.)
- penup**          Abbreviated **pu**. Takes no inputs. Causes the turtle to move without leaving a trail. (Compare to **pendown**, **penerase** and **penreverse**.)
- po**              If given a procedure name as input, prints out the text of the procedure. Can also take a list of procedure names as input, in which case it prints out the text of all procedures in the list. For example:
- po** [SQUARE XSQUARE]  
<Prints the text of "SQUARE"  
and "XSQUARE">
- poall**          With no input, prints all procedure definitions and the values of all variables.
- pons**            Prints the values of all variables.
- popr**            Prints all LOGO primitives.
- pops**            With no input, prints the definitions of all procedures.
- pos**             Outputs a list of two numbers that specify the turtle's current position in x,y coordinates. (Compare to **xcor**, **ycor**, and **setpos**.)
- pots**            With no input, prints the titles of all procedures.
- primitivep**    Outputs true if the input is a primitive, false if it is not.
- print**          Abbreviated **pr**. Takes a variable number of inputs (one is required). Prints them on the screen, separated by spaces, and moves cursor to the next line. When **print** prints lists, the outermost pair of brackets is not

printed. See example 3 below:

```
print "hi
hi
print "hello "out "there
hello out there
print [hello out there]
hello out there
```

Note that this primitive is used in conjunction with many other commands as shown in the examples.

**product**

Multiplies. Takes a variable number of inputs (at least two) and outputs their product.

```
print product 3 4 5
60
```

**quotient**

Divides. Takes two numeric inputs, divides the first by the second, and outputs the quotient, rounded off as a whole number. If the inputs are not integers, it truncates any decimal values. For example:

```
print quotient 5 2
2
```

**random**

Takes one numeric input, a positive whole number  $n$ , and outputs a whole number between 0 and  $n - 1$ . For example:

```
print random 100
53
print random 100
27
```

**readchar**

Abbreviated **rc**. Outputs the first character entered in the character buffer, or if empty, waits for an input character.

**readlist**

Abbreviated **rl**. Waits for an input line to be typed, terminated with RETURN. Outputs the line as a list.

**recycle**

Reclaims unused storage space in the workspace.

**remainder** Takes two numbers as inputs. Divides the first input by the second and outputs the remainder. If the inputs are not integers, it first truncates any decimal values. For example:

```
remainder 13.6 10.8  
Result: 3
```

**repeat** Takes a number and a list as input. Executes the list the specified number of times. For example:

```
repeat 3 [print "hello]  
hello  
hello  
hello
```

**retrieve** Takes as input a filename. Retrieves the given file from the data disk and places it in the workspace. The filename should be 8 characters or less. Destroys any graphics display. This command may be entered manually or with the RETRIEVE key. (See RETRIEVE key.) For example:

```
retrieve "myfile  
<loads "myfile" into the  
workspace>
```

**right** Abbreviated **rt**. Takes a number as input, and rotates the turtle that number of degrees clockwise. Please note that you can also specify a negative number. (Compare to **left**.) For example:

```
right 45  
< turtle rotates 45 degrees  
clockwise >
```

**round** Takes a whole number as input, and outputs the rounded-off integer. For example:

```
print round 5.6  
6  
print round 5.4  
5  
round 5.5  
Result: 6
```



**run** Takes a list as input. Executes the list as if it were a typed-in command line.

```
run [print [good morning] ]
good morning
run list "print [good morning]
make "command" print
make "input [good morning]
run list :command :input
good morning
```

**scrunch** Takes no inputs. Displays the current setting of the graphics screen aspect ratio. (See **setscrunch**.)

**sentence** Abbreviated **se**. Takes a variable number of inputs. If inputs are all lists, outputs their elements as a single list. If any inputs are words, they are regarded as one-word lists in performing this operation. (Compare to **word**). For example:

```
print sentence "hello "there
hello there
print sentence [this is] [a list]
this is a list
print sentence "this [is] [a list]
this is a list
print sentence[ [HERE IS] A] [NESTED LIST]
[HERE IS] A NESTED LIST
```

**setheading** Abbreviated **seth**. Takes one numeric input. Turns the turtle to face the specified heading. Zero is straight up, with heading increasing clockwise. (See **heading**.)

```
setheading 180
< turtle now faces straight down >
```

**setpen** Sets the turtle pen according to the input. For example:

```
setpen [penerase]
< sets pen to penerase >
```

**setpos** Takes a list of two numbers as input and moves the turtle to the specified point. Draws a line if the pen is down. For example:

```
setpos [100 50]
< turtle moves to the coordinate
position (100,50)>
```

**setscrunch** Changes the vertical scale at which LOGO graphics are drawn. Takes one numeric input and uses this to change the scale factor. The default value for the factor is 1. This command may be used to create interesting graphics.

**setx** Takes one numeric input and moves the turtle horizontally to the specified x coordinate. Draws a line if the pen is down. (Compare to **setpos** and **sety**.)

**sety** Takes one numeric input and moves the turtle vertically to the specified y coordinate. Draws a line if the pen is down. (Compare to **setpos** and **setx**.)

**show** Similar to **print**, except that it prints the outer brackets around lists. Show is often useful in debugging procedures that deal with lists. Examples are:

```
print [hi]
hi
show [hi]
[hi]
print "hi
hi
show "hi
hi
```

**shownp** Outputs true if the turtle is currently shown, false otherwise.

**showturtle** Abbreviated **st**. Makes the turtle appear. (Compare to **hideturtle**.)

**sin** Outputs the sine of its input (as an angle in degrees).

**splitscreen** In command mode, gives mixed text/graphics screen. Complementary to **fullscreen** and **textscreen**. The **STYLE** key may be used to switch to any of the three screen modes.

**sqrt** Takes a positive number as input and outputs the square root of that number.

**stop** Causes the current procedure to stop and return control to the calling procedure.

**store** In command mode, accepts a filename of up to eight characters as input. The filename should have no periods. Saves the contents of the workspace on disk. Destroys any graphics display. **store** may be entered manually or by use of the STORE key. You may also specify a single procedure name to be saved. In edit mode, the procedure being edited will be defined in workspace. (See STORE Key.) For example:

```
store "myfile
store "onefile "proc1
```

**sum** Adds. Takes a variable number of inputs (at least two) and outputs their sums. For example:

```
print sum 1 2 3 4
10
```

**text** Takes a procedure name as input and outputs that procedure text as a list in the same format as described for input under define. For example, procedure ptsum is defined as follows:

```
to ptsum :x :y
  print :x
  print :x + :y
end
ptsum may be printed using text as
follows:
text "ptsum
Result: [x y][print :x][print :x + :y]
```

**textscreen** Hides the graphics screen so that the entire screen can be used for text. Complementary to **splitscreen** and **fullscreen**. The STYLE key may be used to switch to any of the three screen modes.

**thing** Outputs the assigned value of its input (which must be a word). **thing** "xxx is normally abbreviated as :xxx. (See **make**). For example:

```
make "orange 50
thing "orange
Result: 50
```

**to** Begins procedure definition. Enters definition mode if typed as a direct command. (Compare **edit**.)

**toplevel** When executed in a procedure, **toplevel** returns the user to the LOGO prompt.

**towards** Takes a list of two numbers as input. These are interpreted as the x and y coordinates of a point on the screen. Outputs the heading from the turtle to the point. **setheading** may then be used to turn the turtle to that heading. For example:

```
setheading towards [100 50]
< turtle is now facing point
(100,50)>
```

**trace** Switches on the trace function. **trace** has three levels, indicated by a numeric input:

**trace 1** Displays the titles of the user-defined procedures as they are called or stopped.

**trace 2** Displays all LOGO primitives as they are executed.

**trace 3** Displays both the user-defined procedures and the LOGO primitives as they are executed or called.

**trace** is invaluable for debugging procedures, because it enables you to observe the flow of execution of procedures and primitives, so that you can track problems. In addition to the functions just described, **trace** also tracks and displays the levels of sub-procedures. Such as those encountered in recursive procedures.

- type** Takes a variable number of inputs, like **print**, but does not move the cursor to the next line after printing. With multiple words, does not print spaces between the words. With lists, it leaves the spaces in place. For example:
- ```

type "to "get "her
together >
type [to get her]
to get her >

```
- window** Takes no inputs. Makes the graphics screen act like a window onto a much larger turtle field. Thus the turtle can move past the edge of the screen. Only the portion of the turtle's trail that lies in the "window" will appear on the screen. The **SIZE** key switches to any of the three screen border commands. (Compare to **wrap** and **fence**.)
- word** Takes a variable number of inputs. Outputs a single word that is the concatenation of the inputs (which must be words). For example:
- ```

print word "mish "mash
mishmash
print word "123 "45 "678
12345678

```
- wrap** Takes no inputs. Causes subsequent turtle commands to "wraparound" in response to attempts to move the turtle past the screen borders. That is to say, a turtle path that goes off the top of the screen will continue from the bottom of the screen at the same horizontal position. A path that goes off the right edge of the screen will continue from the left edge of the screen at the same vertical position.
- xcor** Outputs the turtle's x coordinate as a decimal number. For example:
- ```

setx xcor + 10
<moves the turtle 10 units to the right>

```
- ycor** Outputs the turtle's y coordinate as a decimal number.



## GENERAL TERMS

This section defines terms that you may encounter when using LOGO PROFESSOR. Some of these terms are specific to LOGO, but most are general terms that are found in computer science or math. The definitions presume no background in these fields.

|                               |                                                                                                                                                                                                                                      |
|-------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Boot</b>                   | The process of initiating the operating system of a computer. It can be performed manually or automatically.                                                                                                                         |
| <b>Bug</b>                    | A program error. (See debugging).                                                                                                                                                                                                    |
| <b>Bracket</b>                | A symbol, “[”, used to designate lists in LOGO.                                                                                                                                                                                      |
| <b>Calling Procedure</b>      | A procedure which has, within itself, a command to execute another procedure.                                                                                                                                                        |
| <b>Command</b>                | A term used to create action. LOGO Primitives are commands.                                                                                                                                                                          |
| <b>Command mode</b>           | The interactive state of LOGO PROFESSOR. Command is the only mode that can execute procedures. (Compare to edit mode.)                                                                                                               |
| <b>Conditional Expression</b> | A question whose answer decides the course of action of a procedure. For example, a certain conditional expression may execute a <b>print</b> command if the answer to the question is false, and <b>stop</b> if the answer is true. |
| <b>Control Characters</b>     | Refers to using the CTRL key in conjunction with another key on the keyboard (e.g., CTRL A means, hold the CTRL key down at the same time as the A key).                                                                             |
| <b>Cosine</b>                 | An advanced mathematical function.                                                                                                                                                                                                   |
| <b>CPU</b>                    | Central Processing Unit. The main unit of the computer system. In contains disk drives and memory. It is used in conjunction with the monitor/keyboard and printer.                                                                  |
| <b>Contiguous</b>             | Bordering. In relation to computer memory, contiguous refers to memory that is efficiently organized, with no wasted space.                                                                                                          |
| <b>Cursor</b>                 | A flashing square on the computer terminal. The cursor tells you where your keyboard entry will be displayed.                                                                                                                        |

|                         |                                                                                                                                                                                                                                                                                                                                 |
|-------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Horizontal</b>       | Straight across. Parallel with the horizon.                                                                                                                                                                                                                                                                                     |
| <b>Inputs</b>           | Any information fed into the computer. This includes anything entered with the keyboard.                                                                                                                                                                                                                                        |
| <b>Integer</b>          | A whole number or zero. Cannot contain any value to the right of the decimal point.                                                                                                                                                                                                                                             |
| <b>List</b>             | A list of words, numbers and/or special characters. In LOGO, lists are enclosed in brackets “[ ]”.                                                                                                                                                                                                                              |
| <b>Media</b>            | When used in reference to computers, this term describes the method by which data is transferred or stored. For example, most computer programs are distributed on magnetic media.                                                                                                                                              |
| <b>Memory</b>           | The memory of a computer system is a circuit board inside of the computer system which functions as an intermediate storage area. Programs are held in memory when they are in use by the monitor/keyboard.                                                                                                                     |
| <b>Monitor</b>          | CRT or terminal. The monitor is the display screen of the computer system.                                                                                                                                                                                                                                                      |
| <b>Name</b>             | A title assigned to a variable in LOGO. The variable may a word, list, or numeric value.                                                                                                                                                                                                                                        |
| <b>Node</b>             | A unit of measure for memory space in LOGO PROFESSOR. A node is equal to 5 bytes.                                                                                                                                                                                                                                               |
| <b>Operating System</b> | Programs which control the fundamental operation of computer system software. Sometimes referred to as the “brains” of a computer system.                                                                                                                                                                                       |
| <b>Output</b>           | This term may mean one of the following: <ul style="list-style-type: none"> <li>• Any information exiting the computer. The most common sources are the CRT and the printer.</li> <li>• Any information exiting a LOGO PROFESSOR procedure.</li> <li>• A LOGO primitive. (See the primitive section of the chapter.)</li> </ul> |



|                        |                                                                                                                                           |
|------------------------|-------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Primitive</b>       | A command in LOGO PROFESSOR vocabulary. (See the primitive section of this chapter.)                                                      |
| <b>Printer</b>         | The hardware device that is used to print information from the computer.                                                                  |
| <b>Procedure</b>       | A user-defined command. A procedure may contain turtle graphics commands, programming commands or a combination of both.                  |
| <b>Processor</b>       | See CPU.                                                                                                                                  |
| <b>Program</b>         | A series of instructions that tell the computer how to perform certain tasks.                                                             |
| <b>Prompt</b>          | A computer symbol soliciting an input. The standard LOGO prompt is ">".                                                                   |
| <b>Quotient</b>        | The solution for a division equation.                                                                                                     |
| <b>Recursion</b>       | A procedure which calls itself.                                                                                                           |
| <b>Screen mode</b>     | Three modes are available: <b>fullscreen</b> , <b>textscreen</b> , and <b>splitscreen</b> . (See these in the primitives glossary.)       |
| <b>Software</b>        | A group of programs that perform a similar function.                                                                                      |
| <b>Sum</b>             | The solution of an addition equation.                                                                                                     |
| <b>Terminate</b>       | To end.                                                                                                                                   |
| <b>Toggle</b>          | A switch that switches back and forth between two modes with each push of a single button.                                                |
| <b>Truncate</b>        | To delete a portion of something. For example, if the fractional value of 10.234 were truncated, only the number 10 would remain.         |
| <b>Turtle Graphics</b> | The method used by LOGO PROFESSOR to create impressive drawings on the computer. The user instructs the LOGO turtle how to draw graphics. |
| <b>Value</b>           | The strength or worth of an object. The value of a variable is the number or word that it represents at given point in time.              |

|                  |                                                                                                                                                  |
|------------------|--------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Variable</b>  | A word created in LOGO PROFESSOR whose value is changeable.                                                                                      |
| <b>Vertical</b>  | Straight up and down.                                                                                                                            |
| <b>Word</b>      | A group of characters with no imbedded spaces. In LOGO, words are characterized by preceding quotation marks. ('').                              |
| <b>Workspace</b> | LOGO PROFESSOR'S working area is referred to as workspace. All user defined procedures are held in workspace until they are written to the disk. |

## **BIBLIOGRAPHY AND REFERENCES**

- Abelson, H. Apple LOGO. Byte/McGraw-Hill 1982.
- Coxeter, H.S.M., Regular Polytopes. Dover Publications, 1973.
- Huntley, H.E., The Divine Proportion: A Study in Mathematical Beauty. Dover Publications, 1970.
- Kim S., Inversions. Byte Books/McGraw-Hill, 1981.
- Nye J.F., Physical Properties of Crystals. Oxford University Press, 1967.
- Papert, S., Mindstorms: Children, Computers, and Powerful Ideas. Basic Books, 1980.
- Pearce, P., Structure in Nature Is a Strategy for Design. MIT Press, 1978.
- Phillips, F., An Introduction to Crystallography. Wiley, 1963.
- Stevens, P., Handbook of Regular Patterns: An Introduction to Symmetry in Two Dimensions, MIT Press, 1981.
- Thornburg, D.D., Picture This! An Introduction to Computer Graphics for Kids of All Ages. Addison-Wesley 1982.
- Thornburg, D.D., Picture This Too! An Introduction to Computer Grahpics for Kids of All Ages. Addison-Wesley 1982.
- Thornburg, D.D., Discovering Apple LOGO. Addison Wesley 1983.

## APPENDIX A

### HELPFUL HINTS

This appendix provides a series of tips and suggestions for working with LOGO PROFESSOR. Refer to it as needed as you work with LOGO.

- Check the amount of workspace available after working in LOGO PROFESSOR for long periods of time, or when creating large procedures.
- When working in LOGO PROFESSOR for long periods of time, periodically save your workspace to disk in case of any unforeseen problems that could cause loss of data, such as power failure.
- If a single procedure begins to get large, break it down into smaller, more manageable procedures for ease of programming and debugging.
- Fullscreen mode is used for displaying graphics, but it prohibits the display of text on the screen. When you are in fullscreen mode, if you want to see what LOGO PROFESSOR may be printing, simply turn the printer on (press the PRINT Key or enter **pon**). In this way, you will not miss any text messages or displays, because these will print on the printer.
- Do not use the expand function unless you are very familiar with LOGO workspace. Remember that LOGO PROFESSOR will automatically perform the recycle and expand when it needs more workspace.
- Please note that using **penreverse** in **setscrunch** mode will leave irregular dot patterns (see Chapter 4, Glossary, for definitions of these primitives).
- Remember that workspace is cleared when you exit LOGO PROFESSOR. Be sure to save your procedure (use the STORE key). To load them back into workspace when you need them use the RETRIEVE key.
- Make sure you enter all commands or primitives in lower case. LOGO PROFESSOR will not be able to find a primitive if you request it in upper case.
- Don't leave any unneeded procedures in workspace, as this can cause LOGO PROFESSOR to slow down. To ensure the best execution speed, load into workspace only those procedures that are required.

- If you load (or retrieve) into workspace a file of procedures that was previously saved, and you then erase some of those procedures from workspace, when you save (or STORE) your workspace file back to disk, be sure to give it a new file name. If you use the same file name, save (or STORE) will write over the old file and thus permanently delete those files that were erased from working storage.
- If you specify a procedure name or variable name in upper case, all references to that name must be in upper case.

## APPENDIX B

### AN EXAMPLE OF RECURSION

The following is a **trace** of the “tree” procedure presented in Chapter two, Section 7. **trace 3** was used so it lists all commands and procedures executed. The indentations represent the levels of recursion. At its deepest point, three levels of recursion occurred (i.e., tree called tree called tree). Refer back to the Advanced Procedures and Concepts section in Chapter two for a review of recursion. Then, for a vivid demonstration of the principle of recursion, try running the procedure tree, on your own system.

```
> tree 30 3
tree of 30 3
: if of false [stop]
: if stops
: fd of 30
: fd stops
: lt of 25
: lt stops
: tree of 27.27272 2
: : if of false [stop]
: : if stops
: : fd of 27.27272
: : fd stops
: : lt of 25
: : lt stops
: : tree of 24.79338 1
: : : if of false [stop]
: : : if stops
: : : fd of 24.79338
: : : fd stops
: : : lt of 25
: : : lt stops
: : : tree of 22.53944 0
: : : : if of true [stop]
: : : : stop called
: : : : stop stops
: : : : if stops
: : : tree stops
: : : rt of 50
: : : rt stops
: : : tree of 22.53944 0
: : : : if of true [stop]
: : : : stop called
: : : : stop stops
```

: : : if stops  
: : : tree stops  
: : : lt of 25  
: : : lt stops  
: : : bk of 24.79338  
: : : bk stops  
: : tree stops  
: : rt of 50  
: : rt stops  
: : tree of 24.79338 1  
: : : if of false [stop]  
: : : if stops  
: : : fd of 24.79338  
: : : fd stops  
: : : lt of 25  
: : : lt stops  
: : : tree of 22.53944 0  
: : : : if of true [stop]  
: : : : : stop called  
: : : : : stop stops  
: : : : if stops  
: : : tree stops  
: : : rt of 50  
: : : rt stops  
: : : tree of 22.53944 0  
: : : : if of true [stop]  
: : : : : stop called  
: : : : : stop stops  
: : : : if stops  
: : : tree stops  
: : : lt of 25  
: : : lt stops  
: : : bk of 24.79338  
: : : bk stops  
: : tree stops  
: : lt of 25  
: : lt stops  
: : bk of 27.27272  
: : bk stops  
: tree stops  
: rt of 50  
: rt stops  
: tree of 27.27272 2  
: : if of false [stop]  
: : if stops  
: : fd of 27.27272

```

: : fd stops
: : lt of 25
: : lt stops
: : tree of 24.79338 1
: : : if of false [stop]
: : : if stops
: : : fd of 24.79338
: : : fd stops
: : : lt of 25
: : : lt stops
: : : tree of 22.53944 0
: : : : if of true [stop]
: : : : : stop called
: : : : : stop stops
: : : : if stops
: : : tree stops
: : : rt of 50
: : : rt stops
: : : tree of 22.53944 0
: : : : if of true [stop]
: : : : : stop called
: : : : : stop stops
: : : : if stops
: : : tree stops
: : : lt of 25
: : : lt stops
: : : bk of 24.79338
: : : bk stops
: : tree stops
: : rt of 50
: : rt stops
: : tree of 24.79338 1
: : : if of false [stop]
: : : if stops
: : : fd of 24.79338
: : : fd stops
: : : lt of 25
: : : lt stops
: : : tree of 22.53944 0
: : : : if of true [stop]
: : : : : stop called
: : : : : stop stops
: : : : if stops
: : : tree stops
: : : rt of 50
: : : rt stops

```



```
: : : tree of 22.53944 0
: : : : if of true [stop]
: : : : : stop called
: : : : : stop stops
: : : : : if stops
: : : tree stops
: : : lt of 25
: : : lt stops
: : : bk of 24.79338
: : : bk stops
: : tree stops
: : lt of 25
: : lt stops
: : bk of 27.27272
: : bk stops
: tree stops
: lt of 25
: lt stops
: bk of 30
: bk stops
tree stops
>
```

## APPENDIX C

### ERROR MESSAGES

When LOGO PROFESSOR encounters a condition that it can not understand, it displays an error message describing the error. If the error occurs while a procedure is being executed, LOGO PROFESSOR will list the name of the procedure that the error occurred in, the line that was being executed and the appropriate message. This appendix contains the LOGO PROFESSOR error messages, along with an explanation of each. When "primitive," "text," or "procedure" is enclosed by parenthesis, this means that when the actual error message is displayed, the related primitive or procedure name will appear where shown.

#### **(primitive) doesn't like (text) as input**

A specification that has been made to a primitive is not a valid one.

EXAMPLE:

```
> fd [2]
fd doesn't like [2] as input
> fd 2
```

LOGO PROFESSOR would not know how many steps to proceed forward because the bracketed entry is not valid input for **fd**.

#### **(procedure) didn't output to (procedure)**

A procedure didn't output a result to another procedure which expected to use the value as one of its arguments.

#### **(text) has no value**

A variable has been created with no assigned value, and an attempt was made to access it.

EXAMPLE:

```
> make "dist 2
> :dist
RESULT: 2
> erns
> :dist
dist has no value
```

### **(text) is already defined as a variable**

An attempt was made to create a procedure whose name is the same as a currently defined variable.

EXAMPLE:

```
> make "dist 2
> to dist :x
dist is already defined as a variable
> er "dist
> to dist
```

### **(text) is already defined as a procedure**

An attempt was made to define a variable with the same name as a defined procedure.

EXAMPLE:

```
> to orange
> make "orange 2
orange is already defined as a procedure
```

### **(text) is a primitive**

An attempt was made to create a procedure with the same name as a primitive.

EXAMPLE:

```
> to forward
forward is a primitive
```

### **(text) is not a variable or a user defined procedure.**

An attempt was made to save or erase a name that is not a user defined procedure or variable.

EXAMPLE:

```
> er "forward
forward is not a variable or a user defined
procedure
```

### **(text) is not a variable**

An attempt was made to print the value of a word that has not been defined as a variable.

### **(text) is not a procedure**

An attempt was made to print the definition of a name that is not a valid procedure.

### **can not open the file (text)**

An error was encountered when attempting to retrieve or store a file.

### **bad procedure definition**

An unacceptable procedure definition was specified with the **to**, **edit**, or **define** primitive.

EXAMPLE:

```
> to abc [x y z]
bad procedure definition
```

### **can't expand**

There is not enough node space to expand any further.

### **division by zero**

An attempt was made to divide by zero.

### **filename too long**

An attempt was made to save a file with a name more than eight characters long.

### **filename extension too long**

An attempt was made to save a file with the extension longer than three characters.

### **I don't know how to (text)**

There is no procedure or primitive by the name entered. Check to make sure that your entry was spelled properly. If it is a procedure, make sure the procedure was loaded.

EXAMPLE:

```
>forwaed 100
I dont know how to forwaed
>forward 100
```

### **memory overflow**

LOGO doesn't have enough node space to complete the command or procedure.

### **not enough inputs to (text)**

There are not enough arguments to complete the procedure or primitive. For example, if the entry `fd` was entered, without entering the number of steps, this error message would be generated.

EXAMPLE:

```
> forward  
not enough inputs to forward  
> forward 100
```

### **out of memory**

LOGO PROFESSOR ran out of node space. It may be necessary to erase any unnecessary procedures.

### **too few items in (text)**

The `item` primitive was given a number which was greater than the count of the items in the list.

EXAMPLE:

```
> print item 4 [fee fie foe]  
too few items in [fee fie foe]
```

LOGO PROFESSOR was asked to print item 4 when there are only three items in the list.

### **too much inside ( ) 's**


More than one expression was enclosed inside a matching pair of parentheses.

### **too many ) 's**

An expression contained one or more unmatched right parentheses.

EXAMPLE:

```
> product (4 + 5) ) - 2  
> too many ) 's
```



**turtle out of bounds**

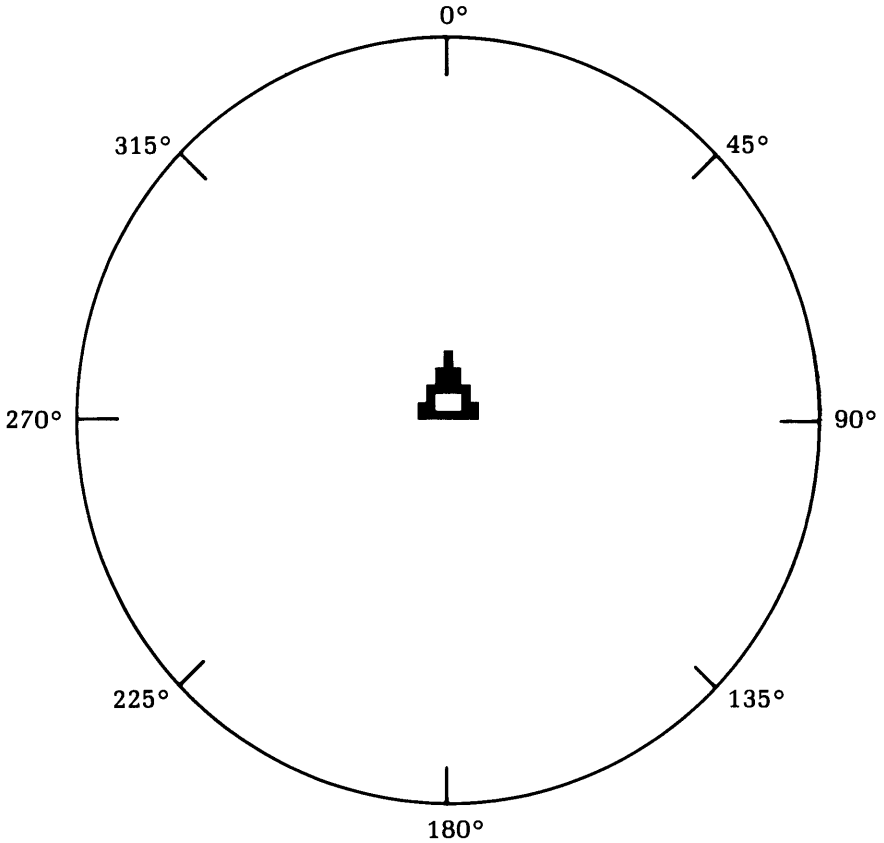
In “fence” mode, an attempt was made to move the turtle beyond the screen boundaries.

**word too long**

A word was keyed that was longer than 120 characters.

# APPENDIX D

## TURTLE COMPASS



Use this illustration to identify the turtle heading in degrees. When the turtle is in the home position, pointing straight up, the heading  $0^\circ$ . Change the turtle's direction by turning left or right a number of degrees. You can also use `setheading` to point the turtle in the desired direction.

# APPENDIX E

## INDEX TO HELP SCREENS

This is an index of the Logo Professor HELP SCREENS. At any time, while using Logo Professor, you may access an individual helpscreen by doing the following:

1. Push the HELP key (to enter "help")
2. Push the INDEX key (to enter "indexed help")
3. Enter the screen number desired.
4. Push [return]

Logo Professor will then bring you directly to the requested screen.

and 527-530  
arctan 714  
Arrow Keys 309-311  
ascii 754-755  
B: Drive 110, 113  
back (bk) 201, 208  
Backup 608  
butfirst 428, 431  
butlast 428, 432  
char 751-753  
clean 215  
clearscreen (cs) 216  
cleartext 441  
Command 105  
Conditionals 500  
cos 710-714  
cosecant 713  
cotangent 712  
count 414  
define 745-750  
Define 115  
definedp 542  
Demonstration 122  
dot 738, 739  
Dots 328  
edit (ed) 301  
Editor 301, 308-319  
349, 600-601  
empty 543  
equalp 536  
erase (er) 617  
erasefile 607  
erns 619  
erps 618  
expand 612, 616  
fence 230, 231  
first 428-429  
forward (fd) 201, 203, 208  
fput 433, 435, 437  
fullscreen 226, 228  
Garbage Collections 615  
Graphics 200  
heading 733, 735  
HELP 118  
hideturtle 212  
home 214  
if 500, 505, 508, 510-511  
in 761  
INDEX 108, 606  
Indexed Help 119  
Inputs 201  
int 716  
item 416  
Item 344, 414-415  
keyp 537  
last 428, 430  
left (lt) 205, 208  
LINE 314  
list 344, 400, 402, 410, 416  
420-421, 439, 440  
listp 539  
lput 434, 436, 438  
Math Functions 220  
make 325-328, 330-331,  
341-342, 345, 543  
mem 615  
Menu 756  
name 325-328, 330-331  
namep 538  
nodes 609-616  
notrace 709  
not 527, 533-534  
numberp 540  
or 527, 531-532



out 762  
output 526  
pen 733, 736  
pendown 211  
penerase 212  
penreverse 212  
penup 211  
po 334  
poall 340  
poff 337  
pon 336, 337  
pons 340  
popr 335  
pops 339  
pos 733  
pots 339  
Primitive 102  
primitivep 545  
print 199, 342, 343, 403, 407  
PRINT 336, 338  
Procedure 105, 115, 300, 307, 348  
product 727  
quotient 723, 724  
random 720  
readchar 520, 521, 524  
readlist 341-342, 344, 345, 520, 522  
Recursion 700, 703-704  
recycle 613-614  
reminder 725, 726  
repeat 219  
RETRIEVE 106, 112, 605  
right (rt) 205, 206, 208  
round 716  
run 115, 439  
Screen Size 202  
secant 713  
Segments 615, 616  
sentence 418  
setheading 744  
setpen 740  
setpos 742-743  
setscrunch 729  
setx 741  
sety 741  
scrunch 730  
show 403, 412-413  
shownp 544  
showturtle 213  
sin 710-714  
SIZE 230  
splitscreen 226, 229  
square 217  
sqrt 728  
STOP 318, 349  
STORE 306, 317, 348, 600-604  
STYLE 226  
sum 722  
Tail Recursion 701-702  
tangent 712  
text 440, 747  
textscreen 226, 227  
thing 341-342, 346  
Title 114, 115, 330  
to 301, 302  
toplevel 513  
trace 0 709  
trace 1 706  
trace 2 707  
trace 3 708  
type 403, 406, 408-409  
UNDO 116, 315  
Variable 325-328, 330-331, 507  
window 230, 232  
word 400-401, 411, 417  
wordp 541  
Workspace 105, 116, 117  
wrap 230, 233  
xcor 733, 734  
ycor 733, 734  
.call 760  
.deposit 759  
.examine 759

# USER GUIDE INDEX

/ 46  
( ) 15  
[ ] 24, 91  
" 24  
: 20  
\* 14, 71  
= 15, 71  
> 15, 72  
< 15, 71  
- 14, 72  
+ 14, 72  
/ 14, 72  
and 35, 36, 72  
An Array 65  
Arrow Keys 18, 19, 69  
ascii 53, 73  
axis 52, 54  
back (bk) 10, 73  
back slash 46  
butfirst (bf) 28, 29, 73  
butlast (bl) 28, 29, 73  
Calling Procedure 91  
.call 53  
char 52, 73  
clean 12, 74  
clearscreen (cs) 12, 74  
cleartext 30, 31, 74  
Command 7, 91  
Command Mode 18, 91  
Conditional Expressions 34, 91  
Contents of Package 3  
Control Keys 18-20, 68, 69, 91  
cos 49, 74  
Cosecant 49  
Cosine 48, 91  
Cotangent 49  
count 28, 29, 74  
CTRL A 68  
CTRL B 68  
CTRL D 68  
CTRL E 68  
CTRL F 68  
CTRL H 68  
CTRL J 68  
CTRL N 68  
CTRL O 20, 68  
CTRL P 68  
CTRL Q 20, 68  
CTRL V 68  
CTRL W 20, 68  
define 45, 74  
definedp 38, 75  
Degrees 92  
. deposit 53, 75  
diskette 4, 92  
disk drives 92  
dot 51, 75  
DRAW Key 14, 66  
edit (ed) 75  
Edit Mode 17, 18, 92  
empty 38, 76  
end 17, 76  
equalp 37, 76  
erase (er) 76  
erasefile 41, 76  
erns 40, 41, 76  
erps 40, 41, 76  
Error Messages C-1  
.examine 53, 76  
expand 42, 43, A-1  
fence 13, 77  
Filename 7, 92  
first 29, 77  
forward (fd) 7, 10, 77  
fput 28, 77  
Full Help 5  
fullscreen 9, 77, A-1  
Function Keys 5, 66, 67, 92  
Hardware 92  
heading 50, 77  
Help, Full 5  
Help, Indexed 5  
HELP Key 5, 66  
Help Menu 5, 92  
hideturtle (ht) 11, 12, 78  
home 12, 78  
if 34, 35, 44, 78  
in 53, 78  
Indexed Help 5  
INDEX Key 7, 8, 42, 66  
index 7, 8, 42, 66, 78  
int 46, 78  
integer 46, 93  
item 29, 79

keyp 37, 79  
 last 27, 29, 79  
 left (lt) 10, 79  
 LINE Key 19, 66  
 list 27, 79  
 listp 37, 79  
 lput 28, 29, 80  
 Machine Code Primitives 53  
 made 21-23, 30, 80  
 Media 93  
 mem 42, 43, 80  
 memberp 80  
 Memory 93  
 Monitor 93  
 MENU Key 53, 66  
 menu 53, 66, 80  
 Name 21, 93  
 namep 37, 80  
 nodes 42, 43, 81, 93  
 not 35, 36, 81  
 notrace 46, 81  
 numberp 38, 81  
 Operating System 93  
 or 36, 81  
 out 53, 81  
 output 24, 36, 81, 93  
 pen 52, 81  
 pendown (pd) 11, 12, 81  
 penerase (pe) 11, 12, 82  
 penreverse (px) 11, 12, 82, A-1  
 penup (pu) 11, 12, 82  
 po 33, 82  
 poff 7, 8, 33, 67  
 pon 7, 8, 33, 67  
 poall 33, 82  
 pons 33, 82  
 popr 33, 82  
 pops 33, 82  
 pos 50, 82  
 pots 33, 82  
 Primitive 6, 7, 93  
 primitivp 39, 82  
 PRINT Key 7, 8, 33, 67  
 print 24-26, 28, 29, 82, 83  
 Procedure 6, 7, 17, 94  
 Procedure Names 17  
 product 48, 83  
 quotient 47, 83, 94  
 random 49, 83  
 readchar (rc) 30, 31, 83  
 readlist (rl) 30, 31, 83  
 Recursion 44, 45, 94, B-1-B-4  
 recycle 42, 43, 83  
 remainder 48, 84  
 repeat 11, 84  
 RETREIVE Key 7, 8, 41, 42, 67, A-1  
 retrieve 7, 8, 41, 42, 67, 84, A-1  
 right (rt) 10, 84  
 round 46, 47, 84  
 run 45, 46, 85  
 Sample Procedures 59-65  
 Screen Modes 9, 94  
 scrunch 52, 85  
 Secant 49  
 sentence (se) 27, 85  
 setheading (seth) 52, 85  
 setpen 52, 85  
 setpos 51, 86  
 setscrunch 52, 86, A-1  
 setx 52, 86  
 sety 52, 86  
 show 26, 86  
 shownp 38, 86  
 showturtle (st) 11, 12, 86  
 SIZE Key 13  
 sin 49, 86  
 Sine 48  
 splitscreen 9, 86  
 sqrt 48, 86  
 stop 34, 35, 44, 87  
 STOP Key 18, 19, 44, 67  
 store 6, 8, 17-19, 40-42, 67, 87, A-1  
 STORE Key 6, 8, 17-19, 40-42, 67, A-1  
 STYLE Key 10, 24, 68  
 sum 47, 87, 94  
 Tangent 49  
 text 31, 45, 87  
 textscreen 9, 24, 87  
 thing 22, 23, 88  
 to 17, 18, 88  
 toplevel 35, 88  
 towards 88  
 trace 46, 88, B-1-B-4  
 Truncate 94

Turtle Compass D-1  
Turtle Graphics 6, 9, 94  
type 25, 26, 89  
UNDO Key 20, 68  
Value 22, 23, 94  
Variable 21-23, 95  
Whole Numbers 46  
window 12, 89  
Words and Lists 24

word 26, 27, 89, 95  
wordp 38  
Workspace 6, 7, 42, 43, 95, A-1  
wrap 12, 89  
xcor 50, 89  
x Coordinates 50  
ycor 50, 89  
y Coordinates 50

QX-10 is a registered trademark of Epson American, Inc.