RM NIMBUS

RM LOGO

RESEARCH MACHINES

Discrepancies in the RM Logo Reference Manual
Reference Section

`arcl, arcr`

The angle of arc is limited to + 360 degrees, not mod 360 as with the other "radial" commands.

`bload`

If you bload the same extension more than once, identical sets of routines will be loaded, as shown by the `bloaded` command.

`build`

It is possible to use build to edit an existing procedure, but not possible to use `edit` to build a non-existant procedure.

`colour, setc, bg, setbg`

When real numbers are supplied to `setc` or `setbg`, only the integer part (mod 16) is used.

`fence`

The fence is one-way: if a turtle is outside the screen area before `fence` is issued, it is possible for the turtle to enter the screen again. The fence then becomes active.

`join`

The maximum length of any single Logo word is 63 characters. If you use `join`, or any similar command, to create a word of more than 63 characters, a word of exactly 63 characters will be produced.

`pennormal, penreverse`

These primitives do NOT lower the pen as stated in the manual.

`power number1 number2`

If number1 <= 0 and number2 is not an integer, an error is generated.

`readfiled`

When using this primitive, if any special Logo character is found in the line that is being read, the line is truncated from the point of the character. The next `readfiled` command starts with the next line in the file.

`sense`

If a turtle hits the edge of the screen, with sense on, Logo does NOT perform a `throw 'touchturtle`. It does a `throw 'fence`.

`touch, point`

These two primitives only work within the screen area.

Other Chapters

Page 7.3

Line 5 of the procedure check.key should be:

```
if :button = 'b [backward 10]
```

Page 10.5-6

The variable `:objects` (used in the procedure `scan.list`) should be the variable `:names`.

Page 11.7

There is an error in bracketing in the example `delete.records.from.file`. Lines 3, 6 and 7 should be:

```
unless infile: filename [say [cannot
  find] <+ :filename escape]
unless closefile:filename [say
  cannot close] <+ :filename escape]
unless erasefile:filename [say
  [cannot erase]  <+ :filename
  escape]
```

Page 16.7

The interrupt mentioned on the fourth line from the bottom should be OD1 hex and NOT OC1 hex as stated.

PN14394

RM Logo
PN 14394

Customers comments are of great value to us in improving our computer systems, publications and services. If you would like to make any comments, please use the reply-paid form at the back of the manual.

Authors: Cathy M. Hand and Barry Morrell.

Editor: Nicola Bourdillon.

Illustrations by Jane Hannah and Inkwell Studios.

Typeset by direct transfer from Research Machines Network to Linotron 202 at Oxford Publishing Services, Oxford.

Printed by The Hazell Press, Wembley.

Research Machines Limited, Mill Street, Oxford OX2 0BW.

Preface

RM Logo is a full and versatile language.
This book, RM Logo, and the accompanying 'Beginning RM Logo' provide:

- an introduction to the RM Logo language
- an introduction to using RM Logo on Nimbus
- a language reference
- a quick reference card

The aim is for you to understand Logo quickly and easily and to use the extra features provided in RM Logo.

Beginning RM Logo by Hilary Shuard and Fred Daly, is available from Research Machines, PN 14393. It gives an introduction to RM Logo.

This book, RM Logo, is divided into two sections and an index:

- a Concepts section
- a Logo primitives section
- index

The Concepts section consists of sixteen chapters which progress quickly from starting with Logo to introducing the special features of RM Logo. Many example programs are included for you to try out, and are given in the following format:

The `1:` prompt
: Anything following the `1:` is to be typed in by you.

Long Lines
: Some Logo lines are longer than the width of these pages. Any indented Logo lines in this book are a continuation of the previous line, to be typed in without pressing the <ENTER> key. If the line is longer than the screen width, it appears on the next screen line.

Procedures
: Logo procedures are shown with a `build` command, a blank line and then the text as it appears in the edit window. You will need to type in all of the text except for the first line (which results from the `build` command).

The first chapter includes <ENTER> at the end of the Logo lines to remind you to press the <ENTER> key. Later chapters leave it out.

The Primitives section is the reference part of the book. It lists all of the Logo primitives in alphabetical order, the special Logo characters, keywords and signals. Some examples are included to show how the words are used in programs.

The index is at the back of the book but it covers only the Concepts section. As the primitives are listed in an alphabetical order, they are easy to find without the additional task of looking in an index.

The Logo examples in the book have been tested but we cannot guarantee a perfect performance when you use them.

Contents

Part One Concepts

Chapter 1
Getting Started

RM Logo

Logo is a computer language that originated in the 1960s and has kept growing in popularity. It originated as a language reflecting a philosophy of learning: beginners start with little understanding but, through learning, become increasingly more sophisticated. In turn, Logo has become a full language, offering a complete range of activities but retaining its one original founding principle: it is easy to use.

Logo can be used without any knowledge of the internal workings of the computer, and without any other specialized knowledge. You can start to make things happen with instructions such as `forward` and `backward`. Such words carry their every day meaning into the Logo language and help to make it simple to use.

Turtle graphics are sometimes believed to be the whole of Logo. The graphics are certainly important, generating much of the enthusiasm for Logo. The turtle is a marker on screen which allows you to draw pictures — but the opportunity exists for you to add a floor turtle (a mechanical device which moves around on the floor in response to Logo instructions). RM Logo is also able to let you use up to eight turtles on screen at the same time, to change the shape of the turtle (to a bicycle for example) and to let turtles sense other events happening on screen.

Logo's ease of use is especially beneficial in an educational environment because Logo doesn't ask for specialized knowledge or experience. For example, turtle graphics lets you draw pictures on screen without needing

to know about coordinate geometry. However, you are likely to learn without realizing it as you explore Logo! For example, you might learn by experiment that the turtle takes 360 steps to completely turn around. In finding out this, the idea of 360 degrees in a complete turn is also suggested.

Logo instructions are carried out immediately after you type them in. Interest is maintained by actually seeing things happen after typing an instruction. If you want a set of instructions to be carried out together then you can put them into a procedure. One of the features of Logo is the way one procedure can call another, and RM Logo will let you run a number of procedures in parallel.

RM Logo is a versatile language and the aim of this book is to let you find and use the full features of the language.

Your RM Logo Disk and Guides

You will have received a disk called the RM Logo Disk.

(As with any such distribution disk, the contents must be copied to another 'working' disk which is used. Then, if your working disk is lost or damaged, another copy can be made from the master.)

This disk contains all the files needed to use RM Logo and to demonstrate its capabilities.

The files that RM Logo needs to work are:

LOGO.EXE — the Logo interpreter
PWORD1.OVR — the Logo editor
START.LGC — the standard Logo environment
PROWORD.EXE — to use the Logo editor outside of Logo

These are accompanied on the disk by some demonstration files. These have the file extension `.def`

An explanation of how to load and run the demonstration files follows in the section, 'Loading, Running and Saving Files'. Demonstration files are treated just like files or procedures that you have created yourself.

As well as this guide which briefly introduces RM Logo and includes a reference section of the language, an introductory book is available from Research Machines: Beginning RM Logo by Hilary Shuard and Fred Daly, PN 14393.

Starting Up

On a Network Nimbus

If you have a Network Nimbus, it is assumed that everything has been prepared by the Network Manager for you to use RM Logo and that you can join the following instructions where they specify you type:

```
logo
```

On a Standalone Nimbus

Your Nimbus should be switched on and displaying either the initial Welcome screen or a drive prompt (such as A>).

- When Nimbus is displaying the welcome screen and a message `Welcome - Please supply an operating system` insert the Logo working disk into a drive.
  Either amend the date and time or press <ENTER> twice to keep the date and time shown.

  The drive prompt of the drive holding the Logo working disk appears on the screen.

- If you start with a drive prompt on the screen, make sure that your Logo disk is either in that drive or can be accessed from it.

Following the screen prompt, type:

```
logo
```

and press the <ENTER> key.

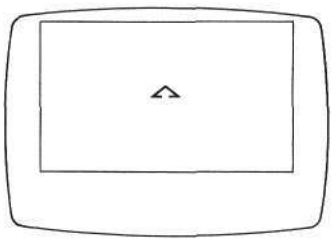
You are now in RM Logo!

The screen should be clear except for `1:` in the top lefthand corner. The prompt `1:` indicates that Logo is waiting for you to type a command at the keyboard. When Logo replies to you on screen, the `1:` prompt doesn't appear. In the examples in this book, use the prompt to distinguish between Logo's responses and the instructions you type in. You type in everything following the `1:`.

The position of the `1:` prompt in the top left corner indicates that you are in *text mode*: you cannot draw graphics in this mode. However, by typing:

```
1: clearscreen <ENTER>
```

the screen changes to display a triangle shape at the centre. This triangle is called the *turtle* and indicates you are in *graphics mode*. All but five lines at the bottom of the screen are used for drawing with the turtle. The five lines are reserved for commands (text).



Text characters appearing in graphics mode are double the width of text characters typed in text mode or through the Logo editor.

Leaving Logo

When you want to leave Logo, type:

```
1: exit <ENTER>
```

or

```
1: goodbye <ENTER>
```

This will return the screen prompt which existed before you typed `logo` to enter into the Logo system. However, before leaving Logo, remember to save your procedures that you might want to use again.

If you only want to get out of a running program or procedure, press the <ESC> key. The message `Stopped!` appears on screen.

Loading, Running and Saving Files

Files are 'containers' for your programs or procedures. Each of the demonstration files for example contains a program to demonstrate features of RM Logo. A list of these files can be seen on screen by typing:

```
1: demofiles
```

Each can be loaded typing the filename preceded by `load "` For example, load the demonstration file `cage.def` by typing:

```
1: load "cage.def
1:
```

Logo returns the prompt `1:` to show the file has been successfully loaded. If the file can't be loaded then an error message comes on screen to indicate why.

The contents of the demonstration files are run by typing the filename. These are the same as the name so file `cage.def` can be run by typing:

```
1: cage
```

The above file has already been saved but if you create programs or procedures that you want to keep, use the `save` command to transfer them to disk.

Introducing Logo

Logo Primitives

`textscreen` and `clearscreen` are words which do a specific action in Logo. Such words are known as *primitives* and are built into Logo. When you type a primitive as a command, you need to press the <ENTER> key to make it take effect.

Primitives introduced or connected with material in following chapters, are listed at the end of each chapter. If they are listed, but not included in the chapter, then they will be explained in the reference section at the end of this guide. Primitives are important because they act as basic building blocks for more complicated commands.

Some primitives need no other information to perform their actions. For example:

```
clearscreen
```

Others need *inputs* which you can change to get different effects. For example:

```
forward 50
```

This moves the turtle forward by a fifty steps.

```
backward 27
```

This moves the turtle backwards by a twenty-seven steps.

```
left 90
```

This turns the turtle left by a ninety degrees.

```
right 43
```

This turns the turtle right by a forty-three degrees.

Notice there is a space between the primitive and its input. Try omitting this and see what happens. Logo will warn you with an error message and you can then retype the correct line.

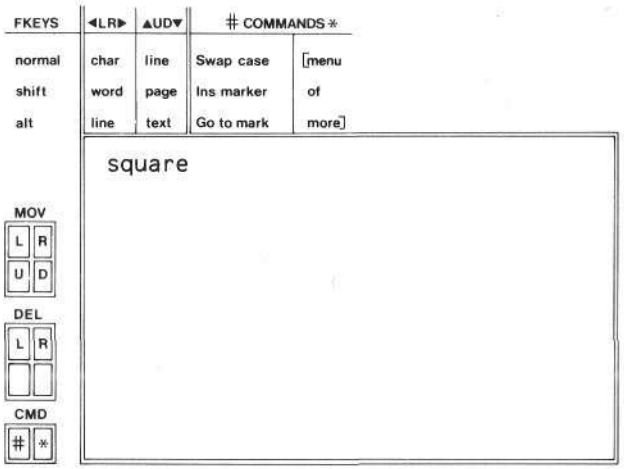
Procedures

You can build new Logo commands from any commands that Logo already knows. These new commands are called *procedures* and they are created using the `build` primitive. For example, the following procedure draws a square.

Type:

```
1: build 'square <ENTER>
```

The screen changes to the following format:



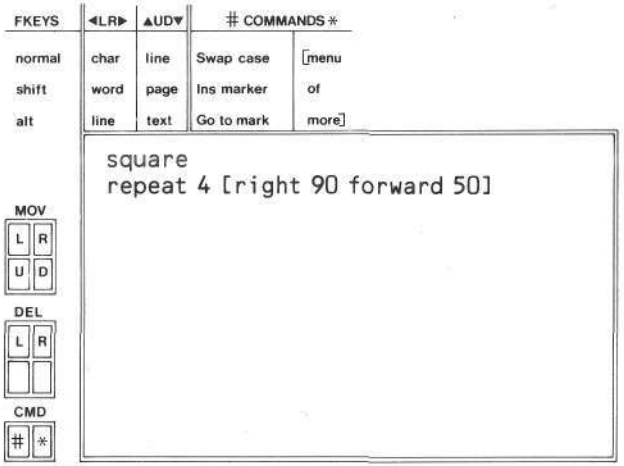
This is the edit window and RM Logo is now in *edit mode*. The cursor is positioned under the first letter of the procedure name: in this case `square`.

Now you can type in the rest of the procedure `square`.

```
repeat 4 [right 90 forward 50]
```

If you make a typing error then remove it by pressing the <BACKSPACE> key. This rubs out the character to the left of the cursor. (More editing commands are explained in Chapter 4, Using the Editor.

The complete edit window containing the procedure `square` looks like this:



Leave edit mode and return to graphics mode by pressing the escape key marked <ESC>. Now you can run the procedure by typing:

```
1: square<ENTER>
```

If you want to change the contents of `square` you can do so using `edit 'square` or `build 'square`. The contents will be displayed and you can change them using the keys described in Chapter 3.

Try creating another procedure to draw a triangle. The `build` command will take you into the edit window.

```
1: build 'triangle<ENTER>
```

The following text is the body of the procedure and, except for the first line, will need to be typed in by you.

```
triangle
repeat 3 [forward 80 left 120]<ESC>
```

So far procedures have been made using primitives. However, procedures can also be built up of other procedures. For example, procedures `square` and `triangle` can be used to create another procedure `house`:

```
1: build 'house

house
square
right 90
triangle
```

Produce the picture by typing:

```
1: house
```

All the following examples in this guide are shown with the `build` command, a blank line, and then the text as it appears in the edit window.

Abbreviating Primitives

Logo allows you to type short forms of some primitives. For example, instead of `forward` you could type `fd`. The short forms of primitives are given under the description of each primitive in the reference part of this book.

Logo also allows you to type a number of primitives on the same line, if you wish. If you keep on typing when you reach the end of the line, text will continue onto the next line.

For example, instead of typing the following:

```
1: forward 50
1: left 120
1: forward 50
1: left 120
1: forward 50
1: left 120
```

you could type:

```
1: fd 50
1: lt 120
1: fd 50
1: lt 120
1: fd 50
1: lt 120
```

or

```
1: fd 50 lt 120 fd 50 lt 120 fd 50 lt 120
```

You can also link commands together with **and**, for example:

```
1: forward 50 and right 45
```

This is mainly used to make Logo more readable.

Words and Lists

Logo *words* are similar to words of spoken languages in that they consist of characters. For example:

```
'cat
'computers
'Train
'2and3
```

Notice that they all start with a single quotation mark but there isn't a terminating mark. This tells Logo that the string which follows is a word. You can use the character ' on its own to show a word with no characters in it (called the *empty word*).

You can join words to form longer words and split them up to form shorter ones. These operations are covered in detail in Chapter 9.

A *list* consists of zero or more *elements* surrounded by square brackets. Each element can be a Logo object: a number, a word or another list. For example:

```
[cats dogs birds]
[[a b] [c d]]
[a b [a[c]] d]
[]
```

You can handle lists in the same way as words: they can be printed, broken into smaller lists or joined to make longer ones. In fact, some primitives will operate upon words and lists. For example:

```
1: say [cats dogs birds]
cats dogs birds

1: say 'cats
cats
```

Lists are described in more detail in Chapter 9.

Names

Logo names can consist of letters, numbers and punctuation characters. Logo primitives (listed in the reference section of this book) can be written in upper or lower case, but those you define yourself need to be in lower case. Upper case characters slow Logo down a little. All the Logo examples in this book are given in lower case.

Use punctuation characters and underlines to make names more readable. Spaces cannot be used within a name because they are used to separate different items such as procedure names and inputs.

```
equilateral.triangle
isosceles_triangle
```

You have already come across one method of assigning names: when defining a procedure, you name both the procedure and its inputs. You can also name constant and variable items by using the `make` primitive. For example, the following command gives the name `angle` a value of 90:

```
1: make 'angle 90
```

Here, the `make` primitive creates a 'box' and gives it the name `angle`. It then puts the value 90 into the box. If you now want to look at the contents of the box, you prefix its name with : (dots), as shown below:

```
1: say :angle
90
```

You can also look at the contents of a named object by using the `value` primitive. For example:

```
1: make 'angle 90
1: say value 'angle
90
```

This may seem more long-winded than using dots and in cases like the one above it is. However, `value` allows you to do things you can't do with dots. For example:

```
1: make 'bathroom [bath sink towel
       rubber.duck]
1: say value first [bathroom kitchen bedroom]
bath sink towel rubber.duck
```

Numbers and Arithmetic

Logo numbers consist of a string of one or more digits and may contain a decimal point. You must not precede them with a quotation mark because they are not words.

You can perform arithmetic upon numbers and two types of operators are available.

Infix operators go between the items and include + (add), — (subtract), / (divide) and * (multiply). For example:

```
1: say 10 * 13
130
```

Prefix operators go before the items and include `add`, `divide`, `multiply` and `remainder`. For example:

```
1: say add 10 13
23
```

Other functions available include `cos`, `pi`, `sin` and `tan`.

Special Characters

This section describes characters which have a special meaning in Logo, for example:

```
' : [ ]
```

If you want to use any of these special characters as ordinary characters in text, you must prefix each one with the special character (\) whenever it is used. The escape character will be printed by `print` but not by `say`.

Quotation mark or '

This indicates that what follows is to be used as a word or a name. It is not the name of a procedure so it can't be run!

Dots or :

This refers to the contents of a variable.

Square brackets or [ ]

These are used to surround a list. Note that words in the list do not necessarily start with a quotation mark.

Backslash or \

This is the special character indicating that the subsequent character is to be treated as an ordinary text character and not a special character. For example:

```
1: say  'Cathy\'s.shoes
```

will show as `Cathy's.shoes` on the screen. Try it without the backslash to see the difference.

The sequence\\ prints as\.

\ can also be used to give the corresponding character of subsequent hexadecimal digits. For example:

```
1: say  '\01 <ENTER>
```

produces a face character.

Round brackets, parentheses or ( )

These are used to group items of an expression into the order in which you want Logo to evaluate them.

Comment or ;

Comments in a program are preceded by a semicolon. For example:

```
if :x = 0 [stop] ;Finished
```

Setting Up A Logo Microworld

You may want to alter or reduce the facilities that Logo offers, or extend them to produce a Logo learning environment, or *microworld*. You can:

- Redefine some of the primitives to change their effect. For example, you could redefine `forward` so that `forward 10` moves the turtle by 100 steps
- Treat some of your procedures as 'primitives' which cannot be edited by users
- Rename primitives for use with procedures from other dialects of Logo
- Create a news file whose contents are displayed whenever someone starts up the system

The actions needed are described in Chapter 15.

Chapter 2
Graphics

Introducing RM Logo Graphics

Logo's immediate appeal is due largely to its graphics. RM Logo is a very powerful version of Logo that lets you use the fast and powerful graphic facilities of the RM Nimbus.

Logo is an excellent language for learning and exploring programming because it gives you a symbol to think with; the turtle shows you where you are and which way you are going. You can make the turtle turn, move in straight lines or arcs; drawing, not drawing or erasing as it goes. You can define the shape of your turtle, make it print its shape and make it invisible.

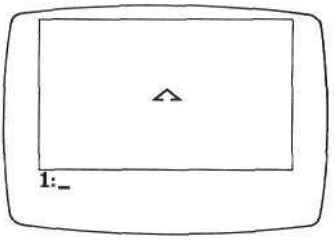
The first section of this chapter describes Logo's Turtle Graphics, including most of the actions possible with one turtle. Multiple turtles are explained in Chapter 13.

This is followed by an explanation of using colours. You can choose the colours of the turtle, of the lines it draws and of the background it draws on.

As well as turtle graphics, where your instructions move the turtle *from* its present position, Logo has absolute graphics, where the screen is defined as a set of coordinates which your instructions draw *to*. The final section of this chapter describes this use.

Turtle Graphics

Type `clearscreen` to go into graphics mode. The graphics screen has a turtle shape in the centre and the bottom five lines are reserved for text. To clear pictures from the graphics area, use the command `clean`. Text can be wiped off the writing area at the bottom of the screen using `cleantext`.



The area that the turtle can be seen to move in is the grid shown above. You can put a boundary around this area using `fence` which will stop the turtle going out of the area. When `fence` is not applied, the turtle can move out of the area and out of sight.

Directing and Moving the Turtle

The simplest and probably the most used commands to move the turtle are `forward` and `backward`. The turtle moves either forwards or backwards in the direction it is pointing.

The turtle has a pen which can be lowered to mark wherever the turtle moves (`drop`) or raised so that the turtle moves without tracing its direction (`lift`). The pen is usually down when you enter Logo.

Following the command `clearscreen`, the turtle points "north" or up towards the top of the screen. The direction that the turtle points is called its *drawing heading* and measured in clockwise degrees from "north".

Typing:

```
1: right 90
```

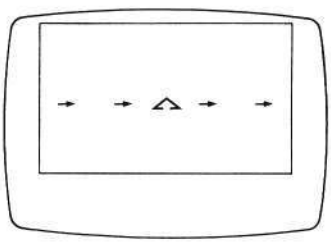
turns the turtle to face "east".

`left`, `right` and `seth` are all commands that change the direction that the turtle is pointing. Explanations of how to use them are covered in the reference section.

You can set the direction for the turtle to move without changing the direction it is pointing. The direction the turtle moves in is known as the *movement heading* and can be changed using `setdir`. The turtle's initial speed, like the movement heading, is preset to zero degrees. Use `setspeed` to make it move. For example:

```
1: cs
1: setdir 90
1: setspeed 20
```

the turtle will move to the "east" while pointing "north".



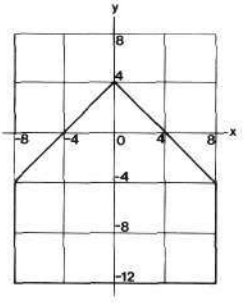
Whenever you want the turtle to stop moving, reset it's speed to 0. To return it to the centre of the graphics screen, use the `centre` primitive.

Changing the Turtle Shape

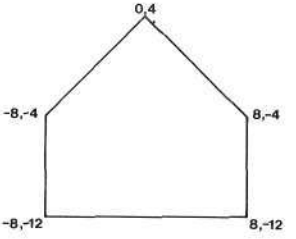
The turtle has appeared until now as an arrow shape but this can be changed. A new shape can be created using `defineshape` and taken up by the turtle using the command `setshape`, or by using the procedure `esh.def` given on your RM Logo Disk. The following is an example of the first method.

You might find it easiest to draw out the shape on paper before you actually use `defineshape`. The following steps show you how to define a house as a turtle shape.

- Draw the shape out on a grid. Your shape should be drawn a sensible size to move around the screen. The house has been drawn on a grid 4 x 5 units



- Fill in the cartesian coordinates for each point.



- Write out the coordinates as a list. This list follows the name of the shape, house.

The whole command can be completed now.

```
1: defineshape [house [-8-12] [8-12] [8-4] [0 4]
        [-8-4] [-8-12]]
```

The last coordinate pair are the same as the first so that the shape is 'closed'.

To change the current turtle's shape to the new one, type:

```
setshape 'house
```

To get back to the original turtle shape, type:

```
1: tell 1
1: vanish
1: tell 1
```

The `tell 1` command shows you are directing turtle number one. You can have up to eight turtles on the screen (see Chapter 14), numbered 1 to 8.

You can also choose to make the turtle disappear from view using `hideturtle`. Bring it back into view with `showturtle`.

Using Colour

RM Logo allows you control over the:

turtle colour
pen colour
background colour

(but the border colour surrounding the Logo screen remains the same).

You can change and check the colours using the primitives:

`bg`
: Returns background colour

`colour`
: Returns current turtle colour

`pc`
: Returns pen colour

`setbg`
: Sets background colour

`setc`
: Sets current turtle colour

`setpc`
: Sets pen colour

The table below shows the numbers associated with colours:

Number	*Colour*
0	`black`
1	`dark.blue`
2	`dark.red`
3	`purple`
4	`dark.green`
5	`dark.cyan`
6	`brown`
7	`light.grey`
8	`dark.grey`
9	`light.blue`
10	`light.red`
11	`magenta`
12	`light.green`
13	`cyan`
14	`yellow`
15	`white`

The colours you can get on your monitor depend on the monitor you are using. The full range of colours listed above can be seen on a monitor capable of generating all sixteen colours. You might otherwise be using a colour monitor which displays eight of the colours, or a monochrome monitor giving sixteen shades of grey.

You can use colour numbers in commands. For example:

```
1: cs
1: setbg 1
```

will change the background colour to dark blue.

You may find it easier to remember the colours by name rather than by number. The procedure `colours.lgp` on your RM Logo disk allows you to use the names of the colours as well as the number. For example:

```
1: cs
1: setbg dark.blue
```

will change the background colour to dark blue.

If you try `setbg cyan` and `setbg magenta` you will see that cyan is greenish blue and magenta is crimson.

XOR Plotting

There are two ways of drawing on the screen:

- destructive overdrawing the default
- non-destructive overdrawing (XOR or exclusive OR plotting)

With destructive overdrawing, you draw over anything that is already there, and the colour of the new drawing replaces the covered part of the old. Non-destructive overdrawing merges the colours of the new and old.

If you want to plot colour in the non-destructive mode, you do so by giving the `penreverse` command. For example:

```
1: setpc 14
1: penreverse
```

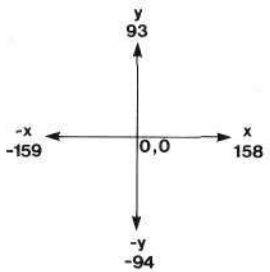
gives non-destructive plotting in yellow. You can return to the default over plotting by using `pennormal`.

The turtle shape is always drawn in non-destructive mode.

It is also worth pointing out the colour drawn by the turtle's pen depends on the background colour. A black background will guarantee an accurate turtle colour.

Absolute Graphics

The movement primitives `forward, back, left` and `right` all move the turtle relative to its current position. If you wish, you can move the turtle relative to the system of coordinates shown below by using the primitives `setx, sety` and `setpos`.



and `setpos [-159 -94]` moves the turtle to the bottom left corner of the screen (without drawing a line).

Logo allows you to draw a line between any two points on the screen using `line`, and to draw coloured points on the screen using `setpoint`.

Summary of Primitives

`arcl`
: Draws left hand arc

`arcr`
: Draws right hand arc

`bg`
: Returns background colour

`centre, center, (ct)`
: Moves turtle to home position

`clean (cl)`
: Clears graphics area of screen

`cleantext`
: Clears text area of screen

`clearscreen (cs)`
: Clears screen

`colour (colour)`
: Returns turtle colour

`defineshape (dsh)`
: Defines turtle shape as list

`dir`
: Returns direction of movement

`drop`
: Drops turtle's pen

`eraser`
: Erases lines over which it passes

`fence`
: Prevents turtle going off screen

`fenceq`
: Tests if `fence` has been used

`fill`
: Fills area of screen

`forward (fd)`
: Moves turtle forwards

`heading`
: Returns turtles drawing heading

`hideturtle (ht)`
: Hides turtle shape

`label`
: Prints text in graphics area

`left (lt)`
: Turns turtle to left

`lift`
: Lifts turtle's pen

`line`
: Draws line on screen

`near`
: Tells you if turtle is close to another turtle

`nofence`
: Allows turtle to move off screen

`nosense`
: Cancels `sense` command

`pc`
: Returns current pen colour

`penreverse (px)`
: Lowers the turtle's pen to draw in XOR mode

`point`
: Returns position of current point on screen

`revereseq`
: Tells you if turtle's pen is reversed

`right (rt)`
: Turns turtle to right

`rubber`
: Erases lines which turtle passes over

`sense`
: Turtle senses presence of another turtle or change in background colour

`setbg`
: Changes background colour

`setc`
: Changes turtle colour

`setdir`
: Changes turtle's direction of movement

`seth`
: Changes turtle's direction of drawing

`setpc`
: Changes pen colour

`setpoint`
: Sets a coloured dot on screen

`setpos`
: Changes turtle's position to [x y]

`setshape`
: Changes current turtle shape

`setspeed`
: Gives turtle a constant speed

`setx`
: Moves turtle in x direction

`sety`
: Moves turtle in y direction

`shape`
: Returns current turtle shape

`shapedef`
: Returns shape as a list

`shapes`
: Returns list of shapes defined

`showturtle (st)`
: Makes turtle visible

`speed`
: Returns turtle's current speed

`stamp`
: Stamps a shape on screen

`tell`
: Addresses subsequent commands to named turtle(s)

`textscreen (ts)`
: Reserves screen for text

`told`
: Returns name of current turtle

`touch`
: Returns the background colour under the pen

`towards`
: Returns heading and distance to named point

`turtles`
: Returns list of active turtles

`upq`
: Returns `'true` if pen is up

`vanish`
: Removes turtle(s) from list of active turtles

`wrap`
: Wraps turtle movement around screen

`wrapq`
: Tells you if `wrap` has been selected

`xcor`
: Returns turtle's x coordinate

`ycor`
: Returns turtle's y coordinate

Chapter 3
More On Procedures

Procedures allow you to approach programming problems in a structured and logical way. You can break a complex problem into its smaller components, and tackle the smaller problems by building procedures to solve each one separately. Use of procedures encourages a structured approach that often leads to efficient, elegant programs that are easy to check and easy to develop.

Some languages, including Pascal and Logo, allow you to store your procedures independently, and so to build up a library of the procedures you want to use repeatedly.

In Logo, procedures will run without a calling program so you can test them as you build them. You can edit and delete them, or you can treat them exactly like the primitives. It may be helpful to think of building a procedure as "teaching Logo a new command".

Building and Scrapping

You build procedures using the `build` primitive described in Chapter 1. If you want to, you can delete them from the workspace by using `scrap`.

Listing Available Procedures

You can get a list of all the procedures you have copied or created in the workspace by using `titles`.

```
1: say titles
square triangle polyspi
```

Using Inputs to a Procedure

Primitives such as `forward` and `left` use inputs to know what precise action to take: for example, `forward 55` for the turtle to go forwards 55 units. Your procedures can have inputs too.

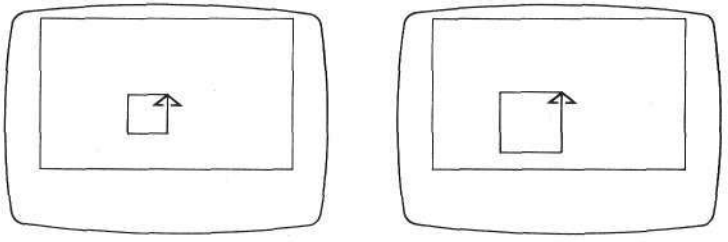
Change the procedure `square` in Chapter 1 by typing:

```
1: edit 'square
```

so that it looks like:

```
square 'side
repeat 4 [left 90 forward :side]
```

This allows you to change the size of the square each time the procedure is used. Here is the screen picture after running first `square 50` and then `square 75`.



This is what happens. `:side` in the first line creates a 'box' called `side`. When you run the procedure, the value following the procedure name goes into the 'box'. The value in the box is used whenever `:side` appears in the procedure. So, when you type in:

```
square 50
```

Logo takes the number 50 and puts it into the box. It then uses the contents of the box as the value for the primitive `forward` in the line
`repeat 4 [left 90 forward :side].`

The colon `:` is called *dots* in Logo and it indicates the contents of something (in this case the box called `side`).

If you want, you can have a number of inputs to a procedure. For example:

```
1: build 'rectangle

rectangle 'side1 'side2
repeat 2 [forward :side1 right 90
        forward :side2 right 90]
```

This procedure would need two inputs to run. For example:

```
1: rectangle 30 60
```

If you only give one input by mistake, an appropriate error message will appear on the screen. `rectangle` will take the first two inputs if you give more than two numbers, and appear to ignore the others.

Getting Results from Procedures

Your procedures can also return values after doing tests or calculations. They do this using the `result` primitive. For example, the following procedure calculates the square of a number and returns the result:

```
1: build 'number.square

number.square 'no
result :no * :no
```

The full stop is used in `number.square` to make it more readable. You can't use a space for this purpose here; if you did, `number` would be used as the procedure name.

When you give this procedure a number as an input, it outputs the square of the number and you can print this using `say`:

```
1: say number.square 13
169
```

Renaming Procedures

If you want to call a procedure by another name, use the primitive `rename`.

```
rename 'polygon 'six.sided.figure
```

This completely erases the name `polygon` and the procedure takes the new name `six.sided.figure`.

However you can rename a procedure during an edit by replacing the old name with the new name. This gives you two copies of the procedure: one as it was before the edit with the old name and text; one with the new name and edited text.

If, instead, you want to give the procedure an alternative name and still let it be known by its original name, you can use `alias`. For example:

```
alias 'six.sided.figure 'hexagon
```

will let you use either of the names `hexagon` and `six.sided.figure` for the previous procedure. When you change the contents of one, you change the contents of the other too.

Procedures as Lists

Sometimes, you might want to manipulate a procedure in the form of a list. `define` allows you to create a procedure in this way and `text` lets you list it in the same form. For example:

```
1: define [[square.number 'no] [result :no * :no]]
1: say square.number 12
144

1: print text 'square.number
[[square.number 'no] [result :no * :no]]
```

The input to `define` consists of a list of lists. The first list holds the procedure's title line. The rest consist of each procedure line in the form of a list.

`define` is most useful when you want to write procedures which define other procedures. It isn't worth using `define` to build a procedure from command level: if the procedure is a big one, you are likely to make mistakes by mismatching the square brackets. Use `build` instead for the procedures 'built' from commands.

An example of the use of `define` follows. It effectively runs a procedure as you are defining it.

```
1: build create

create 'name
define putfirst :name get.line readlist

1: build get.line

get.line 'text
if :text = [quit] [result[]]
run :text
result putfirst :text get.line readlist
```

When you now use `create`, each Logo line that you type in will be executed and then stored. When you type `quit`, the procedure will be created. You cannot have inputs to the new procedure in this version of `create`. For example:

```
1: create 'new.house
triangle
right 90
square
quit
```

Summary of Primitives

`alias`
: Renames procedure but remembers old name

`build`
: Invokes the editor

`bury`
: Buries procedures so that they cannot be edited, listed, saved, renamed or deleted

`define`
: Defines procedure in the form of a list

`edit`
: Invokes the editor

`editlist`
: Invokes the editor to edit a list and returns the list in the form it was given

`edlist`
: Invokes the editor to edit a list and returns the list as a list of lists

`expose`
: Unburies procedures

`rename`
: Renames a procedure

`scrap`
: Destroys a procedure

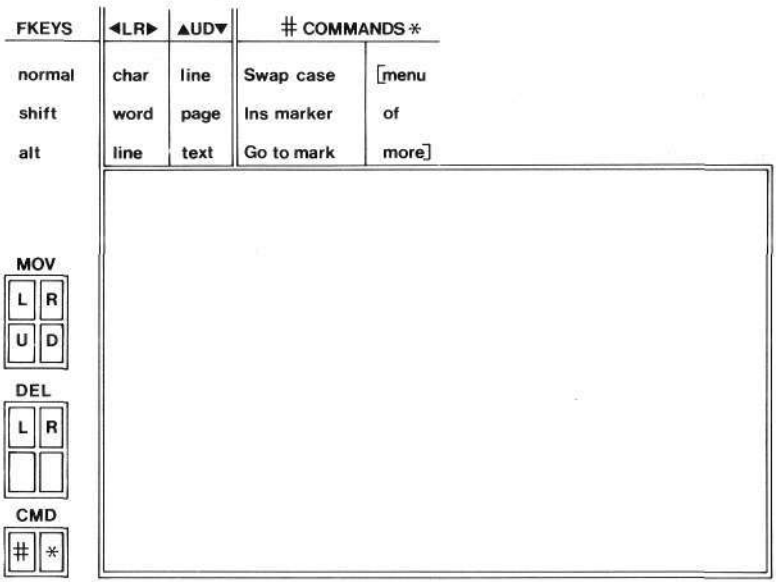
`titles`
: Returns list of (unburied) procedures

`text`
: Returns definition of a procedure as a list

Chapter 4
Using the Editor

The RM Logo editor is used to create and change your procedures. It can be called from either text or graphics mode. If you want to know more about any of the keys mentioned in this chapter, please refer to your Nimbus Owners Handbook which gives a full description of the Nimbus keyboard.

When you use `build` to create a procedure, the editor screen is displayed and it looks like this:



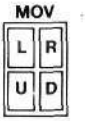
When you type in the lines of your procedure, they will appear in the central "window".

Function Keys For Editing

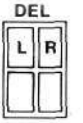
The keys on the left of the keyboard marked <F1> to <F10> are function keys and are used to edit text in the window.

They are displayed on the left side of the edit window, not with the numbers <F1> to <F10>, but showing their use:

cursor movement
(left right up and down)



text deletion
(left right up and down)



special command keys

Text at the top of the window tells you the effect of pressing one of these function keys either on its own or while pressing the <SHIFT> or <ALT> keys.

FKEYS	◄LR►	▲UD▼	# COMMANDS *	
normal	char	line	Swap case	[menu
shift	word	page	Ins marker	of
alt	line	text	Go to mark	more]

The top row
(identified by the word "normal" at its left hand side) shows the effect of pressing a function key on its own.

For example, pressing <F1> moves the cursor one character to the left and pressing <F9> changes the case of the character underlined by the cursor.

The middle row
(identified by the word "shift") shows the effect of holding down the <SHIFT> key and then pressing a function key.

For example, using <SHIFT> and <F1> moves the cursor one word to the left. Pressing <SHIFT> and <F6> deletes one word to the right of the cursor.

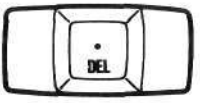
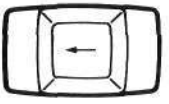
The bottom row
(identified by the word "alt") shows the effect of holding down the <ALT> key and then pressing the appropriate function key.

For example, using <ALT> and <F1> moves the cursor to the start of the current line. <ALT> and <F6> deletes text on the line to the right of the cursor.

The best way to become familiar with these keys is to type in a simple procedure and then try using them. Once you have practised and have a basic understanding of their use, you will find the screen text a useful quick reference.

Editing with Numeric Keys

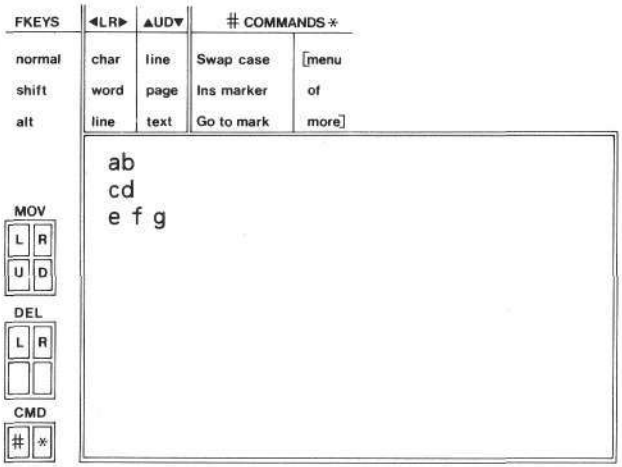
The numeric keypad on the right of the keyboard can also be used in editing. It is quite easy to anticipate what happens but here is a table of the keys and their actions:

Editing Action	*Key*
Move cursor one character to left	
Move cursor one character to right	
Move cursor up one line	
Move cursor down one line	
Move cursor to beginning of text	
Move cursor to end of text	
Move cursor up one page	
Move cursor down one page	
Delete character under cursor	
Delete character to left of cursor	

Editing a List

One helpful feature of the RM Logo editor is the way it can be used to edit lists. If you want to edit a list, use the primitive `edlist` and follow it with the name or actual list. The entries in the list will then be shown in the editor window in the following way:

```
1: make 'newlist edlist [ab cd [e f g]]
```



After amending the list press <ESC> to leave the editor. The list can be printed on screen in its amended form. For example:

```
1: print :newlist
[[ab] [cd] [e f g h]]
```

Leaving an Edit

You can get out of the editor and preserve your procedure by pressing <ESC>. Alternatively, you can leave the editor and destroy what you typed by pressing <F10> and then <A>. The procedure or list will still exist as it was before the editing.

Errors In Your Editing

Certain errors (notably unmatched brackets) have to be corrected before you can exit from the editor using <ESC>. In this case, an error message will appear on the bottom line of the screen and you won't be able to leave the editor (other than by using the <F10> and <A> keys) until you find the error and correct it.

It is also worth pointing out that you mustn't break up Logo instructions that need to be on one line. For example, the `branch...case` command must be on one line. Long lines are shown in this book by indenting the continuation lines. For example:

```
branch :x>0 [result 'positive] case :x=0
       [result 'zero] default [result 'negative]
```

should be typed in on one line without a carriage return:

If you keep typing into the editor window, text moves out of the window on the left to allow you to see that you are continuing to type on the same line.

Summary of Primitives

`build`
: Invokes the editor

`edit`
: Invokes the editor

`editlist`
: Invokes the editor to edit a list and returns the list in the form it was given

`edlist`
: Invokes the editor to edit a list and returns the list as a list of lists.

Chapter 5
Changing the Flow of Control

The simplest programs "start at the beginning, go to the end and then stop" — their instructions are carried out one after the other and they may be said to have a *linear flow of control.* For example, the following simple program gives the eight steps necessary to draw a square.

```
1: forward 50
1: left 90
1: forward 50
1: left 90
1: forward 50
1: left 90
```

There are three structures which enable you to write complex programs without having to itemise every step.

Logo allows you to write just one instruction, telling the computer to do the *repetition*, either for a given number of times, or forever.

You can tell the computer to carry out instructions only if certain *conditions* apply, and to carry out other instructions if not.

You can also build procedures that call themselves, probably using conditions to test whether they should stop. This is called *recursion.*

These three ways of determining the order that instructions are obeyed enable you to write powerful programs quite simply. You define the logical sequence of instructions and then tell the computer to do the work.

Repetition

You can repeat a list of commands a number of times using the `repeat` primitive. For example:

```
1: build 'square

square
repeat 4 [forward 50 left 90]

1: build 'spinning.squares

spinning.squares
repeat 6 [left 60 square]
```

The `repeat` lines tell Logo to obey the list of primitives inside the [ ] brackets 4 and 6 times, respectively.

If you want to, you may separate commands by using `and`. For example, you could change `spinning.squares` to the following:

```
1: edit 'spinning.squares

spinning.squares
repeat 6 [left 60 and square]
```

Using `and` also gives you fuller and more helpful error messages if you make a mistake.

If you aren't sure how many times you want a list of commands to be repeated, you can use `forever`:

```
1: build 'polygon

polygon 'angle 'side
forever [forward :side left :angle]
```

This will repeat the commands inside the [ ] brackets indefinitely and draw a closed shape. You can stop the turtle drawing by pressing <ESC>. The type of figure is determined by `:angle`; for example, a value of 90 draws a square, a value of 60 draws a hexagon. The size of the figure depends upon `:side`.

Using Conditionals

We use the word *if* in everyday speech. For example:

> 'If we have some eggs, I'll make an omelette.'
> 'If Father Christmas comes down the chimney, the trip wires will get him!'

Sometimes, it appears with *otherwise*:

> 'If there's anything decent on television, I'll watch it, otherwise I'll switch it off.'

Each of these sentences starts with a *test* or *condition* which is either true or false. After that, there is an *action* which will be taken if the condition is true. In the last case, there are two actions: one is taken if the condition is true, the other if the condition is false.

Logo has an `if` statement, as well, and it is called a *conditional*. For example:

```
1: build 'compass

compass
if heading = 0 [say 'north]
if heading = 90 [say 'east]
if heading = 180 [say 'south]
if heading = 270 [say 'west]
if heading = 360 [say 'north]

1: repeat 360 [right 1 compass]
```

This prints out the main compass points as the turtle turns. Each `if` statement tests for a condition (for example, `heading = 0`) and takes an action if the condition is true.

In the example, there is only one action and it is held within the [ ] brackets. You could, if you wanted, have two sets of [ ] brackets and this would correspond to the second form of *if*: (if...*action1*, otherwise...*action2*).

For example:

```
1: build 'sign

sign 'number
if :number < 0 [result 'negative]
       [result 'positive]
```

Here, if the condition `:number < 0` is true, the contents of the first [ ] brackets are obeyed; if it is false, Logo obeys the contents of the second [] brackets. For example, try running `sign 14` and `sign -20`.

Two other conditionals similar to `if` are available in RM Logo: `unless` and `branch`.

`unless` executes a command unless a condition is true.

`branch` is a little more complex and is described in the Reference part of this book.

There are also two other conditionals which are similar to `repeat` and `forever`: `do...until` and `do...while`.

`do...until` repeats a command list until a specified condition is true. The commands are run one or more times.

`do...while` repeats a command list as long as a specified condition is true. The commands are run zero or more times.

One command that sometimes gets you out of a problem is `goto...` Use it when you need to escape from your procedure.

```
if not assertedq :p :v [goto 'trouble]
.
.
trouble:—say [sorry I can/'t help] stop
```

`trouble:—` is a *tag* and the `goto` primitive passes control to the command after this tag. `goto` only works within a procedure. If you want to pass control outside of the procedure you will need to use `throw`.

Try not use `goto` frequently. Structured programs are clear, efficient and easily checked, so it is usually better to use procedures.

RM Logo gives you a wide variety of tests in the infix form (for example, = and <) and also the prefix form (for example, `equalq` and `lessq`). They are described in detail in the Reference part of this book.

Recursion

Recursion is another way of repeating some actions when you don't know how many repetitions are needed. It involves a procedure calling itself. For example:

```
1: build 'spiral.square

spiral.square 'side
forward :side
right 90
spiral.square :side + 2
```

If you type something like:

```
1: spiral.square 10
```

this will draw lines of increasing size, producing a square pattern. The last line is called the *recursive line* or the *recursant*.

You can stop Logo drawing lines by pressing <ESC>, but it is better if your procedure stops itself using `if`:

```
1: edit 'spiral.square

spiral.square 'side
if :side > 150 [stop]
forward :side
right 90
spiral.square :side + 10
```

Below is an example of recursion using numbers:

```
1: build 'countdown

countdown 'number
say :number
countdown :number -1
```

When you run `countdown`, the following happens (very quickly):

```
1: countdown 4
4
3
2
1
0
-1
-2
.
.
```

and so on, until you press <ESC>.

When you type in:

```
1: countdown 4
```

the first line of `countdown` prints `4` and the next line is then executed. This is effectively:

```
countdown 3
```

and it has the same effect as if you had typed it in from the keyboard: it prints `3` and then tells Logo to do the following:

```
countdown 2
```

This carries on until you press <ESC>. If you want to stop `countdown` when it reaches zero, you can do so with the following modification:

```
1: edit 'countdown

countdown 'number
say :number
if :number = 0 [stop]
countdown :number -1
```

After running `countdown` it will end on the screen with:

```
2
1
0
1:
```

The `1:` is the Logo screen prompt of course.

Here is another numeric problem that needs a recursive procedure. It involves the addition of several consecutive numbers:

```
1 + 2 = 3
1 + 2 + 3 = 6
1 + 2 + 3 + 4 = 10
.
.
.
1 + 2 + 3 + 4 + ... + n = ...
```

In each of these cases, the addition is the same as the one on the previous line, but one more number is added. In other words, the sum of all the numbers up to and including n is the sum of numbers up to and including (n–1) plus n.

This can be done in Logo with the following recursive procedures:

```
1: build 'sum.n

sum.n 'n
if :n = 1 [result 1]
result :n + (sum.n :n-1)
```

or

```
sum.n :n
result if :n = 1 [1] [:n + (sum.n :n-1)]
```

When run, the procedures produce the same result. For example:

```
1: say sum.n 4
10
```

Throwing and Catching Control

Control has been shown to move through procedures. Once the actions in a procedure have all been completed, control either moves to the next procedure or stops if there are no more.

`throw` and `catch` are a way of conditionally transferring control from a procedure back to another which has been marked to receive control. The procedure that throws control need not have been completed. The following procedures demonstrate this happening.

```
1: build 'top.prog

top.prog
catch 'rock [mid.prog] ;top.prog catches control
say 'done

1: build 'mid.prog

mid.prog
say 'all
bottom.prog
say 'lost

1: build 'bottom.prog

bottom.prog
say 'is
throw 'rock ;bottom.prog throws control
say 'completely
```

The program is run by typing `top.prog` and returns:

```
all
is
done
```

`rock` is a *signal* which `bottom.prog` throws to the procedure that called `bottom.prog`. If the calling procedure can't deal with the signal `rock`, it throws it to the procedure that called it. This action repeats until either the signal is caught or the program ends. Throwing a signal which is not caught does not give an error.

The RM Logo system throws signals that you can catch in your programs. The two most important of are:

`'error`
: throws a signal on encountering an error. `'a + 'b` is an illegal sum but `catch 'error [print 'a + 'b]` will print nothing.

`'fence`
: If you have fenced the drawing area of the turtle then a signal is thrown if the turtle crosses the edge.

More information on handling errors with `catch` and `throw` is given in chapter 12. Other system commands throwing signals are included in the reference section.

Summary of Primitives

`await`
: Suspends calling process until a condition is `'true`

`begin`
: Runs a command in parallel with current process

`catch`
: Accepts control from a `throw`

`do...until`
: Runs a list of commands until condition is `true`

`eequalq (==)`
: Tests if inputs are exactly equal

`equalq (eqq,=)`
: Tests if inputs are equal

`escape`
: Stops current procedure and any calling procedure

`end`
: Stops current procedure

`forever`
: Repeats a command forever

`goto`
: Jumps to a given label

`greaterequalq (geq, >=)`
: Tests if first input is greater than or equal to the second

`greaterq (grq, >)`
: Tests if first input is greater than the second

`if`
: Executes one of two commands depending upon the state of a condition

`lessequalq (leq,<=)`
: Tests if first input is less than or equal to the second

`lessq (lsq,<)`
: Tests if first input is less than the second

`parallel`
: Suspends calling process and runs a command

`repeat`
: Repeats a command a number of times

`result`
: Returns result of a procedure

`run`
: Runs a list of commands

`stop`
: Stops current procedure and returns to calling procedure

`throw`
: Passes control to the appropriate `catch`

`unequalq (ueq,~=)`
: Tests if inputs are unequal

`unless`
: Executes a command unless a condition is `'true`

`whenever`
: Evaluates a test repeatedly and runs a command when it is `'true`. Used in parallel processing

`while`
: Executes a command repeatedly when a test returns true

Chapter 6
Managing Your Workspace

Your computer's memory is called its *workspace* and is used to hold procedures and variables. You can print the contents of the workspace, delete them and modify them using primitives described in the first section of this chapter.

When you switch off your computer, the contents of the workspace are lost, so you need to copy them on to disk if you want to use them again without having to type them in again. The way you do this is described in the second section.

The third section shows how you can 'capture' a sequence of commands typed in and replay them at any time.

Finally, the last section describes a number of file maintenance primitives.

Manipulating the Contents of your Workspace

You can display the contents of your workspace on screen using `titles`. For example:

```
1: say titles
square triangle rhombus
```

You can display ("print out") procedures on screen using `po`. To show the contents of one procedure for example:

```
1: po 'triangle
triangle 'steps
repeat 3 [fd :steps lt 120]
```

Or you can display the contents of several procedures together. For example:

```
1: po [triangle square]
```

Unlike `build` and `edit`, any word input to `po` must have a quotation mark.

Alternatively, `text` will return the contents of a procedure in the form of a list:

```
1: text 'triangle
```

returns:

```
[[triangle 'side] [repeat 3
        [fd :side lt 120]]]
```

If you want to delete a procedure, you can do so using `scrap`:

```
1: say titles
square triangle rhombus

1: scrap [triangle rhombus]

1: say titles
square
```

Preserving your Work on Disk

You can copy your procedures onto disk using the `save` command. They are then preserved when the computer is switched off. For example, the following command saves all your procedures in the file `shapes.lgp`:

```
1: save 'shapes.lgp
```

Now, when you switch on your computer again, you can load the procedures back into workspace by typing:

```
1: load 'shapes.lgp
```

All of the procedures in the file `shapes` will be loaded.

Replaying a Sequence of Commands

You can make a record of everything typed using the `dribble` primitive. For example, if you type:

```
1: dribble 'session
```

all subsequent commands will be written (dribbled) to the file `session`. If this file already exists, the commands will be added to the end of it.

When you type `nodribble`, dribbling will stop.

If you now want to replay the sequence of commands typed in, just type:

```
1: replay 'session
```

The replay will stop when the end of the file is reached. If an error occurs, the file will continue to be read.

If you want to ensure that the replay stops when an error occurs, you must use `consult`:

```
1: consult 'session
```

File Maintenance Operations

At some point you will want to do some or all of the following operations:

- Delete (or erase) a file
- Rename a file
- Look at the disk directory

The primitive `erasefile` allows you to do the first of these. For example:

```
1: erasefile 'shapes.lgp
```

will erase the file `shapes`, if it exists, and return the value `'true`. If not, it will return the value `'false`.

To rename a file you use `renamefile`. For example:

```
1: renamefile 'shapes.lgp 'newshapes.lgp
```

changes the name of the file `shapes` to `newshapes`.

You can return the contents of a disk directory to your program, in the form of a list, using the `directory` primitive.

`directory` takes one input in the MS-DOS format but remember that the * and : characters are special to Logo; a backslash character \ is needed to protect them. For example:

```
1: print directory '\*.lgc
or
1: print directory 'shapes.\*
or
1: print directory '\b:\*.lgp
```

Summary of Primitives

`consult`
: Replays command file and stops on error

`dribble`
: Writes all subsequent commands to a command file for replay

`dribbleq`
: Tests whether session is being recorded

`erasefile`
: Deletes a file

`keep *`
: Saves named procedures in a file

`load *`
: Loads procedures from a file

`po`
: Prints contents of procedures

`print`
: Prints text on screen including list brackets and special characters

`renamefile`
: Renames a file

`replay`
: Replays a command file and ignore errors

`save *`
: Save procedures in a file

`say`
: Print text on screen without list brackets

`scrap`
: Erase a procedure from workspace

`text`
: Return procedure in the form of a list

* These are Logo library procedures, loaded and buried when you enter Logo. See Chapter 15 for more details.

Chapter 7
Simple Input/Output

This chapter describes a number of types of simple input and output which do not fall within the scope of turtle graphics.

Printing on the Screen

You can print on the text area of the screen using the primitives `say`, `print` and `type`.

`say` prints the value input to it and follows this with a carriage return. For example:

```
1: say 1 + 3
4

1: say [hello there!]
hello there!
```

Notice that lists are printed without their outermost brackets.

`print` has a similar effect to `say`, but lists are printed with their outermost brackets. For example:

```
1: print 1 + 3
4
1: print [hello there!]
[hello there!]
```

`type` is similar to `say`, but the text output is not followed by a carriage return. For example:

```
1: type 'banana
banana1:
```

`type` is ideal for sending escape sequences because it does not automatically give a new line.

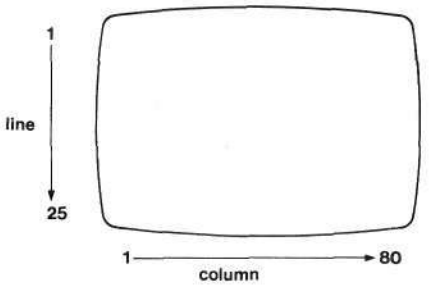
You can output text to the graphics part of the screen using `label`. This prints text at the current turtle position.

Two other primitives which can be used to effect with `say`, `print` and `type` are `setcursor` and `cursor`.

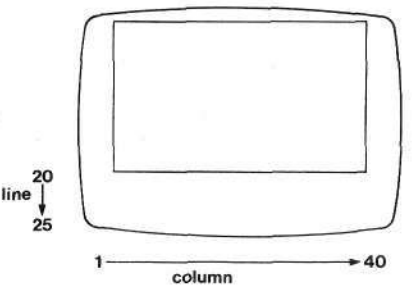
`setcursor` moves the text cursor to a given position and `cursor` returns its position in the form of a list. For example,

```
setcursor [22 16]
```

moves the text cursor to line 22 and column 16. Text mode has a different arrangement to graphics mode:



text mode



graphics mode

Input from the Keyboard

The primitive **key** will suspend the process that called it until you strike a key. It will then return the value of the key struck. Digits are returned as a Logo number, other characters are returned as one-character words. If Logo is reading a command file, **key** still returns the next character from the keyboard.

The following procedure uses **key**. It lets you control the turtle's movement using only five keys:

```
1: build 'move

move
check.key key
move

1: build check.key

check.key 'button
if :button = 'l [left 10]
if :button = 'r [right 10]
if :button = 'f [forward 10]
if :button = 'b [back 10]
if :button = 'c [clearscreen]

1: check.key
```

Leave the procedure by pressing the <ESC> key.

If you want, you can make the turtle move forward continuously and just use the `l` and `r` keys to change its direction. To do this, you use the primitive **keyq**. **keyq** returns the value `'true` if a key has been struck and lets you read the key's value using **key**.

The new procedures could look like this:

```
1: edit 'move

move
if keyq [check.key key]
forward 0.01
move

1: edit check.key

check.key :button
if :button = 'l [left 10]
if :button = 'r [right 10]
```

Summary of Primitives

`cursor`
: Returns position of cursor in form of list

`key`
: Suspends calling process until key is struck and returns key value

`keyq`
: Returns `'true` if key is struck but does not suspend calling process

`print`
: Prints text with list brackets

`say`
: Prints text without list brackets

`setcursor`
: Moves cursor to specific position

`type`
: Prints text but does not send final carriage return

Chapter 8
Arithmetic

Logo handles numbers in a variety of forms. It also allows you to perform arithmetic operations in your programs.

Positive and Negative Numbers

A Logo number consists of one or more numerals and it can contain a minus sign (-). However, if you have a unary minus that could be interpreted as a binary minus then it must be appropriately bracketed. For example:

```
1: print add 10 -20

Logo can't do "add" in that command because "add"
needs more inputs
1: print add -10 20
10
1: print add 20 (-10)
10
```

This does not apply inside a list. For example you can use `setpos [50 -50]` and it will work.

Numbers can be separated from other items on a line by using spaces:

```
1: say 200
200
1: say 100 -200
-100
```

Numbers can be integers or decimal numbers and they are printed with a precision of up to 15 decimal places.

Arithmetic Operators

The Logo arithmetic operators include:

*	multiply
/	divide
+	add
—	subtract
↑	power

These are known as *infix* operators, because they appear in between two numbers. For example:

```
1: say 10 * 3
30
```

Infix operators have equivalent *prefix* operators, which appear in front of two numbers. For example:

```
1: say multiply 10 3
30

1: say divide 10 2
5
```

Infix operators are associated with the appropriate prefix operators in the Reference part of this book.

Trigonometric primitives are also supplied. These include `sin` (sine), `cos` (cosine), `tan` (tangent) and `atan` (arctangent).

They take an input in *degrees*, for example

```
1: say sin 30
0.5
```

The transcendental functions `log`, `ln`, `exp` and `sqt` are also included in Logo.

`int` and `frac` are the two useful primitives that separate integer and fractional parts of a number. `int` chops the number off at the decimal point and `frac` gives the part of the number chopped off.

Random Numbers

Logo is able to return a random integer number to you in the range of 1 up to a number that you specify. This uses the `pick` command: Logo picks an integer out from the given range. For example:

```
1: repeat 20 [say pick 33]
```

will return a list of twenty integers chosen at random from the range 1 to 33.

However, if you wanted a random number in the range 0 to 1 then use the `random` primitive. This returns a number of fourteen decimal places.

Summary of Primitives

`abs`
: Absolute value

`acros`
: Arc cosine

`add (+)`
: Addition

`asin`
: Arc sine

`atan`
: Arctangent (result in degrees)

`cos`
: Cosine

`divide (div, /)`
: Division

`exp`
: Exponential function

`frac`
: Fractional part

`int`
: Integer value

`ln`
: Natural logarithm

`log`
: Logarithm to base 10

`multiply (mul, *)`
: Multiplication

`pi`
: Returns the value pi

`pick`
: Returns pseudo-random integer

`power (↑)`
: Raising one number to the power of another

`random`
: Returns random fraction between 0 and 1

`remainder (rem, %)`
: Returns remainder after a division

`share (//)`
: Returns integer quotient after division

`sin`
: Sine

`sqt`
: Square root

`subtract sub (-)`
: Subtraction

`tan`
: Tangent

Chapter 9
Words And Lists

Text manipulation is one of the significant parts of Logo. Text comes in the form of *words* and *lists* and you can produce some very sophisticated programs using the list-handling primitives. This chapter describes words and lists, together with these primitives.

Words

In spoken languages, a word is a group of letters which conveys some idea. The concept of a word in Logo is similar.

Logo words must be preceded by a quotation mark. For example, the following are all words:

```
'cats
'rats
'r2d2
'12
'1066
```

The quotation mark tells Logo that whatever follows is to be treated as a word. Quotation marks in this position are not regarded as part of the word and will not be printed by `say`:

```
1: say 'cats
cats
```

You can break words into smaller words using the primitives `first`, `last`, `butfirst`, `rest` and `butlast`.

For example:

```
1: say first 'tortoise
t

1: say last 'tortoise
e

1: say rest 'tortoise
ortoise
```

You can also join (concatenate) words using `join`:

```
1: say join 'tortoise 'shell
tortoiseshell
```

`join`, like some of the other primitives, has an infix form ++. For example:

```
1: say 'tortoise ++ 'shell
tortoiseshell
```

If you type the following:

```
1: say butfirst 'x
```

this prints a word with no letters (called the *empty word*). You can use the empty word in a command or procedure by typing a quotation mark on its own. For example:

```
1: say '
```

The following commands show how you can use the empty word to stop a procedure from running:

```
1: build 'tri.print
```

```
tri.print 'letters
if :letters = ' [stop]
say :letters
tri.print butfirst :letters

1: tri.print 'logo
logo
ogo
go
o
```

Lists

Logo's ability to combine data into structures called *lists* is very useful. A list consists of a number of *elements* separated by spaces and surrounded by a pair of square brackets. For example:

```
[1 2 3 4 5]
[cats dogs hamsters]
```

The elements of a list can be words, as shown above (notice that words do not need preceding quotes when used in a list). They can also be numbers or other lists. For example:

```
[[tortoiseshell tabby persian]
        [labrador alsatian mongrel]]
```

You can print the contents of a list using `print` and `say`. `say` will strip off the outermost brackets, `print` will not. For example:

```
1: print [kitchen dining.room stairs]
[kitchen dining.room stairs]

1: say [kitchen dining.room stairs]
kitchen dining.room stairs
```

You can break lists into smaller lists or words using the operators `first`, `last`, `butfirst` and `butlast`. For example, this command:

```
1: make 'house [[kitchen dining.room lounge]
        [stairs] [bedroom bathroom]]
```

groups the rooms of a house into upstairs rooms, downstairs rooms and the stairs. The following commands isolate each of these groups.

```
1: print first :house
[kitchen dining.room lounge]

1: print last :house
[bedroom bathroom]

1: print butfirst butlast :house
[stairs]
```

You can insert an element into a list using the primitives `putfirst` and `putlast`:

```
1: print putfirst 'study [kitchen dining.room
        lounge]
[study kitchen dining.room lounge]

1: print putlast [bedroom bathroom]
        'shower.room
[bedroom bathroom shower.room]
```

`putfirst` and `putlast` have the infix forms +> and <+ respectively. So another way of writing the two examples above would be:

```
print 'study +> [kitchen dining.room lounge]

print [bedroom bathroom] <+ 'shower.room
```

`amongq` or `memberq` will tell you if a list contains a specified element:

```
1: say amongq 'study [study kitchen
       dining.room] true
1: say amongq 'bathroom [study kitchen
       dining.room] false
```

List Pointers

is the list pointer. It enables you to look at a particular element in a list or place an element in a list.

If you create a list `pets` for example:

```
1: make 'pets [dog fish cat hamster frog]
```

you might later on want to be reminded what the third entry in the list is. This can be done by:

```
1: print :pets # 3
'cat
```

also works for identifying elements of a list in a list, such as a list of coordinate pairs for example:

```
1: make 'points [[0 0] [50 0] [50 50] [0 50]]
1: print :points #1
[0 0]
1: print :points #2 #1
50
```

Now change an entry in the list of pets. If the third entry should be spider then `pets` is amended by:

```
1: make 'pets #3 'spider
```

This makes the updated list:

```
[dog fish spider hamster dog]
```

To amend the second entry in the list of coordinate pairs you use # in a similar way.

```
1: make 'points #2 #1 32
```

The list `points` is now:

```
[[0 0] [32 0] [50 50] [0 50]]
```

Other Operations on Words and Lists

RM Logo allows you to perform other operations upon words and lists. For example:

- `editlist` will let you edit a list in the edit window and will return it in the form it was given in
- `edlist` will let you edit a list in the edit window and will return it as a list of lists
- `count` will return the number of elements in a word or list
- `emptyq` will tell you if a variable contains the empty word or the empty list
- `explode` will turn the contents of a word into a list, each element of which is a letter or number of the word
- `implode` will join the elements of a list into one word

Summary of Primitives

`amongq (memberq)`
: Tests if an item is an element of a list

`butfirst, rest (bf)`
: Returns all elements of a word or list other than the first

`butlast (bl)`
: Returns all elements of a word or list other than the last

`count`
: Returns the number of elements in a word or list

`emptyq (emq)`
: Tests for the empty list or empty word

`eval`
: Evaluates the contents of a list

`explode`
: Separates elements of word into list

`first`
: Returns first element of word/list

`implode`
: Joins elements of list into word

`#`
: Returns nth element of list

`join (++)`
: Joins two words or lists

`last`
: Returns last element of word/list

`lowercase (lcase)`
: Changes text to lower case

`putfirst (pf, +>)`
: Puts item as first element of list

`putlast (pl, <+)`
: Puts item as last element of list

`rest`
: Same as `butfirst`

`sentence (se, &&)`
: Joins two lists

`uppercase (ucase)`
: Changes text to upper case

Chapter 10
Organising Information

Introduction

RM Logo allows you to build up a filing system, or *database*, in the computer's memory and use it to store information in an ordered way: for example to hold names and addresses.

The categories which allow you to build a database in Logo are called *properties*. For example, you might have a name `'rover` with the *property* `'species` and `'species` could have the *value* `dog`. You can associate these three items by using the `assert` primitive:

```
1: assert 'rover 'species 'dog
```

You could give a number of other names the property `'species'` in a similar way:

```
1: assert 'whiskers 'species 'cat
1: assert 'patch 'species 'dog
1: assert 'joey 'species 'parrot
1: assert 'fido 'species 'dog
1: assert 'scotty 'species 'dog
```

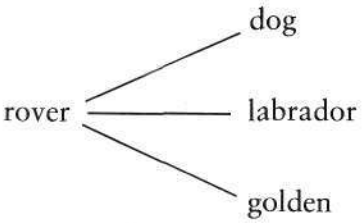
You can now build up your filing system by giving a name a number of other properties:

```
1: assert 'rover 'breed 'labrador
1: assert 'rover 'colour 'golden
1: assert 'patch 'breed 'dalmatian
1: assert 'patch 'colour [black and white]
```

If you want to, you can now examine the property values using the primitives `asserted` and `assertions`. `asserted` returns one property value of a name:

```
1: say asserted 'rover 'breed
labrador
1: say asserted 'patch 'breed
dalmatian
```

Up to now, you have set up a number of assertions which you can picture as linked to each name:



You can get hold of all the property values of a name (the categories it belongs in) using the primitive `assertions`:

```
1: say assertions 'rover
[species dog] [breed labrador] [colour golden]
```

You can remove a property and its value using `deny`:

```
1:deny 'fido 'species
1:deny fido 'type
```

You can find out what names the database knows about by using the `objects` primitive. For example:

```
1: say objects
```

might produce the following names:

```
[rover fido whiskers patch joey scotty]
```

A Simple Database

Let's now develop a procedure `del.props` that will allow you to delete all the properties of a name. The procedure can find the list of property values using `assertions` and then pass this list on to another procedure, `del.props.of`. This in turn will work through the list and use `deny` to delete each of the properties in turn.

A procedure `del.props` is needed which will be called as though you typed:

```
del.props 'rover
```

Another procedure, `del.props.of` is needed to be called as though you typed:

```
del.props.of 'rover [[species dog]
        [breed labrador] [colour golden]]
```

Before doing anything else, `del.props.of` should ask whether the list of assertions is empty; if so, it can stop. From this, it can be worked out that the first two lines of `del.props.of` must be:

```
del.props.of :name :assns
if emptyq :assns [stop]
```

Otherwise, the `first` element (the property name) in the `first` property pair of `:assns` needs to be looked at. Specifically, this is `species`, and it is all that is needed to deny the property pair.

The remaining properties can be deleted using `del.props.of` again, with the `rest` of `assns` as input.

The complete procedures are listed below:

```
1: build 'del.props

del.props 'name
del.props.of :name assertions :name

1: build del.props.of

del.props.of 'name 'assns
if emptyq :assns [stop]
deny :name first first :assns
del.props.of :name rest :assns
```

All you need do is call `del.props` with the appropriate name as input. For example:

```
1: del.props 'fido
```

will delete all properties and their values for the name `'fido`.

Retrieving Information

Now that the rudiments of a database have been built up, you can look at it in more interesting ways and manipulate it. For example, you could find out which names have the property `'species` using the primitive `classified`:

```
1: print classified 'species
[rover whiskers patch joey scotty]
```

It would be useful if you could also find out which of the above names belong to the species `'dog`. Lets develop a procedure `get.details` to do this. The procedure will be called as though you type something like the following:

```
get.details 'species 'dog
```

It will first need to use `classified` to get a list of names which have the property `species`. It will then pass on this list, together with the property value `dog`, to a procedure `scan.list`.

`scan.list` should first look at the list and check that it is not empty. If it is, then no names satisfy the condition and `scan.list` can return the empty list as its result. The procedures so far are as follows:

```
1: build 'get.details

get.details 'property 'propvalue
result scan.list classifield :property
        :property :propvalue

1: build 'scan.list

scan.list 'names 'property 'propvalue
if emptyq :names [result []]
.
.
.
```

Now the first name can be looked up in the database using `asserted` to find the property value:

```
asserted first :names :property
```

If the value of this is `dog`, it is included in the list returned by `scan.list` as its result. The following (incomplete) expression now exists:

```
if (asserted first :objects :property) = :propvalue
        [result putfirst (first :names) scan.list rest
        :names :property :propvalue]
```

If the property value isn't one of interest, the name needs to be ignored. However, there may be other names in the list that are of interest.

So one last statement is added:

```
[result scan.list rest :names :property
        :propvalue]
```

This is the final line of `scan.list`, and the procedures are now as follows:

```
1: build 'get.details

get.details 'property 'propvalue
result scan.list classifield :property
        :property :propvalue

1: build 'scan.list

scan.list 'names 'property 'propvalue
if emptyq :names [result []]
if (asserted first :objects :property) =
        :propvalue [result putfirst (first :names)
        scan.list rest :names :property :propvalue]
        [result scan.list rest :names :property
        :propvalue]
```

Building a more Sophisticated Database

The rest of this section is about developing a database containing names and details of people. Some procedures will be introduced to help you to manipulate the database in a more flexible way.

First think of categories of information (properties) you might want to record for each person. It might include their name, address, telephone number and interests. The facilities already described allow you to create a database containing this information and manipulate it in a simple way. However, it would be useful if you could input all of the properties for a given name at the same time, instead of doing them one at a time using `assert`.

The following procedure, `create.props`, will do this for you:

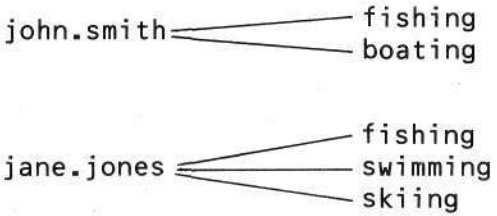
```
1: build 'create.props

create.props :name :list
if :list = [] [stop]
assert :name (first :list) (first rest :list)
create.props :name rest rest :list
```

This is how you use it:

```
1: create.props 'John.Smith [address
    [33 Tin Pan Alley, Newtown] Telephone [0222 55555]
    interests [fishing boating]]

1: create.props 'Jane.Jones [address
    [Hawthorns, Billingsbrooke] Telephone [0111 59555]
    interests [fishing swimming skiing]]
```

Notice how "telephone" has a one-to-one relationship with the telephone number:

"Interests" however, is a one-to-many relationship:



All of one person's interests are stored as a single Logo

list.
This is equivalent to typing:

```
assert  'john.smith  'interests
        [fishing boating]
assert  'jane.jones  'interests
        [fishing swimming skiing]
```

To find out whether John Smith is interested in chess you could use:

```
1: amongq 'chess asserted 'john.smith 'interests
```

Another question you might want to answer is "who is interested in what?", meaning who is interested in fishing, swimming or skiing? The primitive `objects` is an excellent starting point as it returns a list of every object in the database. Try typing:

```
1: print objects
```

The following program `scan` allows you to find an answer to the question "who has the property with this property value(s)?". For example:

```
1: say scan  'interests  'fishing
```

returns a list of those names whose property values include fishing.

```
1: build  'scan

scan  'property  'value
result scan1 objects :property :value
```

```
1: build 'scan1

scan 'objs 'property 'value
if emq :objs [result []]
if assertedq first :objs :property &
        amongq :value asserted first :objs
        :property [result pf first :objs scan1
        rest :objs :property :value]
result scan1 rest :objs :property :value
```

Two other facilities would be useful to know:

- Adding items to property values (for example, to add new `interests` in the example above
- Deleting several properties at the same time

If you want to add items to property values, you can do so using the following procedure:

```
build 'add.props

add.props 'name 'property 'object
if not assertedq :name :property [assert :name
        :property :object stop]
assert :name :property sentence (asserted :name
        :property) (:object)
```

For example, the following command will add two new interests to John Smith's entry:

```
1: add.props 'John.Smith 'interests [skating mabel]
```

You can delete several properties at once using the following procedure:

```
1: build 'delete.props

delete.props 'name 'object
if :object = [] [stop]
if wordq :object [deny :name :object stop]
deny :name first :object
delete.props :name butfirst :object
```

For example, the following will delete Jane Jone's telephone number and interests:

```
1: delete.props 'Jane.Jones [telephone interests]
```

You can then type in new ones using `create.props`.

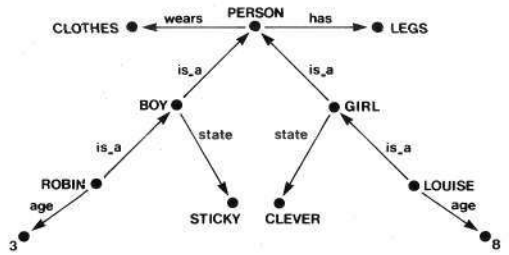
There are many other ways in which you can improve upon this database and its methods of access. For example, you could try modifying `check.props` and `scan.list` to allow you to search the database for people with two common interests, instead of just one.

Reasoning by Inference

Think about this database:

```
1: assert 'Robin 'age 3
1: assert 'Robin 'is_a 'boy

1: assert 'Louise 'age 8
1: assert 'Louise 'is_a 'girl

1: assert 'boy 'state 'sticky
1: assert 'girl 'state 'clever
1: assert 'boy 'is_a 'person
1: assert 'girl 'is_a 'person

1: assert 'person 'wears 'clothes
1: assert 'person 'has 'legs
```

It can be represented diagrammatically as follows:



You can now write a procedure `infer` which will try to find the value of a specific property for a given object. If it can't, it will use the property `is_a` to make an inference, or deduction:

```
1: build 'infer

infer 'object 'property
if assertedq :object :property [result asserted
        :object :property]
if assertedq :object 'is_a [result infer
        (asserted :object 'is_a) :property]
```

For example:

```
1: say infer 'Louise 'age
8

1: say infer 'Robin 'state
sticky

1: say infer 'Louise 'has
legs

1: say infer 'Robin 'wears
clothes
```

The first line of `infer` tries to get the value of the appropriate property for a given name. If it can't, it searches 'backwards' up the tree along the `is_a` property route to try and find the property at a higher level.

Summary of Primitives

`assert`
: Assigns a property value

`asserted`
: Returns property value of a name

`assertedq`
: Tests for existence of a property

`assertions`
: Returns list of property pairs

`classified`
: Returns list of names which have a given property

`deny`
: Deletes a property and its value

`objects`
: Returns a list of all the values that are associated with properties

Chapter 11
File Handling

Disks and Files

Disks are one of the most versatile ways of storing computer data. They are compact and can hold a large amount of information which can be retrieved quickly and easily. RM Nimbus can use 3.5-inch disks or winchester disks as part of a standalone or network Nimbus.

Disk file storage is similar to the storage method used in libraries. Each book in a library contains *information* and it has an entry in a central index. This index is a type of *directory* which tells you where to look for books.

You can think of disks as containing *files* which correspond to the books in a library. Each file has a name, usually one to eight letters, a dot and three letters. Each file can hold information such as text, programs and raw data. For each file there is an entry in a *directory* which holds, for example, the *filename* and the position of the file on the disk.

The analogy can be taken further. To find a book in the library, you look at the index (directory) to find out where the book is. You then take the book out, *read* its contents and, finally, *close* it and put it back.

With a disk filing system, Logo would look at the disk directory to see where the file is and to *open* it. Your program would then *read* information from the file and, finally, *close* it. The action of opening the file is like looking in the library's index.

When a librarian wants to put a new book into the library, an entry is created for it in the index which puts the

book into its correct position in the library. With disks, the procedure is similar, but this time the actions are carried out by your program and Logo. When your program asks Logo to *create* a new file, Logo creates a new entry for the file in the disk's directory, specifying the filename. Your program can then *write* records to the file and *close* it.

A book can be put in the library, but no-one else will know where it is until the library's index has been updated. Similarly with disk files, Logo will not know where the file is on the disk until the file has been written and closed, at which point the directory is updated.

To summarise, a disk is organised into a number of files and these are indexed by a short directory. When your program creates, or makes, a file three things happen:

- a new directory entry for the file is created
- information is written to the file
- the file is closed

When your program reads a file, again three things happen:

- the file is opened (after looking in the directory to find out where the file exists on the disk)
- information is read from the file
- the file is closed

The primitives involved in these actions are:

Opening files: `infile outfile appfile`
Closing files: `closefile`
Reading an item: `readfiled, readfilec, readfilel`
Writing an item: `writefiled, writefilec, writefilel`

Creating a Simple File

Suppose you want to create a file which contains details of the people in a company. You might want to store the following lines as separate records, for example:

```
1 Taylor Judy 44
2 Maslin Roger 20
10 Smith Tina 20
11 Jenkins Tom 30
100 Watson Philip 30
        .
        .
```

The first entry on each line is the 'personal number', the second the name, and the last entry, the age. You might allocate a particular range of personal numbers for each department.

For example:

Range	Department
1–9	Secretaries
10–100	Sales Staff
100–1000	Programmers

If you look at the entries shown, you will see that there is a jump from personal number 2 to 10 and from 11 to 100. This is to allow for the addition of staff at a later date.

These entries can be written to a file using the program `write.names.to.file`. This reads each line as a list from the keyboard and writes it to a named disk file until you press ~ followed by <ENTER>.

```
1: build 'write.names.to.file

write.names.to.file 'filename
unless outfile :filename [say
        [cannot create file] escape]
unless (write.file :filename readlist) &
        (closefile :filename) [say [write error]
        escape]

1: build 'write.file

write.file 'filename 'record
if :record = [*] [result 'true]
unless writefiled :record :filename
        [result 'false]
result write.file :filename readlist
```

Let's look at these procedures a little more closely. `write.names.to.file` creates an *output file* (using the primitive `outfile`) and you specify the name of the output file as input to `write.names.to.file` itself. The procedure `write.file` is then called to write the records to the file.

`write.file` uses the primitive `writefiled` to write each record to the file in the form of a data item which you type in at the keyboard (using `readlist`). `writefiled` returns the value `'true` if the operation was successful and `'false` if it was not. There are four main reasons why it may not have been successful:

- The disk may be full (this corresponds to the shelves being full in our library analogy)
- The directory may be full (this corresponds to a full central index in the analogy)
- The disk may be protected via a write protect notch
- The disk is damaged

`write.file` is recursive: it keeps running until you type in ~ at the keyboard, at which point `write.names.to.file` closes the file using the primitive `closefile`. `closefile` also returns `'true` if the operation was successful and `'false` if it was not.

You can run the program and write records to the file `names` as follows:

```
1: write.names.to.file 'names
? 1 [Taylor Judy] 44
? 2 [Maslin Roger] 20
? 10 [Smith Tina] 20
? 11 [Jenkins Tom] 30
? 100 [Watson Philip] 30
? *
```

Reading a Simple File

If you now want to read data back from the file `names` and print it on the screen, you can do so using the program given below:

```
1: build 'read.names.from.file

read.names.from.file 'filename
unless infile :filename [say
        [file does not exist] escape]
catch 'endfile [readfile :filename]
unless closefile :filename
        [say [read error] escape]

1: build 'readfile

readfile 'filename
forever [say readfiled :filename]
```

The first command of `read.names.from.file` is analogous to the first command of `write.names.to.file`

in that it opens a file for input (using `infile`).
The next command reads records from it using the procedure `readfile`. Ignore the last command for the present and look at `readfile` first.

`readfile` uses the primitive `readfiled` to read each record from the file in the form of a data item and then prints it on your screen. `readfile` runs continuously until the end of the file is reached. When this happens, `readfiled` generates a `throw 'endfile` and this is "caught" by the `catch 'endfile` in `read.names.from.file`. The file is then closed (using the primitive `closefile`).

You can run this program and read back records from the file `names` by typing:

```
1: read.names.from.file 'names
1 [Taylor Judy] 44
2 [Maslin Roger] 20
10 [Smith Tina] 20
11 [Jenkins Tom] 30
100 [Watson Philip] 30
```

Changing Data in a File

You can change data in an existing file in three ways: you can *append* data to it, you can *delete* data in it, or you can *replace* existing data with new data.

Appending data to an existing file is simple: the method is the same as for creating a new file, but instead of using `outfile` to open the file you use the primitive `appfile`.

The other two operations are a little more complex because you can't modify the existing file: instead, you must read

its contents and write them, selectively, to a new output file. The delete operation first is looked at first.

If you want to delete records in an arbitrary way, you would give a list of the numbers to be deleted as an input to `delete.records`

```
1: build delete.records.from.file

delete.records.from.file 'filename 'deletions
unless outfile 'tempfile [say [cannot open
        temporary file] escape]
unless infile :filename [say [cannot find]
        <+ :filename] escape]
delete.records :filename :deletions (rfd
        :filename)
unless closefile 'tempfile [say [cannot
        close temporary file] escape]
unless closefile :filename [say [cannot
        close] <+ :filename] escape]
unless erasefile :filename [say [cannot
        erase <+ :filename] escape]
unless renamefile 'tempfile :filename [say
        [cannot rename temporary file] escape]

1: build 'delete.records

delete.records 'filename 'deletions 'record
if 'record = 'endfile [stop]
unless amongq first :record :deletions
        [unless writefiledata :record
        :filename [say [write failed] escape]]
delete.records :filename :deletions
        (rfd :filename)
```

The last amendment to try is adding records to the file. There are different ways to do this, but the most efficient is to put the records in ascending order into a new file and then to process this file against the input file.

Let's call the file of additional records `insertions`, the input file `names`, and create a procedure `add.records` to read the two files and merge their contents into one file in the correct order.

The procedure `add.records` would be called by:

```
1: add.records 'names 'insertions
```

The actual procedures would be:

```
1: build 'add.records

add.records 'insertions 'mainfile
unless outfile 'tempfile [say [cannot
        create temporary file] escape]
unless infile :insertions [say [cannot
        open] <+ :insertions escape]
unless infile :mainfile [say [cannot
        open] <+ :mainfile escape]
merge :insertions :mainfile (rfd
        :insertions) (rfd :mainfile)
unless closefile :insertions & closefile
        :mainfile [say [read error] escape]
unless closefile 'tempfile [say [cannot
        close temporary file] escape]
unless erasefile :mainfile [say [cannot
        erase] <+ :mainfile] escape]
unless renamefile 'tempfile :mainfile [say
        [cannot rename temporary file] escape]
```

```
1: build 'merge

merge 'insertions 'mainfile 'ins_record
        'main_record
if :ins_record = 'endfile & :main_record =
        'endfile [stop]
if :ins_record = 'endfile | first :main_record
        < first :ins_record
[if wfd :main_record 'tempfile
[merge :insertions :mainfile :ins_record
(rfd :mainfile)] [say [write error] escape]]
if :main_record = 'endfile | first
        :main_record > first :ins_record
        [if wfd :ins_record 'tempfile [merge
        :insertions :mainfile (rfd :ins_record)
        :main_record] [say [write error] escape]]
say [personal number is duplicated: writing
        new record]
if wfd :ins_record 'tempfile
        [merge :insertions :mainfile (rfd :ins_record)
        (rfd :main_record)] [say [write error] escape]
```

A Few Last Words on Files

This chapter is intended to give you an introduction to some of the Logo features. It has not covered all of the primitives available for file handling; for example, you can write characters to a file, instead of lists, and read them back in the same way. The full range of file handling facilities are covered in the list of primitives at the end of this chapter.

The library file `listfile.def`, provided on your RM Logo disk, lists the contents of any file using the primitive `readfilec`.

File Names

You identify a file using the file name and an extension which describes the nature of the file, for example: `names.lgd`. You can also specify the disk drive code, in which case you must follow it with a colon, for example: `a\:names.dat`. The colon needs to be prefixed with \ because it is a Logo special character.

The filename consists of one to eight alphanumeric characters (A to Z and 0 to 9) but the first character must be alphabetic.

The extension consists of three alphanumeric characters. You can use any combination you want. These are usual in RM Logo:

`lgc`
Logo commands (to be read by `consult` or `replay`)
`lgd`
Logo data
`lgo`
Logo special (news file)
`lgp`
Logo procedures
`lgx`
Logo machine code extension
`def`
Logo procedures provided as library definitions

Using Temporary Files

The use of temporary files when amending a data file is recommended!

In practice, you should not use temporary files with fixed names. A better way to name them is to follow the procedure whereby the latest copy of a file has the extension `dat` for example, and the previous copy is kept as backup, with the extension `bak`.

When you process the file, your program takes the following actions:

- If the input file is called `names.dat`, call the temporary file `names.$$$`
- At the end of processing the files, delete any file called `names.bak` (backup files should always have the extension `bak` as many text editors expect it)
- Next, rename `names.dat` to `names.bak`, thereby making the "old" file the backup file
- Finally, rename `names.$$$` to `names.dat`

This means that you will need to be able to separate out the filename and extension when you call a procedure and the filename is given as input. For example:

```
1: write.names.to.file 'names.dat
```

In conclusion, you need these two procedures to allow you to do this:

```
1: build 'file.name

file.name 'name
if emq :name | first :name = '.
       (result '') (result first :name ++
       file.name rest :name)

1: build 'change.extension

change.extension 'original.name
       'new.extension
result renamefile :original.name
       (file.name :original.name) ++
       :new.extension
```

You can now change the name of the file `names.dat` to `names.bak` and return `'true` by typing:

```
1: change.extension 'names.dat '.bak
```

Sorting out Disk Problems

If, for example, you mess up your disk by taking it out of the drive while Logo has a file open for writing, you can sort out the problem using the MS-DOS utility `chkdsk`.

Please see the Nimbus Owners Handbook for details of this utility.

Summary of Primitives

`appfile`
: Opens current file for appending

`closefile`
: Closes current file

`consult`
: Replays commands from a file

`directory`
: Returns directory information as list

`dribble`
: Writes subsequent commands to a file

`dribbleq`
: Checks if command dribbling is on

`erasefile`
: Erases a file

`infile`
: Opens a file for input

`infiles`
: Returns names of files open for input

`nodribble`
: Turns off dribbling

`outfile`
: Opens a file for output

`outfiles`
: Returns names of files open for output

`readfilec, rfc`
: Reads a character from a disk file

`readfiled, rfd`
: Reads a data item from a disk file

`readfilel, rfl`
: Reads a list from a disk file

`renamefile`
: Renames a file

`replay`
: Replays a sequence of commands from a file

`writefilec, wfc`
: Writes a character to a disk file

`writefiled, wfd`
: Writes a data item to a disk file

`writefilel, wfl`
: Writes a list to a disk file

Chapter 12
Error Handling and Debugging

The first section of this chapter describes how to handle errors, both within your programs and from command mode. The second section describes what to do when your programs don't work.

Error Handling

Handling Keyboard Mistakes

When Logo can't do what you want it to, it prints a message on your screen. Try typing the following, for example:

```
1: print ad 1 2

Logo can't do "ad" in that command
because "ad" does not exist
1:
```

If you spot an error before you press <ENTER> you can correct it by using <BACKSPACE>.

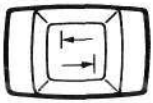
Other keys which will help you when inputting from the keyboard are summarised as follows:

<CTRL/G>
Erase everything on line

<CTRL/R>
Repeat last line input

<LEFT>
Move cursor one character left

<RIGHT>
Move cursor one character right



<TAB>
Move cursor right by one word

<SHIFT/TAB>
Move cursor left by one word

<F1>
Move cursor to start of command

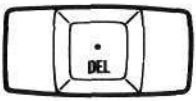
<F2>
Move cursor to end of command

<F3>
Erase word to left of cursor

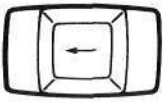
<F4>
Erase word to right of cursor

<F5>
Erase everything to left of cursor

<F6>
Erase everything to right of cursor



<DEL>
Erase character under cursor



<BACKSPACE>
Erase character to left of cursor

Handling Errors in your Program

The `catch` and `throw` primitives let you trap errors in your programs. They can be used to trap events other than errors — such as the end of a file.

`catch` will run a list of instructions. If `throw` is called during the execution of these instructions, Logo returns control to the `catch` statement.

Look at the following procedures, for example. They print the squares of any number which you type.

```
1: build 'print.squares

print.squares
catch 'finish [forever [do.square ask
        [please type a number]]]
print.squares ; continue if 'finish is thrown

1: build 'do.square

do.square 'text
unless count :text = 1 & numberq first :text
        [say [please type one digit] throw 'finish]
say first :text * first :text
```

If you type something other than a number, the procedure `do.square` prints a warning message then returns control to `print.squares` after the `catch` statement. The process then continues.

Debugging Your Programs

A program which does not work is said to have a *bug* in it. The process of finding and removing bugs is called *debugging*.

You can tell when a program doesn't work in two ways: when you run the program, Logo may print an error message, or the program may not do what you expected.

The first action you can take is one of *prevention*. You can reduce the number of potential bugs in a program by designing it as a number of procedures, each of which is so small that it is unlikely to contain more than one bug. Each procedure can then be tested separately.

If problems still exist, the next thing to do is look at the program carefully. You can often 'squash' a bug by sitting down with a pencil and a listing of the program, and working out carefully what is going on inside the computer. You can get a listing of the program on paper by using the `copy` command, for example:

```
1: copy 'face
```

will print out the procedure `face`. You might need to type `face` first however! It is listed on the following page. You can display the program on the screen by using `printout` (or `po`) command or the `edit` command. For example:

```
1: po 'face
1: edit 'face
```

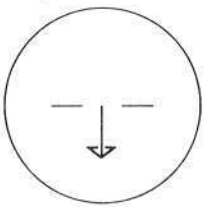
Following an edit command, the procedure appears in the "window" for editing, and you can change it as described in Chapter 4. The editing keys are summarised around the edit window remember.

Normally, `edit`, `po` and `copy` will tell you all you need to know to correct a program fault, but occasionally you may need Logo's special debugging tools. These are the `walk`, `trace` and `bug` commands, and symbolic dumps, and they are described in the rest of this Chapter.

Using `walk`

The most useful and powerful debugging tool is the command **`walk`**. Imagine that you've written a program called **`face`**, and it doesn't work exactly as you want it to. You really need to study it line by line as the program runs. Here is the program face:

```
1: build 'face

face
arcr 50 360
rt 90 lift
fd 20 drop
fd 20 lift
fd 20 drop
fd 20 lift
bk 30
rt 90 drop
fd 30
```



Now type the command:

```
1: walk 'face
```

Nothing happens, but Logo will remember that the program **`face`** is to be "walked" not "run" when you type its name.

As **`face`** uses the graphics screen, remember to type **`cs`** and then type:

```
1: face
```

Logo replies by typing the first line of the program, like this:

```
arcr 50 360...?
```

You can do one of four things now:

Press <ENTER>
Tells Logo that you want to run only this line of the program. Logo carries out the `arcr 50 360` command. Afterwards, it will print the next line, and wait for another response from you.

Press <ESC>
Tells Logo to stop immediately and return the `1:` prompt.

Press <F9>
Tells Logo to run through the rest of the procedure. When the procedure finishes, or invokes another, Logo starts walking again. After finishing, the program will be walked if you run it again.

Press <F10>
This tells Logo to stop walking and to run until the program stops or hits an error. After finishing, the program will be walked if you run it again.

You may also give `walk` a list of program names, like this:

```
1: walk [face arms legs]
```

To undo the effect of `walk`, you use `unwalk`:

```
1: unwalk 'face
```

Now face will run!

Using `trace`

For programs that do a calculation, the `trace` command may be more useful. It is used like the `walk` command:

`1: trace 'face` or `1: trace [face arms legs]`

and its effect is cancelled using the `untrace` command.

When the program `face` is now run, you will get a message each time it starts and each time it finishes. When it starts you get this message:

```
face was called...?
```

and when it finishes, you get this message:

```
face finished without a result
```

`trace` is useful with programs that do a calculation. For example, imagine that you wrote a program called `sum` which is supposed to return the sum of all the numbers held in a Logo list `'x`.

```
1: build 'sum

sum 'list
if emq :list [result 0]
result add first :list sum :list

1: make 'x [1 2 3 4 5]
```

The expected result is 15. However, try `sum` and you'll find that you need to use the <ESC> key to stop the inaction. The message `Panic: No nodes left` might appear on the screen but don't panic — there isn't anything wrong. Logo has detected a never-ending program.

To find out where `sum` is at fault, use the `trace` command. In this example, `sum` is obviously the culprit since you are not calling any other programs. However, sometimes you will be running a main program that calls two or three subroutines, and you can then `trace` any, or all, suspect subroutines at the same time.

If you issue the following two commands:

```
1: trace 'sum
1: print sum :x
```

Logo replies:

```
sum was called with these inputs
'list = [1 2 3 4 5]...?

sum was called in this line of "sum"
result add first :list sum :list
with these inputs
'list = [1 2 3 4 5]...?
```

In this example, the list is printed out by `trace`. You will find that the message is repeated on screen and the list is not reducing as might be expected. The line of the procedure that action is sticking at is:

```
result add first :list sum :list
```

After a closer look, you might realise that it ought to be:

```
result add first :list sum butfirst :list
```

Press the <ESC> key and then amend `sum`.

Using bug

A third tool is `bug`. If you want to be notified every time the variable called `x` changes, type:

```
1: bug 'x
```

Then, when you type

```
1: make 'x 5
```

Logo acknowledges it with: `x = 5...?`

As with `trace` and `walk`, you can give `bug` a list of variable names. `bug` and `trace` will also tell you which line of which program was running:

A variable must exist before you can `bug` it. If you haven't created a variable, then `bug` fails. This means that inside a program that uses the `new` instruction, you will have to insert `bug` like this:

```
new "x
bug "x
```

To undo the effect of `bug`, use the `unbug` command.

Symbolic Dumps

A symbolic dump will print the current values of all global variables. Type:

```
1: dump
```

You can stop a symbolic dump by pressing <ESC>.

`dump` is provided as a library procedure on your RM Logo disk.

Summary of Primitives

`bug`
: Prints message when contents of variable changes

`error`
: Returns data about last error

`grievance`
: Returns text of latest error message

`moan`
: Reproduces the last error

`trace`
: Gives message when procedure is used

`walk`
: Prints lines before execution

`unbug`
: Cancels effect of bug

`untrace`
: Cancels effect of trace

`unwalk`
: Cancels effect of walk

Chapter 13
Parallel Processing

Introduction

You can quite often split a problem into a number of smaller problems, and solve them separately. This approach is very useful in programming and it makes debugging programs a lot easier.

RM Logo allows you to take this approach a step further. In some cases procedures can be run independently of one another. RM Logo allows you to treat such procedures as separate *parallel processes* and run them *all* at the same time.

The significance of the `1:` prompt should become apparent. When a process is created, it is given a *process number*. Whenever the keyboard is communicating with that process, the process number is included in the prompt. Thus, if you are 'talking' to process 3, the prompt will become:

```
3:
```

When you are not using more than one process, you are talking to process 1. Hence the normal `1:` prompt.

To see how parallel processes work, try a simple example. First, build the following procedure:

```
1: build 'spin.square

spin.square
forever [square lt 43]
```

Carefully type the following sequence of instructions and watch what happens:

```
1: spin.square <ENTER>
```
(press <CTRL/X>)
```
2:tell 2 setx -50 forever [square lt 31] <ENTER>
```
(press <CTRL/X>)
```
3:say 5 + 6 * 7
47
```

The first line you type runs the procedure `spin.square` as process 1. When you press <CTRL/X> this allows you to create and run process 2. Once process two is running, the third process can be run: it merely performs a calculation.

You can use the next procedure in the same way:

```
1: build 'count_up

count_up 'no 'position
setcursor :position
say :no
count_up :no + 1 :position
```

This moves the cursor to a given position and counts up from a given number. You can set two counters going at once with the commands:

```
1: count_up 1 [5 20]
```
(press <CTRL/X>)
```
2: count_up 100 [5 30]
```

To stop both processes, press <ESC>.

If you run more than two processes in parallel, or if the initiation sequences are complicated, this method of running becomes tedious.

If this happens, you can use RM Logo's `parallel` primitive. This runs a number of other processes in parallel and suspends the current process until they all stop. For example, you could set 3 counters going as follows:

```
1: parallel [[count_up 1 [5 20]] [count_up
        100 [5 30]] [count_up 1000 [5 40]]]
```

Sometimes `begin` is more appropriate than `parallel`. `begin` starts a new process which runs concurrently with the existing processes.

When using parallel processes, you should remember that a parallel process does not inherit turtles from the process which started it. So the following will cause an error:

```
begin [forward 50]
```

You can get round such a problem by using:

```
begin [tell 1 forward 50]
```

Problems With Parallel Processing

When two or more processes interact, problems of mutual exclusion and synchronization can occur. These are problems of timing and each is described in the following sections.

Mutual Exclusion

The need for mutual exclusion is illustrated by the following piece of code which has two processes adding elements to the end of a list.

```
1: make 'x []
1: parallel [[repeat 10 [make 'x putlast
        :x 'a ]] [repeat 10 [make 'x putlast
        :x 'b ]]]
```

You might expect the end product to be a list with 20 elements; however, fewer than 20 are left in it. This is because both processes try to expand the list at the same time and overwrite each other.

mutual exclusion is needed to ensure that only one process can access the list at any time: the `single` primitive gives us this facility. To see how it works, first of all, define the following procedure:

```
1: 'build add_to_list

add_to_list 'data
single
make 'x putlast :x :data
multiple
```

Now type the following:

```
1: make 'x []
1: parallel [[repeat 10 [add_to_list 'a]]
        [repeat 10 [add_to_list 'b]]]
```

The procedure `add_to_list` updates the list and the command `single` tells RM Logo that no other process can run while it is doing this. So you now end up with all 20 elements in the list. Logo reverts to the default state, where several processes can access one procedure, when the primitive `multiple` is used.

Synchronization

The second problem of parallel processing is one of *synchronization* between processes. When parallel processes are running, there are times when one process cannot continue until a specific event has taken place. The process must indicate that it is waiting for the event or that the event has taken place, so that other processes can continue.

Suppose, for example, you have a global variable `x` which is initially set to 0, and two processes, `counter1` and `counter2`, access it. `counter1` adds one to `x` until `x` is 100, prints the message `[x is 100]` and then waits until `counter2` finishes and then sets `x` to 0.

The procedures can be defined as follows:

```
1: build 'counter1

counter1
make 'x :x + 1
if :x = 100 [say [x is 100] await :x = 0]
counter1

1: build 'counter2

counter2
await :x = 100
make 'y :y + 1
say :y
make 'x 0
counter2
```

In each procedure, the primitive `await` is used to stop each process until a specific event occurs.

Now run the procedures by typing the following:

```
1: make 'x 0
1: make 'y 0
1: counter1
(press <CTRL/X>)
2: counter2
```

Synchonization is also important when two procedures are accessing the same list: for example `add.to.list` is adding data to the list and `take.from.list` is extracting data from it.

At some instant the list looks like the following:

```
[p o n m l]
```

If `add.to.list` requests to add `'k` to the list (making it `[p o n m l k]`) and `take.from.list`

requests to delete `'p` from the list (making it `[o n m l]`), there is a possibility of errors occurring while amending the list.

Mutual exclusion is needed to ensure that `add.to.list` and `take.from.list` are not accessing the list at exactly the same time. However, the two processes also need to be synchronized so that, when the list is empty, `take.from.list` must wait until `add.to.list` has added an item to the list. The two procedures are:

```
1: build 'add.to.list

add.to.list 'data.item
single
make 'datalist putlast :datalist :data.item
multiple

1: build 'take.from.list

take.from.list 'var.name
single
if emptyq :datalist [multiple result 'false]
make :var.name first :datalist
make 'datalist rest :datalist
multiple
result 'true
```

Problems with Local Variables

A parallel process does not inherit local variables from the process which started it off. Hence:

```
1: build 'my.program

my.prog 'x
begin [print :x]
```

will either print the global variable `:x` or crash, since

:x has no value in the new process. The following example shows how you can get around this:

```
1: build 'each

each 'action 'group
begin putlast putlast [each.do] :action :group

1: build 'each.do

each.do 'action 'group
if emptyq :group [stop]
tell first :group
run :action
each.do :action rest :group
```

Using these procedures, the following line will change the colour of four turtles to red but will not change the number of the turtle that process 1 is talking to:

```
1: each [lt pick 90 fd pick 50] [William
        Henry Douglas Fred]
```

A further subtle problem emerges when you want to allot one command to each turtle, rather than giving a single process to act on each turtle in turn. You might want to do this if the process was going to be time consuming.

The previous procedure could be rewritten as:

```
each 'action 'group
if emq :group [stop]
begin pl pl [each_do] :action first :group
        each :action rest :group
each_do 'action 'name
tell :name
run :action
```

This will fail with the mystifying error "William does not exist". Yet you know William does!

The mistake happens because each `each_do` receives the unquoted word `William.first` yields 'William, but `putlast` strips the quote off again and `begin` attempts to execute:

```
begin [each_do [setc 10] William]
```

There are two possible remedies. One is to put quote in the list as:

```
each [setc 10] ['William 'Henry 'Douglas 'Fred]
```

The other is to join a quote onto each element as it is extracted from the list, so changing line 2 of `each` to:

```
begin pl pl [each_do] :action join '\' first :group
```

Example of Parallel Processing

The file `cage.def` on your RM Logo disk is a simulation of the lift example in Paul Chung's research paper 243*, showing an example of synchronization with a 'bounded' buffer.

* P.W.H. Chung. Concurrent Logo: a language for teaching model applications; DAI Research Paper 243, Oct 1984; in Logo Almanack *1*, Part 2.

Summary of Primitives

`await`
: Suspends process until a condition is `'true`

`begin`
: Runs command at same time as calling process

`multiple`
: Turns parallel processing on

`parallel`
: Suspends calling process and runs list of commands

`single`
: Turns off parallel processing

`whenever`
: When condition is true, Logo runs a given command

Chapter 14
Using Multiple Turtles

RM Logo allows you to have up to eight turtles on your screen at any time. You can use them to make the drawing of several complex shapes simultaneously or to build up moving pictures. This chapter explains how to do both these things.

Drawing Complex Shapes Simultaneously

The following procedure `spin.square` draws twelve "spinning squares" at the centre of your screen using the procedure `square`:

```
1: build 'square

square
repeat 4 [fd 30 lt 90]

1: build 'spin.square

spin.square
repeat 12 [lt 30 square]
```

Up to now, if you had wanted to draw a spinning square in each corner of the screen and one at the centre, you would have had to draw them one at a time. However, using the RM Logo multiple turtle feature you can have them drawn simultaneously.

Type in the following procedure for example, and try it out:

```
1: build 'smoother.squares

smoother.squares
tell 1
tell 2 setpos [-50 50]
tell 3 setpos [50 50]
tell 4 setpos [-50 -50]
tell 5 setpos [50 -50]
tell [1 2 3 4 5]
spin.square
```

The `tell` command "selects" zero or more turtles by name or number and applies subsequent commands to them until you use another `tell` command. A new turtle is created at the centre of the drawing area whenever you use `tell` followed by a name or number of a turtle not yet known to Logo.

The sixth `tell` is slightly different. It addresses all five turtles and applies subsequent commands to all five at the same time. Try it and see.

Turtles can be addressed by name, instead of by number, if you wish. You can remove them from the list of active turtles with the primitive `vanish`.

Each turtle can take on a different shape, a different colour and a different pen colour. For example, change `smoother.squares` to the following and try it again:

```
1: edit 'smoother.squares

smoother.squares
tell 1 setc 1 setpc 1
tell 2 setpos [-50 50] setc 2 setpc 6
tell 3 setpos [50 50] setc 3 setpc 7
tell 4 setpos [-50 -50] setc 4 setpc 8
tell 5 setpos [50 -50] setc 5 setpc 9
tell [1 2 3 4 5]
spin.square
```

Each turtle can also take on its own speed and direction of motion.

Drawing Different Shapes Simultaneously

The method described above is very effective if all the shapes you want to draw are the same. If they are not, the resulting program is not very elegant.

When you want to draw different shapes at the same time, the most effective method is to use parallel processes. For example, suppose you want to draw a spinning square at each corner of the screen and a "spinning triangle" at the centre. You could do it with the following additional procedures:

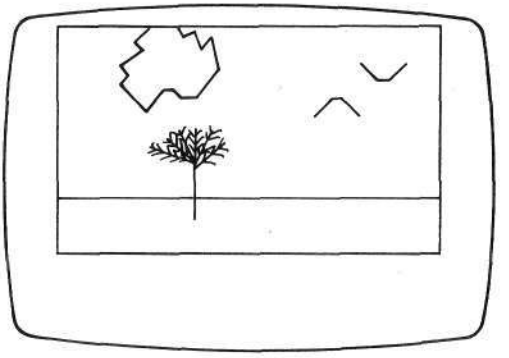
```
1: build 'picture

picture
tell 1 setc 1 setpc 1
tell 2 setpos [-50 50] setc 2 setpc 2
tell 3 setpos [50 50] setc 2 setpc 2
tell 4 setpos [-50 -50] setc 2 setpc 2
tell 5 setpos [50 -50] setc 2 setpc 2
parallel [[corners] [middle]]

1: build 'corners

corners
tell [2 3 4 5]
spin.square

1: build 'middle

middle
tell 1
spin.triangle
```

```
1: build 'spin.triangle

spin.triangle
repeat 12 [rt 30 triangle]

1: build 'triangle

triangle
repeat 3 [fd 50 lt 120]
```

`picture` creates all of the turtles and defines their colours and positions. It then starts two separate processes (`corners` and `middle`) in parallel, which draw the spinning squares and triangle independently of one another.

Creating Moving Pictures

One of the more interesting ways of using multiple turtles is to create a moving picture, like that shown (without motion!) below.



The picture consists of the earth, sky, a tree, moving birds and a cloud which moves from left to right. The easiest way to build such a picture is to program each of its parts as a separate procedure and then call them all from one controlling procedure.

For example:

```
1: build 'scene

scene
cs tell 1
cloud.shape
bird.shapes
create.earth.and.sky
create.tree
create.birds.and.fly
create.cloud.and.move
```

By doing it in this way, an error in one part of the scene will not affect the rest of the program. If you are working in a group, it also means that each person or sub-group can design, build and debug one part of the program.

Let's look at the stationary parts of the scene first: the earth, sky and tree. The earth and sky are easy; all we need do is draw the "horizon" on the screen then fill the areas below and above it with colour. The following procedure will do this:

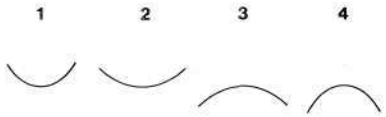
```
1: build 'create.earth.and.sky

create.earth.and.sky
setpc 4
setpos [-160 -50]
seth 90
forward 320
setpos [0 -70]
fill  0  0
```

The instruction before the `fill` command is there because you must move the turtle within the area before you can `fill` it with colour.

Now, the tree can be drawn using the following procedures:

```
1: build 'create.tree

create.tree
setpc 6
setpos [-50 -50]
seth 0
forward 25
canopy
backward 25

1: build 'canopy

canopy
left 60
repeat 6 [bough right 20]
bough
left 60

1: build 'bough

forward 7
twig
forward 7
twig
forward 5
backward 20

1: build 'twig

twig
left 45
forward 5
backward 5
right 90
forward 5
backward 5
left 45
```

The birds require a little more thought. First create four bird shapes:




```
1: build 'bird.shapes

bird.shapes
dsh [bird1 [0 0] lift [-10 6] [-8 4] [-6 2]
        [0 0] [6 2] [8 4] [10 6]]
dsh [bird2 [0 0] lift [-10 2] [-8 1] [0 0]
        [8 1] [10 2]]

dsh [bird3 [0 0] lift [-10 -4] [-8 -2]
        [-6 -1] [0 0] [6 -1] [8 -2] [10 -4]]
dsh [bird4 [0 0] lift [-10 -8] [-8 -5]
        [-6 -3] [0 0] [6 -3] [8 -5] [10 -8]]
```

Position the turtles:

```
1: build 'create.birds.and.fly

create.birds.and.fly
tell 2
setpos [50 50]
setc 3
tell 3
setpos [100 70]
setc 3
```

To change the screen turtles to moving bird shapes, create the procedure `fly`.

```
1: build 'fly

fly 'speed
setshape 'bird1 pause :speed
setshape 'bird2 pause :speed
setshape 'bird3 pause :speed
setshape 'bird4 pause :speed / 2
setshape 'bird3 pause :speed / 2
setshape 'bird2 pause :speed / 2
fly :speed

1: build 'pause

pause 'n
local 'x
repeat int :n [make 'x :n]
```

A similar method is used to create a cloud.

```
1: build 'cloud.shape

cloud.shape
dsh [cloud [0 0] lift [-30 -20] [-32 -15]
        [-28 -8] [-35 0] [-20 5] [-15 15] [-18 8]
        [0 25] [5 20] [15 20] [20 5] [25 10] [30 -5]
        [25 -10] [15 -20] [5 -25] [0 -20] [-5 -20]
        [-20 -35] [-30 -20]]]
```

Having created a cloud shaped turtle and called it `'cloud`, a final procedure is needed to move it across the screen.

```
1: build 'create.cloud.and.move

create.cloud.and.move
tell 1
sety 70
setshape 'cloud
setdir 90
setspeed 15
begin [tell 2 fly 20] begin [tell 3 fly 30]
```

The picture building procedures are now complete and you can run them via the procedure `scene`.

Summary of Primitives

`defineshape, dsh`
: Specifies the shape of the turtle

`dir`
: Returns direction of movement

`hideturtle, ht`
: Hides turtle shape

`near`
: Tells you if turtle is close to another turtle

`nosense`
: Cancels `sense` command

`setc`
: Changes turtle colour

`setdir`
: Specifies the direction the turtle moves

`sense`
: Turtle senses presence of another turtle or change in background colour

`setshape`
: Changes current turtle shape

`setspeed`
: Gives turtle a constant speed

`shape`
: Returns current turtle shape

`shapedef`
: Returns shape as a list

`shapes`
: Returns list of defined turtle shapes

`showturtle, st`
: Makes turtle visible

`speed`
: Returns turtle's current speed

`tell`
: Applies subsequent commands to named turtles

`told`
: Returns name of current turtle

`toldq`
: Returns true if a turtle is obeying graphics commands

`touch`
: Returns the background colour under the pen

`vanish`
: Removes turtle from list of active turtles

Chapter 15
Setting Up A Logo Microworld

You may want to restrict the facilities that Logo offers, or extend them in some way, to produce a Logo learning environment, or *microworld*. You could:

- Redefine some of the primitives to change their effect. For example, you could redefine `forward` so that `forward 10` moves the turtle by 50 steps, instead of 10.
- Change the colours used at start up.
- Treat some of your procedures as 'primitives' which cannot be edited by users.
- Rename primitives for use with other languages.
- Create a news file to be displayed whenever someone starts up the system.

Suppose, for example, you want to create a turtle graphics microworld for young children such that:

- `f` means `forward 50`
- `b` means `back 50`
- `l` means `left 50`
- `r` means `right 50`

You would first create the procedures `f, b, l` and `r` as follows:

```
1: build 'f

f
forward 50

1: build 'b

b
backward 50

1: build 'r

r
right 50

1: build 'l

l
left 50
```

Now when you type `f` and press <ENTER> for example, it will have the same effect as:

```
forward 50
```

However, other users are still able to edit the new procedures, rename them or even delete them. If you want to stop them from doing this you should type:

```
1: bury [f b r l]
```

This 'buries' the named procedures in your workspace so that they now look like primitives. You can 'unbury' or 'expose' them at any time by typing:

```
1: expose [f b r l]
```

Suppose you now want to introduce children to the normal primitives `forward` and `backward`, together with the idea of inputs, but you want to redefine their range

such that `forward 10` and `backward 10` both give movements of 20 steps.

First edit `f` and `b` to take an input `distance` and to multiply it by two. As `forward` and `backward` are to be hidden, a special prefix `$` is used to distinguish between Logo's original version of `forward` and the modified version. `f` and `b` become:

```
1: edit 'f

f 'distance
$forward (:distance * 2)

1: edit 'b

b 'distance
$backward (:distance * 2)
```

Now type:

```
1: alias 'f 'forward
1: alias 'f 'fd
1: alias 'b 'backward
1: alias 'b 'bk
1: bury [f b forward fd backward bk]
```

This buries the new procedures as well as the original definitions of `forward` and `backward`.

The commands:

```
1: expose [f b forward fd backward bk]
1: scrap [backward bk forward fd]
```

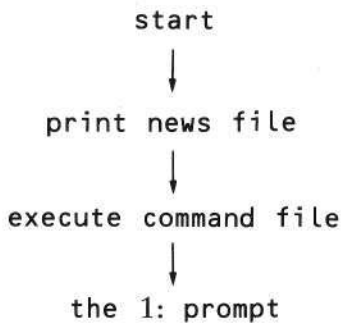
will cancel the aliasing and return `forward`, `fd`, `bk` and `backward` to their usual use.

You could use a similar approach to rename the basic movement primitives for other languages. For example, the following instructions allow the German words for the primitives `forward`, `backward`, `left` and `right` to be recognised.

```
1: alias 'forward 'vorwaerts
1: alias 'backward 'rueckwaerts
1: alias 'left 'links
1: alias 'right 'rechts
```

Preserving The Microworld

Ideally, your microworld should exist when Logo is loaded. The events which occur when Logo starts are shown in the diagram below:



The *news file* is named `news.lgo` and it allows information to be passed to users at the start of each 'session'. This information can be anything you want: instructions on how to use the microworld or school news, for example.

Now look at how you can create the turtle graphics microworld described earlier. All you need do is make a text file containing something like the following:

```
define [[f] [forward 50]]
define [[b] [backwards 50]]
define [[l] [left 50]]
define [[r] [right 50]]
bury [f b l r]
```

If the file containing the above is called `turtle.lgc` then it will be loaded if you start Logo with the MS-DOS command:

```
Logo turtle.lgc
```

rather than just Logo. This will then create the procedures `f, b, l` and `r` and make them appear as primitives while Logo is running. If you want to use the standard start-up file as well then add the line:

```
consult 'start.lgc
```

to the file `turtle.lgc`.

The file `start.lgc` contains the following:

- the filing primitives `load, save, get, keep`
- procedures giving the colour numbers by name
- the procedure `copy` which sends definitions of procedures to the standard MS-DOS printer channel `'prn`.
- the procedure `pos` which returns a list of the current turtle's x and y coordinates
- the procedure `find` which takes two inputs. The first is an object and the second a list. `find` returns a list pointing to the occurences of object in the list.
- the procedure `toldq` which takes an input `'name` and returns `'true` if `:name` is a turtle being addressed by the current process, and `'false` otherwise.

- the definitions of the shapes `train, coach, car, bicycle`
- the procedure `dump` which prints out the contents of all global variables.

Any of these files, including `news.lgo` and `start.lgc` can be edited outside Logo using a text editor.

You can also use Logo's editor to change them. First type:

```
1: load 'editfile.def
```

and then type `editfile` and the name of the file to edit. For example:

```
1: editfile 'start.lgc
```

You can also use the Logo editor outside Logo. To do this type `proword` and the name of the file to be edited after the MS-DOS drive prompt.

Summary of Primitives

`alias`
: Allows another name to refer to a procedure. The new name may be the name of a primitive.

`bury`
: Buries procedures so that they cannot be edited, listed, saved, renamed or deleted

`define`
: Defines a procedure in the form of a list

`expose`
: Unburies procedures

`rename`
: Renames a procedure (not a primitive)

Chapter 16
Writing Extensions To Logo

Introduction

This chapter is about extending Logo to communicate with devices beyond the control of the standard language.

An *extension* is a short machine code program which will let you do this. For example, a *floor turtle* is a small robot which trundles around the floor in response to Logo commands. You will need to load an extension into RM Logo if you want to drive a turtle with Logo commands given at the keyboard. This is called a *turtle driver*. Anyone can load a driver if it is ready-made, but you may want to write your own. You will also need to write your own extension to use Logo to control a device you've made yourself.

Following a general description of floor turtles, the first sections of this chapter explain how to load a ready made turtle and other extensions.

The requirements and explanations to prepare and write your own turtle driver or extensions are given in the remaining sections of the chapter.

Floor Turtles

A floor turtle is a small robot which responds to the Logo commands `forward`, `backward`, `left` and `right`. It also has a pen which responds to `lift` and `drop`. It sometimes has other features like a hooter, flashing lights, or touch sensitive switches.

With young children it is often very helpful to introduce the floor turtle before going on to the screen turtles. To do this, you need to add a program which can convert information about the primitive being run on Nimbus (`forward`, `backward` and the number given as argument) into motor movements. This program is called a *turtle driver*.

If you are using one of the common commercially sold turtles, Research Machines may have supplied a driver for it. If you are building your own turtle or using one for which you can't get a ready-made driver, you will have to write it yourself.

Loading a Ready-made Turtle Driver

Give the command:

```
driver  'abcd.lgx
```

This will unload any driver that is loaded already, and load the file ABCD.LGX as a driver. To load the driver without loading another, use the command `nodriver`. It has no inputs.

While a driver is loaded the commands `forward`, `backward` `left`, `right`, `lift` and `drop` communicate with the floor turtle as well as with any screen turtles. If you run both screen turtles and a floor turtle at the same time, then the floor turtle will move before the screen turtles. It also becomes legal to give these commands without first giving any `clearscreen` or `tell` commands.

Loading Ready-made Extensions

An *extension* is a short machine code program which lets you send signals to, and receive signals from, a feature of Nimbus which Logo cannot reach in its standard form.

The `bload` (*B*inary *LOAD*) command takes a word or a list as input, so:

```
1: bload 'abcd.lgx
```

loads the procedures defined in the extension `abcd.lgx`.

```
1: bload [abcd.lgx efgh.lgx]
```

loads the procedures defined in the extension files `abcd.lgx` and `efgh.lgx`.

You may have a number of extensions in Logo at the same time. To delete an extension give the command:

```
1: unbload 'abcd.lgx
```

To find out the names of the procedures that the extensions define (which may not be the same as the filenames) give the command:

```
1: say bloaded
```

Preparing to Write your own Turtle Driver or Extensions

To write a driver or extension you will need to understand:

- the way the 80186 chip works
- an assembly language
- an editor, an assembler and the MS-DOS utilities LINK and EXE2BIN.

The steps in writing a machine code extension are:

1. Use an editor to create the source file, for example: ABCD.ASM

2. Use an assembler to create an object file, for example: A:MASM ABCD

3. Use LINK to convert the object file into MS-DOS's load file format, for example: A:LINK ABCD

4. Use EXE2BIN to convert the file to binary format, for example: A:EXE2BIN ABCD

5. It has mnemonic value to rename the file, for example: REN ABCD.BIN TILLY.LGX

The file TILLY.LGX is now ready to be loaded into Logo.

If you want to invoke the floor turtle driver, load the two registers that the floor turtle routines checks first (AX and BX) and issue interrupt 0D4. If your floor turtle routine is likely to use registers SI or DI and you are treating them as reserved, then you must preserve them. Remember that floor turtle function codes run from 0 to 7 inclusive; yours should start at, say, 16 to allow a little room for RM functions to expand.

Writing a Floor Turtle Driver

The driver or extension should be a binary file in the form of a far procedure, ending with a far return instruction. The segment registers are all reserved. If you are calling the floor turtle from an extension procedure then registers SI and DI may also be reserved. An error exit is made by issuing the INT D2 hex instruction.

When the driver is invoked, the BX register is set to a function number and the AX register to the value of the input, if there is one; be warned that this may be negative.

The significance of the number in BX is:

0. Turtle driver is about to be unloaded. Turn off lights etc.

1. Turtle driver was just loaded. Turn on lights etc.

2. Forward command. Distance in AX

3. Backward command. Distance in AX

4. Left command. Angle in degrees in AX

5. Right command. Angle in degrees in AX

6. Pen up

7. Pen down

Writing Your Own Extensions

Extensions are very flexible; they can accept one or more numbers, words or lists as input and they can return a result which may also be a number, word or list. There is a restriction that neither input nor output lists may contain sublists, and input and output numbers must be integers in the range –32,768 to 32,767. They can invoke the turtle driver, if one is loaded, or the Logo error handling routine.

Format of Extension Files

Your extension should be a binary file in the form of a far procedure, ending with a far return instruction.

Before the code itself, there must be a header giving the name of every primitive in the extension. The names should be made of contiguous lower case characters; the name must be terminated with a null byte.

Immediately following this there should be one byte giving the arity (number of inputs) of the procedure, and following that two bytes giving the offset from the start of the extension to the start of the code for the primitive. The entire header should be terminated with a null byte.

For example:

```
db      'mouse'             ; name
db      0                   ; terminates name
db      0                   ; has no inputs
dw      offset              ; offset of code start
db      'button'
db      0
db      1                   ; has one input
dw      offset dobutton
db      0                   ; terminates header
```

is a valid header for an extension defining the primitives `mouse` and `button`, where `mouse` has no input and `button` has one.

Reading Inputs

If you want to read inputs then you must not disturb the DI register until you have read all the inputs you want. It's your responsibility to make sure the number of inputs you read is not greater than the arity declared in the header.

To read an input invoke interrupt 0D0 hex. This will alter the content of the DI, AX and BX registers. Leave DI alone. The BX register contains type information to allow you to make sense of the AX register, as follows:

BX	*AX*	*means*
2	--	Input is a list. (See below for how to scan it)
0	N	N is a number
1	SSSS	ES:AX is a null-terminated string of characters; the input was a word.

To scan a list invoke interrupt OD1 hex before invoking interrupt 0D0 again. The BX and AX registers will give you the first, or next, element of the list as follows:

BX	*AX*	*means*
2	--	Element is a sublist. It cannot be read but it can be stepped over with another interrupt 0C1.
0	N	Number, as above
1	SSSS	String, as above
–1	--	End of list

Returning Results

Even if your extension does not return a result you should be careful what is in the BX register when you return from your procedure with a far return instruction. It is interpreted as follows:

BX (on exit)	*result*
0 or any number not shown here	Your primitive does not return a result
1	The number in AX
2	The word formed from the null-terminated character string at DS:AX
3	The word `'false` if AX is zero and the word `'true` otherwise
4	A list built up as explained below

Returning Lists

To return a list, first clear the SI register and issue interrupt 0D3. The SI register is reserved from now on. The procedure will now return an empty list if you put 4 in BX and return. To append elements to the list load AX and BX as for exit (BX will always be 1, 2 or 3) and issue another interupt 0D3. To end the list, put 4 in BX and return with a far return instruction.

Error Exit

If you encounter an error, for example someone using your extension has supplied an inappropriate input, then use MS-DOS function 9 (put string) to display a message if you want, then invoke interrupt 0D2. This is the Logo error exit. It can be caught with `catch 'error` though this will not suppress the message.

Part 2 Reference

Logo Primitives

The following pages describe the Logo primitives in alphabetical order, one to a page. The primitive # is the final primitive to be described.

Most descriptions include an example of how to use the primitive. Remember that an indented Logo line indicates that it is a continuation of the line above.

Logo Keywords and Signals

The Logo *keywords*:

```
case, default, until, true, false
```

are listed following the special characters.

The Logo signals (things Logo catches) are explained following the Logo keywords. The signals are:

```
cancel, endfile, error, fence,
touch, touchturtle, escape
```

`catch` and `throw` primitives make use of these signals to change the flow of control. See Chapter 5 in the Concepts Section of this book.

Special Logo Characters

The special Logo characters:

```
: ; :-[ ] ' ( ) \ * + -/ | ↑ ~ $ # < > <-% &
```

are documented at the end of the Logo primitives.

Inputs to Logo Primitives

Where inputs are required, these are shown in italics. For example:

`add` *number number*
`first` *nwl*

The words used to define these inputs are as follows:

a,b
: An expression which is `'true` or `'false`

angle
: An angle specified in degrees

filename
: A quoted word identifying a file unambiguously and including any file extension.

list
: One or more *nwl*s enclosed by square brackets, or the empty list `[]`

integer
: An integer number

number
: A number, integer or fraction

nwl
: A number, word or list

word
: A quoted word: any set of alphanumeric or non-special characters preceded by `'`. Special characters must be prefixed with\.

  In a few instances, an unquoted word may also be accepted. These are not detailed explicitly in this book and their use is discouraged in order to keep words in the same form.

Any of the above definitions can be replaced by an expression or procedure, if the replacement returns an equivalent result.

abs *number*

Remarks

Returns the absolute value of the input (including a zero if the input is zero).

Examples

```
1: print abs -2
2
1: print abs 2
2
```

acos *number*

Remarks

Returns the arccosine of *number* as an angle in the range 180 to 0.

Examples

```
1: print acos -1
180
1: print acos 0.5
60
1: print acos 1
0
```

Associated Primitives

`asin, atan`

add *number number*

+

Remarks

Returns the sum of its inputs.

Examples

```
1: say add 10 2
12

1: say 10 + 2
12

1: make 'x 8
1: say add :x 1
9
```

alias *word1 word2*

Remarks

Gives the procedure named by *word1*, a new name *word2*. The original name will still be understood. Any editing on *word2* will change *word1* also.

This is the only way to re-use the name of a primitive. If *word1* is the name of a primitive then calls to *word2* will actually invoke *word1*.

Example

```
alias 'six.sided.figure 'hexagon
```

amongq *nwl list*
memberq

Remarks

Returns the value `'true` if *nwl* is an element of *list*, otherwise it returns `'false`. Differences between upper and lower case are ignored.

Examples

```
1: say amongq 'dog [cat dog hamster rabbit]
true

1: say memberq 'pig [cat dog hamster rabbit]
false
```

The following example tests if the contents of `list1` are a subset of `list2`:

```
1: build 'subsetq

subsetq 'list1 'list2
if emq :list1 [result 'true]
if amongq first :list1 :list2
        [result subsetq rest :list1 :list2]
        [result 'false]
```

Hence:

```
1: print subsetq [toast jam][ toast eggs jam]
'true
1: print subsetq [toast jam][cheese toast eggs]
'false
```

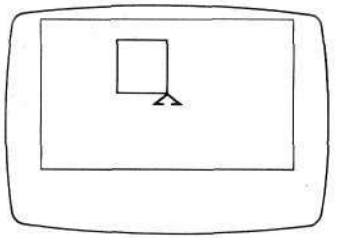
and

Remarks

Commands connected by **and** are run in sequence. **and** is optional.

Example

```
1: repeat 4 [forward 50 and left 90]
```



appfile *filename*

Remarks

Tells Logo to open the file *filename* for writing. The current contents of the file are preserved and new data is appended to the end of it. Returns `'true` if the file is opened successfully and `'false` if it is not.

If you try to append to a non-existent file, a new file will be created.

Example

The following command tries to open the file `datafile` as an 'append file'. If it manages to do this the program continues, otherwise it prints the message 'unable to open append file' and stops.

```
unless appfile 'datafile [say [unable to open
        append file] escape]
```

Associated Primitives

`closefile, outfile, infile`

arcl *number angle*

Remarks

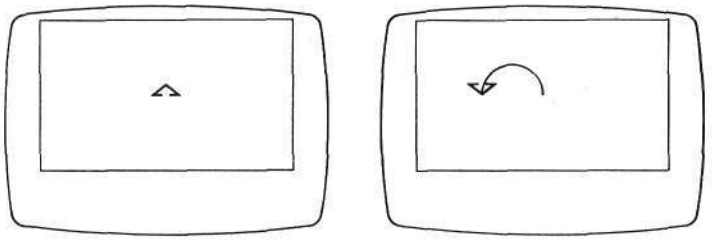
The turtle draws an arc to the left. *number* specifies the radius of the arc, *angle* is its size in degrees.

If the radius is negative, the centre of the arc is to the right of the turtle. If the angle is negative, the turtle moves backwards instead of forwards.

Example

To draw a semi-circle of radius 25 steps to the left:

```
1: cs
1: arcl 25 180
```



Associated Primitive

`arcr`

arcr *number angle*

Remarks

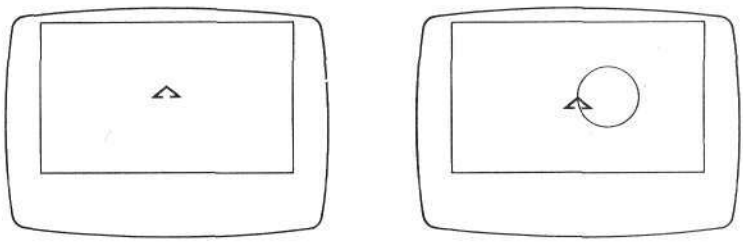
The turtle draws an arc to the right. *number* specifies the radius of the arc, *angle* is its size in degrees.

If the radius is negative, the centre of the arc is to the left of the turtle. If the angle is negative, the turtle moves backwards instead of forwards.

Example

To draw a circle of radius 25 steps to the right:

```
1: cs arcr 25 360
```



Associated Primitive

`arcl`

asin *number*

Remarks

Returns the arctangent of *number* as an angle in the range –90 to 90 degrees.

Examples

```
1: print asin -1
-90

1: print asin 0
0
```

Associated Primitives

`acos, atan`

ask *nwl*

Remarks

Prints *nwl* on the screen without outer brackets but with a question mark. Everthing that is typed from the keyboard in response is returned as a list. The response is ended by <ENTER>. The <ENTER> is not recognized as part of the response.

Example

```
1: make 'name ask [What is your name]
What is your name? Cathy

1: print :name
[Cathy]
```

Associated Primitives

```
key, keyq, readlist
```

assert *word1 word2 nwl*

Remarks

Gives *word1* the property *word2* with value *nwl*.

Examples

```
1: assert 'whiskers 'species 'cat
1: assert 'patch 'species 'dog
1: assert 'patch 'colour [black and white]
```

Associated Primitives

`asserted, assertedq, assertions, classified, deny, objects`

asserted *word1 word2*

Remarks

Returns the value held by the property *word2* of the object *word1*.

Example

```
1: assert 'rover 'breed 'labrador
1: say asserted 'rover 'breed
labrador
```

Associated Primitives

`assert, assertedq, assertions, classified, deny, objects`

assertedq *word1 word2*

Remarks

Returns `'true` if the object *word1* has the property *word2* and returns `'false` otherwise.

Example

`assertedq` can be used to anticipate errors when you are about to use `asserted`

```
1: assert 'GWR 'abbreviation.of [great
      western railway]
1: print assertedq 'GWR 'abbreviation.of
'true
```

Associated Primitives

`assert, asserted, assertions, classified, deny, objects`

assertions *word*

Remarks

Returns a list consisting of one or more sub-lists. Each sub-list consists of a property of the object *word*, together with its associated value.

Example

```
1: say assertions 'rover
[species dog] [breed labrador] [colour golden]
```

Associated Primitives

`assert, asserted, assertedq, classified, deny, objects`

atan *number*

Remarks

Returns the arctangent of *number* as an angle between –90 and 90 degrees.

Examples

```
1: say atan 1
45

1: say atan 0.5
26.565051177078
```

await *a*

Remarks

The process which issued the `await` will be suspended until the expression *a* becomes true. If *a* uses a global variable, `await` can be used by a process to delay another. If *a* is a procedure, it will be run at least once.

Examples

Suppose `'x` is a global variable, initially set to 0. Two processes, `counter1` and `counter2`, access it. `counter1` adds 1 to `'x` until `'x` is 100, prints a message and then waits until `process2` does its work resetting `'x` to 0.

```
1: build 'counter1

counter1
make 'x :x + 1
if :x = 100 [say [x is 100] await :x = 0]
counter1

1: build 'counter2

counter2
await :x = 100
make 'y :y + 1
say :y
make 'x 0
counter2

1: make 'x 0
1: make 'y 0
1: parallel [[counter1][counter2]]
```

Associated Primitives

`begin, parallel, whenever, single, multiple`

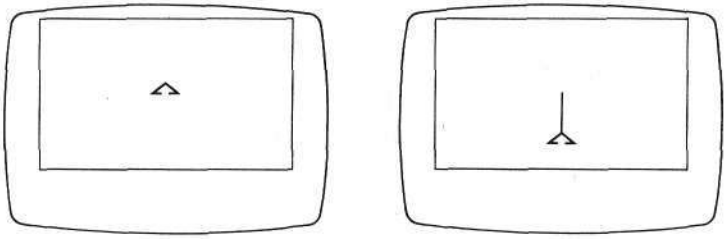
backward *number*
bk

Remarks

Moves the turtle *number* steps backwards. If the turtle's pen is down, the turtle leaves a trace of its path. If *number* has a negative value, the turtle will move forwards.

Example

```
1: cs
1: backward 50
```



Associated Primitive

`forward`

begin *list*

Remarks

The command *list* is run in parallel with the current process. It is usually used to start another process off from within an existing process.

Example

```
begin [tell 5 repeat 4 [forward 50 left 90]
```

Associated Primitives

parallel, run

bg

Remarks

Returns the current background colour, which will always be a number in the range 0 to 15. The colour numbers are listed under the primitive `colour`.

Examples

```
1: say bg
0

1: setbg 2
1: cs
1: say bg
2
```

Associated Primitive

`setbg`

bload *filename*

Remarks

Loads the extension in *filename* into Logo's workspace. The procedures in this extension then become accessible from Logo.

Example

```
1: bload 'control.lgx
```

Associated Primitives

`bloaded, unbload`

bloaded

Remarks

Returns a list of procedures defined by any extensions currently loaded.

Example

```
1: print bloaded
[peek poke epson]
```

Associated Primitives

`bload, unbload`

both *a b*
&

Remarks

Returns `'true` if both the expressions *a* and *b* are `'true`, and `'false` otherwise.

The table below shows how `both` works for different values of *a* and *b*.

a	*b*	both *a b*
`'false`	`'false`	`'false`
`'false`	`'true`	`'false`
`'true`	`'false`	`'false`
`'true`	`'true`	`'true`

`both` does not evaluate its 2nd input if the 1st is `'false`.

Examples

```
1: say both (1 = 1) (2 = 2)
true

1: say both (1 = 2) (2 = 2)
false

1: say (1 = 2) & (2 = 2)
false
```

The following example procedure tests for integer square roots and avoids a failure if the input is negative:

```
1: build 'has_exact_square_root

has_exact_square_root 'n
result (:n>=0) & (sqt :n = int sqt :n)
```

branch *a list* **case** *b list*

Remarks

This statement is a set of condition and list pairs. The conditions are called `cases`.

Logo looks to find the first case that is true. The *list* following that case is run and the `branch` statement is then finished. If a result is returned from the *list* then it is the result of the `branch` statement.

If the final case is the keyword `default` and none of the other cases proved true, then the final expression will be evaluated and run.

Example

```
1: build 'sign

sign 'x
branch :x>0 [result 'positive] case :x=0
        [result 'zero]
default [result 'negative]
```

bug *wl*
bug *list*

Remarks

Whenever the value stored in the named variable(s) changes, Logo print a message. *word* or every variable named in the *list* must exist before you use **bug**.

Example

```
1: bug  'name
1: bug [dog cat]
```

Associated Primitive

unbug

build *word*

Remarks

Invokes the editor. If *word* already exists then the definition appears in the edit window: otherwise only *word* appears. It is an error to leave the editor and return only the *word*.

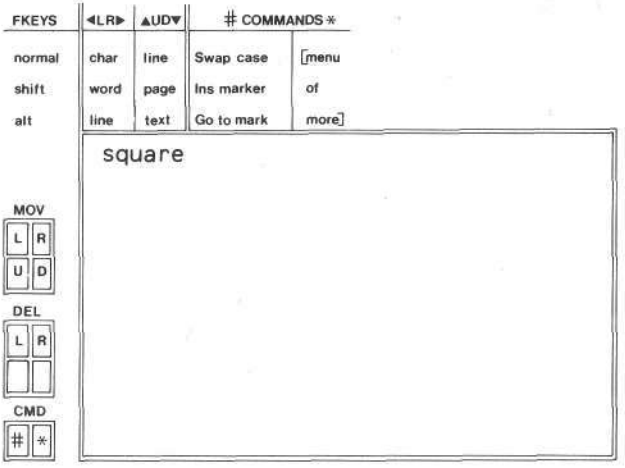
Please see chapter 4 for more details.

Example

If you wanted to create the procedure `square`, you would type:

```
1: build 'square
```

and the edit window would appear:



Associated Primitive

`edit`

bury *wl*

Remarks

Allows you to 'bury' procedures in the workspace so that they cannot be listed, edited, saved, renamed or deleted. The buried procedures will then have the appearance of primitives.

Examples

The following example lists the names of all procedures in the workspace then buries some of them.

```
1: say titles
square triangle rhombus hexagon

1: bury [square triangle hexagon]

1: say titles
rhombus
```

Associated Primitive

`expose`

butfirst *nwl*
bf
rest

Remarks

Returns all but the first element of *nwl*, which can be a number, word or list.

nwl cannot be empty.

Examples

```
1: say butfirst 'cats
ats

1: say rest [tortoiseshell cats are great]
cats are great
```

Associated Primitives

`butlast, first, last`

butlast *nwl*
bl

Remarks

Returns all but the last element of *nwl*, which can be a number, word or list.

nwl cannot be empty.

Examples

```
1: say butlast 'cats
cat

1: say butlast [tortoiseshell cats are great]
tortoiseshell cats are
```

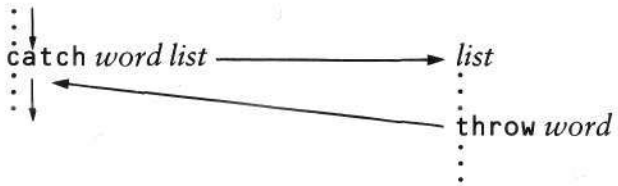
Associated Primitives

`first, last, butfirst, butlast`

catch *word list*

Remarks

The expression *list* is evaluated and run. If a `throw` command with the label *word* occurs while running *list*, control returns immediately to the command following `catch`.



Please see the section 'Throwing and Catching Control' in Chapter 5.

The system throws several signals which can be caught by `catch`. These are thrown by `cancel`, `endfile`, `escape`, `error`, `touch`, and `touchturtle`.

word can be one of the system names, such as `error`.

Example

```
catch "trouble [explore]
```

If `explore`, or anything it calls, contains the command `throw "trouble`, then no further lines of `explore` are run. Instead, Logo continues with the command following `catch`.

Associated Primitive

`throw`

centre
center
ct

Remarks

The turtle is moved to its 'home' position. This is at the centre of the screen and facing upwards (a heading of 0).

When a process issues a `centre` command, only the turtles addressed by that process are affected.

classified *word*

Remarks

Returns a list of objects which have the property *word*.

Example

```
1: print classified 'species
[rover whiskers patch fido joey]
```

Associated Primitives

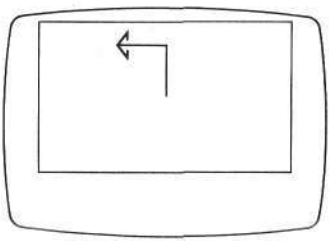
`assert, asserted, assertedq, assertions, deny, objects`

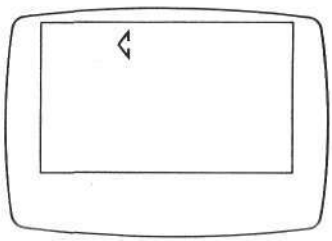
clean
cl

Remarks

Everything except the turtle shapes disappears from the screen. The turtle shapes are not moved.

```
1: cs
1: forward 50
1: left 90
1: forward 50
1: cl
```

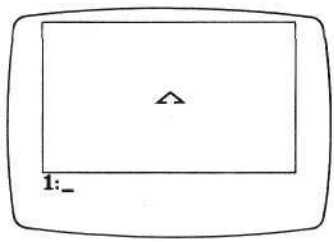




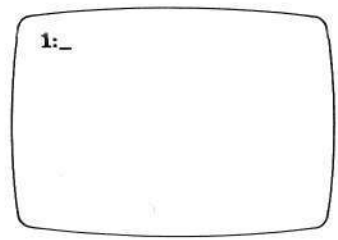
cleantext
ctx

Remarks

Erases all text on the screen, returning the cursor to the top left of the text area. This will not affect the graphics area if you are in graphics mode.



graphics mode



text mode

clearscreen

cs

Remarks

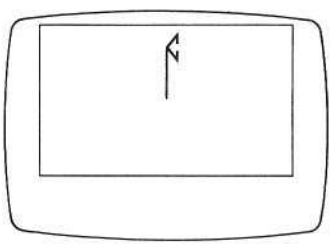
This clears the screen ready for drawing with a turtle. The bottom six lines are for text; the remainder of the screen is the drawing area. Everything on the screen other than the turtle disappears.

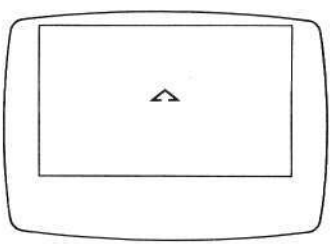
When a process issues a `clearscreen` command, only the turtles addressed by that process are moved to the centre of the screen.

If `clearscreen` is used for the first time, only one turtle appears. It's name is `seymour` and is turtle number one.

Examples

```
1: forward 50
1: left 90
1: cs
```





closefile *filename*

Remarks

Tells Logo to tidy the file *filename* and close it. Returns `'true` if the file was closed successfully and `'false` otherwise.

Examples

The following line tries to close a file. If it manages to do this, it prints `'end of run'`, otherwise, it prints: `'unable to close output file'` and stops.

```
1: if closefile 'outfile.dat [say [end of run]]
[say [unable to close output file] stop]
```

Associated Primitives

`appfile, infile, outfile`

colour
color

Remarks

Returns the colour of the turtle as a number 0 to 15. The numbers associated with the colours are:

Number	*Colour*
0	black
1	dark.blue
2	dark.red
3	purple
4	dark.green
5	dark.cyan
6	brown
7	light.grey
8	dark.grey
9	light.blue
10	light.red
11	magenta
12	light.green
13	cyan
14	yellow
15	white

Example

```
1: setc 3
1: say colour
3
```

Associated Primitive

`setc`

consult *filename*

Remarks

Executes commands previously written (dribbled) to the file *filename*.The commands are not visible on the screen as Logo executes them.Control returns to the keyboard when either an error occurs or the end of the file is reached.

The file can be created from Logo using `dribble` or file primitives, and from outside Logo using a text editor.

Example

```
1: consult 'turtles1
```

Associated Primitives

`dribble, nodribble, replay`

cos *angle*

Remarks

Returns the cosine of *angle*. *angle* is in degrees.

Example

```
1: say cos 60
0.5
```

count *nwl*

Remarks

Returns a number as its result:

- If the input is a number, count returns the number of digits to the left of the decimal point
- If the input is a word, count returns the number of characters in the word
- If the input is a list, count returns the number of elements in the list

Examples

```
1: say count 12.223
2
1: say count 'jeeves
6
1: say count [number of words in list]
5
```

The following example deletes an element of a list using count to check the value input:

```
1: build 'deletes.item

deletes.item 'n 'list
if :n > count :list [say [n is too big] escape]
if :n = 1 [result rest :list]
result putfirst first :list delete.item :n-1
        rest :list
```

Hence:

```
1: print delete.item 3 [the cat sat on the mat]
[the cat on the mat]
1: print delete.item 8 [the cat sat on the mat]
n is too big.
```

cursor

Remarks

Returns a list of two numbers which are the screen line and column number of the cursor position.

Example

```
1: print cursor
[20 16]
```

Associated Primitive

```
setcursor
```

define *list*

Remarks

Defines a procedure in the form of a list. Each line of the procedure is an element of *list.*

The first element is the title line of the defined procedure. If there is no procedure with this name, a new one will be created. If this is already the name of a procedure, the new definition will replace the existing one.

Example

The following command defines a procedure which draws a house shape. It uses two other procedures which draw a square and a triangle.

```
1: define [[house 'side] [right 90]
        [square :side] [left 60] [triangle :side]]
```

Associated Primitive

`text`

defineshape *list*
dsh

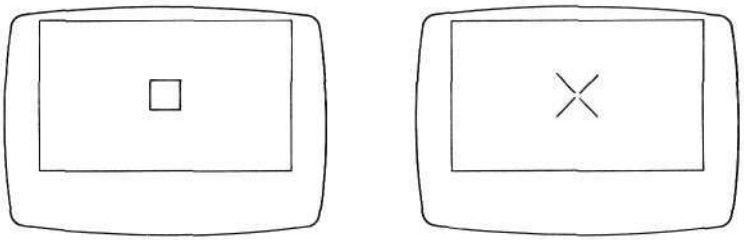
Remarks

Defines a turtle shape in the form of a list. The first element of *list* is the name of the shape; other elements are the coordinates of the vertices of the shape. These vertices are joined to form the shape. Use `lift` if you require a break in the line of the turtle shape.

The turtle will rotate about its centre. The centre of the new turtle shape is always the centre of the previous turtle shape.

Examples

```
1: dsh [box [-8 -8][-8 8][8 8][8 -8][-8 -8]]

1: defineshape [cross [-10 -10][10 10]
       lift [-10 10][10 -10]]
1: setshape 'box
1: setshape 'cross
```



Although neither example contains [0 0], each turtle will rotate about its centre.

deny *word1 word2*

Remarks

Deletes the property *word2* and its value for the object *word1*.

Examples

```
1: deny 'fido 'species
1: deny 'fido 'type
```

Associated Primitives

```
assert, asserted, assertedq, assertions,
classified, objects
```

dir

Remarks

Returns the direction in which the turtle is moving (its *movement heading*, not its drawing heading).

Examples

```
1: clearscreen
1: tell 1
1: setdir 45
1: say dir
45
```

`tell` is not necessary here if you already have a turtle. However, if you haven't, the `setdir` command will cause an error. If you set a negative direction, `dir` will give you the positive equivalent. For example:

```
1: setdir -45
1: say dir
315
```

Associated Primitive

`setdir`

directory *word*

Remarks

Returns directory information from the disk in the form of a Logo list. *word* is used to select the information required and depends upon the operating system.

Example

Under MS-DOS the following command will return a list of all files on disk B with the extension .LGP

```
1: say directory 'b\:\*.lgp
```

Note how the \ character is used to tell Logo to interpret the following character as normal text.

divide *number number*
div
/

Remarks

Returns the number that results from dividing the first input by the second.

Examples

```
1: say divide 13 2
6.5

1: say 12 / 2
6
```

do *list* until *a*

Remarks

The command *list* is repeated until the expression *a* becomes `'true`. *list* is carried out at least once.

Examples

```
1: do [forward 100 left 90] until keyq
```

The following procedures find the first prime number after the number input. It assumes that the number input is a prime number.

```
1: build 'find.first.prime.after

find.first.prime.after 'n
do [make 'n :n+1] until primeq :n
result :n

1: build 'primeq

primeq 'n
result primeq.sub 2 sqt :n :n

1: build 'primeq.sub

primeq.sub 'i 'j 'n
branch rem :n :i =0 [result 'false]
        case :i >:j [result 'true]
        default [result is.prime.sub :i+1 :j :n]
```

dribble *filename*

Remarks

Subsequent commands will be written (dribbled) to the file *filename*.

`dribble` is mainly used when you need a record of everything typed. Hence, the file is opened in append mode, preserving any existing contents.

Example

```
1: dribble 'session
```

Associated Primitives

`consult, dribbleq, nodribble, replay`

dribbleq

Remarks

Returns `'true` if everything you type is being recorded in a dribble file, otherwise it returns `'false`.

Example

```
1: if dribbleq [say [hello I am watching you]]
```

Associated Primitives

`consult, dribble, nodribble, replay`

driver *word*

Remarks

Loads a floor turtle driver contained in the file specified by *word* providing that such a driver exists. There can only be one floor turtle driver present in Logo at any one time.

Please see chapter 16 for details.

Associated Primitive

`nodriver`

drop

Remarks

Replaces the turtle's pen on the paper so that it draws as it moves. This primitive is the opposite of `lift`.

Examples

The following procedure draws a dotted line:

```
1: build 'dotted.line

dotted.line 'length 'dot.size 'pen.state
if :dot.size >:length [fd :length stop]
fd :dot.size
if :pen.state [lift][drop]
dotted.line :length -:dot.size :dot.size
        not :pen.state
```

Hence:

```
1: drop
1: dotted.line 50 8 'true
```

Associated Primitives

`lift, upq`

edit *word*

Remarks

Invokes the editor and places the definition of *word* into the edit window. It is not possible to edit a procedure or primitive which does not exist. `edit` cannot be used to create a procedure or primitive (use `build`).

Example

If the procedure `square` exists and you type:

```
1: edit 'square
```

then the procedure appears in the edit window.

Associated Primitive

`build`

editlist *list*

Remarks

Displays the elements of *list* in the edit window. The list is not broken down into sub-lists (as it is using `edlist`). Once in the edit window, the elements of the list can be edited like any other text.

On leaving the editor, the list is returned in the same form it took before going into the editor — except for any amendments made in the editor.

Associated Primitive

`edlist`

edlist *list*

Remarks

Displays the elements of *list* in the edit window. Once in the edit window, the elements of list can be edited like any other text. Outer brackets are not displayed.

When you exit the editor with <ESC>, `edlist` returns a list of `lists`. Each of the lists represents one line of the edit window.

On exiting the editor with <F10> and <A>, `edlist` does a `throw cancel`. If this is not caught then any editing you have done will be ignored.

Example

```
1: build 'wp

wp 'text
catch 'cancel [make :text edlist
        if valueq :text [value :text] [[]]]


1: wp 'story
```

This is a simple word processor! If `story` exists you can change its content. If it doesn't exist then `edlist` is called with an empty window. If you *cancel* the edit, no assignment is made to `story`.

Associated Primitive

`editlist`

eequalq *nwl1 nwl2*
eeqq
==

Remarks

Returns `'true` if *nwl1* and *nwl2* are exactly equal and `'false` otherwise.

- Two numbers are considered exactly equal if they differ by 1/2000000 or less.
- Words are considered equal only if they contain the same letters in the same order and in the same case.
- Lists are considered equal if their elements are exactly equal and in the same order.

If *nwl1* and *nwl2* are of different types the result will always be `'false`.

Examples

```
1: say eequalq 1 2
false

1: say eeqq 1 1
true

1: say 'HOUSE == 'house
false
```

Associated Primitive

`equalq`

either *a b*

Remarks

Returns `'true` if either or both its inputs are true and `'false` otherwise (the 'inclusive OR' function).

The table below shows how it works for different values of *a* and *b*.

a	*b*	either *a b*
`'false`	`'false`	`'false`
`'false`	`'true`	`'true`
`'true`	`'false`	`'true`
`'true`	`'true`	`'true`

If the first input is `'true` the second is not evaluated.

Example

The following procedure is useful when you want to check whether a user has typed a yes or no answer:

```
1: build 'verify.answer

verify.answer :x
result either :x ='yes :x ='no
```

Associated Primitives

`not, xor`

emptyq *nwl*
emq

Remarks

Returns `'true` if its input is the empty word (') or the empty list ([]) and `'false` otherwise. Numbers are never empty.

Examples

```
1: say emptyq 'fred
'false
1: say emptyq '
'true
1: say emq [Tom Joe]
'false
1: say emq bf bf [Tom Joe]
'true
```

`emptyq` is often used to test whether recursion can continue. For example:

```
1: build 'sum

sum :numbers
if emq :numbers [result 0]
result add first :numbers sum rest :numbers
```

Hence:

```
1: print sum [100 20 3]
123
```

end

Remarks

Used in parallel processing. **end** terminates only the process which executes it.

Please see chapter 13.

Associated Primitives

`await, begin, parallel, whenever`

equalq *nwl1 nwl2*
eqq
=

Remarks

Returns `'true` if *nwl1* and *nwl2* are equal and `'false` otherwise.

- Two numbers are considered equal if they differ by 1/2000000 or less
- Words are considered equal if they contain the same letters in the same order, irrespective of case
- Lists are considered equal if their elements are equal and in the same order.

If *nwl1* and *nwl2* are of different types the result will always be `'false`.

Examples

```
1: say equalq 1 2
false
1: say equalq 1 1
true
1: say equalq 'HOUSE 'house
true
1: say equalq [Oxford London] [Oxford Durham]
false
```

Associated Primitives

`eequalq`

erasefile *filename*

Remarks

Deletes the file *filename* from disk. Returns `'true` if the file was deleted, otherwise it returns `'false`.

Example

```
unless erasefile 'junk.dat [say
        [can/'t delete file]]
```

escape

Remarks

Stops all processes (unlike **end** which stops only the invoking process).

Associated Primitives

`end, stop`

eval *list*

Remarks

Treats the contents of *list* as a set of Logo expressions. Logo returns a list composed of the values of each expression.

If the list contains a procedure which does not return a result, an error is returned.

Examples

```
1: print eval [2+2 7-2 8/4]
[4 5 2]

1: make 'x [here is the news]
1: print eval [first :x rest :x]
[here [is the news]]
```

exp *number*

Remarks

Calculates the exponential function. **exp** returns e raised to the power of *number*.

Examples

```
1: say exp 4
54.5981500331442

1: print ln exp 5
5
```

explode *word*

Remarks

Returns a list made up out of the characters contained by *word*.

Example

```
1: print explode 'slough
[s l o u g h]
```

Associated Primitive

`implode`

expose *nwl*

Remarks

Allows you to recover or 'unbury' procedures from the workspace. It is the opposite of `bury`.

Primitives such as `forward` are buried, by default, when Logo is loaded and cannot be exposed. *expose* only allows you to get at procedures that have been buried.

Examples

```
1: say titles
rhombus
1: expose [square triangle hexagon]
1: say titles
rhombus square triangle hexagon
```

Associated Primitive

`bury`

fence

Remarks

Prevents the turtle from crossing the edge of the screen. If you try to make it cross the edge of the screen, an error message will be returned.

Crossing the edge of the screen will cause a `throw 'fence` which can be caught using `catch 'fence` (see chapter 5).

The turtle's field is normally 'unfenced' at start-up time.

Example

Using:

```
1: forever [catch 'fence [forever [fd 1]]
        seth heading + 180]
```

reverses the direction of the turtle every time it hits the edge.

Associated Primitives

`fenceq, nofence, nowrap, wrap, wrapq`

fenceq

Remarks

Returns `'true` if the turtle's field is 'fenced', otherwise it returns `'false`.

Associated Primitives

`fence, nofence, nowrap, wrap, wrapq`

fill *number1 number2*

Remarks

Fills the graphics plane with the pen colour, taking the current turtle position as the starting point. When a colour other than the background colour is encountered, it is treated as a boundary.

If *number1* is either 0 or 1, the area is filled with a solid colour (the current pen colour) and *number2* is ignored (but must be present).

If *number1* is 2, the area is filled with a pattern. *number2* can take the values 0 to 7 and each gives a different pattern.

If *number1* is 3, *number2* defines a hatching pattern (a bolder pattern). The range of numbers is 0 to 5.

Example

The following commands draw a square and fill it in as a brick wall:

```
1: repeat 4 [fd 80 lt 90]
1: lift lt 45
1: fd 10
1: setpc 2 fill 0 0
1: setpc 14 fill 3 5
```

first *nwl*

Remarks

Returns the first element of its input, which can be a number, a word or a list.

The input cannot be an empty word or an empty list.

Examples

```
1: say first 'wolf
w

1: say first [big bad wolf]
big

1: say first 12345
1
```

Associated Primitives

`butfirst, butlast, last`

forever *list*

Remarks:

Repeats the command *list* forever. You can stop the command by pressing <ESC>, encountering `end`, `stop`, `escape` or using `throw`.

Examples

```
1: forever [forward 1 right 1]
```

Pairing `forever` with `catch` can be very useful. For example to print the file `sales` on the screen:

```
1: catch "endfile [forever [type rfc "sales]]
```

Associated Primitive

`repeat`

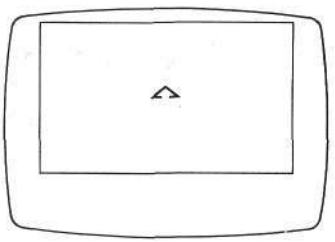
forward *number*
fd

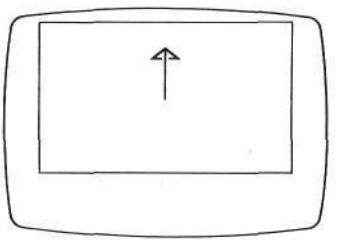
Remarks

Moves the turtle *n* steps in the direction of its current heading. If the turtle's pen is down, the turtle leaves a trace of its path. If *number* has a negative value, the turtle moves backwards.

Example

```
1: cs
1: forward 50
```





Associated Primitive

`backward`

frac *number*

Remarks

Returns the fractional part of *number*.

Example

```
1: print frac exp 1
0.71828172845904
```

gc

Remarks

`gc` stands for garbage collector. It allows Logo to re-use all of the workspace that is no longer required. Logo automatically performs a garbage collection when it begins to run out of unused workspace. This can be observed when the turtle is drawing by a brief halt in the turtle's movement.

Associated Primitive

`nodes`

goodbye
exit

Remarks

Deletes extensions, variables and procedures. Closes dribble file, clears text and graphics and returns to the operating system.

goto *word*

Remarks

Transfers control to a different line within the same program. The line to which control is transferred must begin with a *tag*. A *tag* is a word; optionally with or without a beginning quote mark, followed by colon dash (:-).

Example

This procedure removes one item from a list. The `goto` is completely redundant; it just illustrates how to use the primitive.

```
1: build 'remove

remove 'x 'list
if emptyq :list [qoto 'error]
result if first :list = :x [rest :list]
       [first :list +> remove :x rest :list]
error :—say :x +> [is not in the list] escape
```

It is considered bad programming style to use `goto` statements, but they are occasionally useful in error checking.

greaterequalq *nw1 nw2*
grq
>=

Remarks

Returns `'true` if *nw1* is greater than or equal to *nw2*, otherwise it returns *'false'*.

- Two numbers are considered equal if they differ by 1/200000 or less.
- Words are compared in dictionary order A...Z, irrespective of case.

The inputs *nw1* and *nw2* must be of the same type.

Examples

```
1: if :number >= 0 [panic]
1: if grq 5 2 [say [5 is bigger than 2]]
```

Associated Primitive

```
greaterq
```

greaterq *nw1 nw2*
geq
>

Remarks

Returns `'true` if *nw1* is greater than *nw2*.

- The numbers are considered equal if they differ by 1/2000000 or less
- Words are compared in dictionary order A..Z, irrespective of case.

nw1 and *nw2* must be of the same type.

Example

The following command line tells the turtle that if `distance` is greater than 10 it is to go forward 10 steps, otherwise it is to go forward `distance` steps:

```
1: forward (if :distance > 10 [10] [:distance])
```

Associated Primitive

`greaterequalq`

grievance

Remarks

Returns the text of the message given by Logo in reply to the most recent error.

Example

```
1: say grievance
```

Associated Primitive

`moan`

heading

Remarks

Returns the turtle's current drawing heading (not its movement heading).

Examples

```
1: cs
1: print heading
0

1: left 90
1: print heading
270
```

The following procedure draws a circle and prints the turtle's heading after each step:

```
1: build 'circle

circle
repeat 36 [fd 10 lt 10 print heading]
```

Associated Primitive

```
seth
```

hideturtle
ht

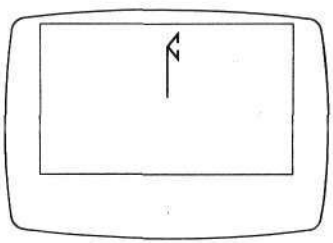
Remarks

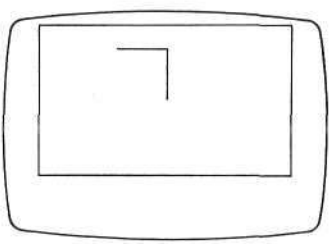
Makes the turtle invisible. The turtle continues to draw and obey commands.

You can make the turtle visible again using `showturtle`.

Example

```
1: cs
1: forward 50
1: left 90
1: hideturtle
1: forward 50
```





Associated Primitive

`showturtle`

if *a list1*
if *a list1 list2*

Remarks

In the first form shown above, if the expression *a* is `'true` the command *list1* is executed, otherwise the next line is executed.

In the second form, if the expression *a* is `'true` the command *list1* is executed; if *a* is `'false`, the command *list2* is executed.

In both forms, if *list1* or *list2* produce a result, this will be passed back as the result of the `if` statement.

Examples

The following example is a procedure which allows the computer to make a decision (yes or no) for you. Three versions are given, each using `if` in a different way.

`if` used to control execution:

The procedure `decision` using `if` with one list:

```
1: build 'decision

decision
if pick 2 = 1 [result 'yes]
result 'no

1: say decision
yes
```

The same procedure `decision` using `if` with two lists:

```
1: edit decision
if pick 2 = 1 [result  'yes] [result  'no]

1: say decision
'no
```

Finally, `if` used to return a result:

```
1: edit  'decision
result if pick 2 = 1 ['yes] ['no]

1: say decision
'no
```

implode *list*

Remarks

Returns the word made by concatenating (joining) all the words in its input list.

`implode` is the opposite of `explode`.

Example

```
1: print implode [s l o u g h]
slough
```

Associated Primitive

`explode`

infile *filename*

Remarks

Opens the file *filename* for input. Returns `'true` if the file is opened successfully and `'false` otherwise.

Example

The following command tries to open the file `datafile` for input. If `datafile` is open it prints the message `'input file already open'` and stops.

```
1: unless infile 'datafile.dat
        [say [input file already open] escape]
```

Associated Primitives

`closefile, infiles, readfilec, readfiled, readfilel`

infiles

Remarks

Returns a list of the names of files open for input.

Example

```
1: print infiles
[datafile1 datafile2]
```

Associated Primitives

```
closefile, infile
```

int *number*

Remarks

Returns the integer value of *number*. Any decimal part is truncated for both positive and negative values of *number*. You can make sure of rounding numbers up by adding 0.5 as shown in an example below.

Examples

```
1: print int -2.3
-2
1: print int exp 1
2
1: print int 44.6
44
1: print int (44.6 + 0.5)
45
```

join *wl1 wl2*

++

Remarks

Returns a word or list by joining the first and second inputs.

Inputs to `join` cannot be numbers and they must both be of the same type.

Examples

```
1: print join [the owl] [and the pussycat]
[the owl and the pussycat]

1: print join 'pussy 'cat
'pussycat

1: print [a b] ++ [c]
[a b c]
```

Associated Primitives

`putfirst, putlast, sentence`

key

Remarks

Delays the calling process until a key is struck and then returns the value of this key without echoing it to the screen. Digits are returned as a Logo number; other characters are returned as one-character words.

If Logo is reading a command file, `key` returns the next character read from the keyboard and not from the command file.

Please see Chapter 7 for a detailed explanation of using `key`.

Special characters returned by `key` will be displayed as an `escape` sequence if you use `print`.

Example

If you press <CTRL> and G together after `print key`, the escape sequence will be displayed as:

```
1: print key
'\07
```

Associated Primitives

`keyq, readlist`

keyq

Remarks

If a key has been struck, `keyq` returns `'true` and the key can be read using `key`. `'false` is otherwise returned. If Logo is reading a command file, `keyq` will still test the keyboard.

Examples

The following procedures make the turtle move forward continuously and let you use the <L> and <R> keys to change its direction:

```
1: build 'move

move
forever [if keyq [check.key key] [forward 1]]

1: build check.key

check.key 'button
make 'button lowercase:button
if :button = 'l [left 10]
if :button = 'r [right 10]
```

Associated Primitive

`key`

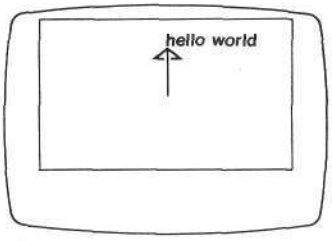
label *nwl*

Remarks

Prints the text *nwl* near to the turtle. If the current process is addressing more than one turtle, then each will print *nwl*. The label is printed in the current turtle pen colour. The turtle is not moved by the label.

Example

```
1: cs
1: forward 50
1: label [hello world]
```



last *nwl*

Remarks

Returns the last element of its input. The input must not be empty.

If the input is a decimal number, the digit before the decimal point is returned. If an integer is input then the last digit is returned.

The last letter of a word is returned, and the last item in a list.

Examples

```
1: say last 34
4
1: say last 45.6
5
1: say last [humpty dumpty sat on a wall]
wall
1: say last 'humpty
y
```

Associated Primitives

`butfirst, butlast, first`

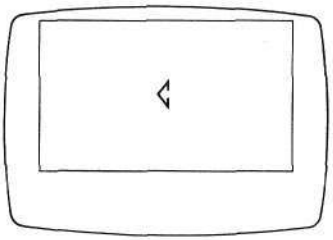
left *angle*
lt

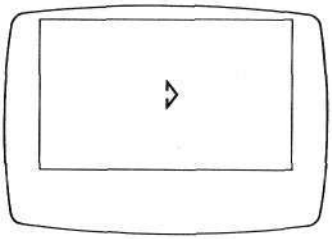
Remarks

Turns the turtle left (anticlockwise) by *angle* degrees. If *angle* has a negative value, the turtle will turn right (clockwise).

Examples

```
1: cs
1: left 90
1: cs
1: left -90
```





Try the following:

```
1: dsh [box [-8 -8][8 -8][8 8][-8 8][-8 -8]]
1: setshape 'box
1: repeat 360 [left 1]
```

Associated Primitive

```
right
```

lessequalq *nw1 nw2*
leq
<=

Remarks

Returns `'true` if the first input is less than or equal to the second, otherwise it returns `'false`.

- Two numbers are considered equal if they differ by 1/200000 or less.
- Words are compared in dictionary order A...Z, irrespective of case.

The inputs *nw1* and *nw2* must be of the same type.

Examples

```
1: say lessequalq 1 3
true
1: say lessequalq 6 2
false
1: say 3 <= 3
true
1: say 5 <= 3
false
1: say 'peter <= 'cathy
false
1: say 'cathy <= 'peter
true
```

Associated Primitive

`lessq`

lessq *nw1 nw2*

Remarks

Returns `'true` if *nw1* is less than *nw2*.

- Two numbers are considered equal if they differ by 1/2000000 or less.
- Words are compared in dictionary order A...Z, irrespective of case.

Examples

```
1: say lessq 1 3
true
1: say lessq 6 2
false
1: say lessq 4.00001 4.00002
true
```

Associated Primitive

`lessequalq`

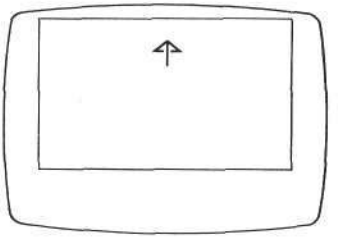
lift

Remarks

Lifts the turtle's pen off the paper so that the turtle doesn't draw when it moves. This primitive is the opposite of `drop`. It is also used in defining the turtle shape (see `defineshape`).

Example

```
1: cs
1: lift
1: fd 20
1: drop
1: fd 20
```



Associated Primitives

`drop, upq`

line *list1 list2 number*

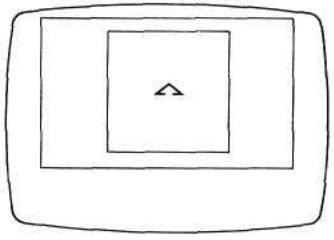
Remarks

Draws a line in colour *number* between the points *list1* and *list2*. *list1* and *list2* are in the form of coordinate pairs.

Example

The following commands draw a square on the screen:

```
1: line [-80 80] [80 80] 3
1: line [80 80] [80 -80] 3
1: line [80 -80] [-80 -80] 3
1: line [-80 -80] [-80 80] 3
```



Associated Primitive

`setpoint`

listq *nwl*

Remarks

Returns `'true` if *nwl* is a list and `'false` if it is not.

Examples

The following procedure, `check`, checks if its input is a list. If so, it prints `'list'`, otherwise it prints `'not a list'`.

```
1: build 'check

check 'object
if listq :object [say [list]] [say [not a list]]

1: make 'towns [Durham Oxford London]
1: make 'town 'Exeter

1: check :towns
list

1: check :town
not a list
```

Associated Primitive

`numberq, wordq`

ln *number*

Remarks

Returns the natural logarithm (log to base e) of *number*.

number must be greater than zero.

Examples

```
1: ln 0.5
-0.69314718055995

1: ln 20
2.99573227355399
```

Associated Primitive

exp

local *wl*

Remarks

Creates one or more new variables within a procedure. These variables can only be used by the procedure which generates them and by any procedures called by the generating procedure.

If a local variable is given the same name as a global variable, the local variable is used until the end of the procedure which created it. The global variable is then available again.

Examples

```
local [p q r]
```

creates local variables called p, q and r.

The following example solves quadratics, giving you a list of all solutions.

```
1: build 'solve.quad

solve.quad 'a 'b 'c
local 'd
'd <-(:b * :b -4 * :a * :c)
result branch :d < 0 [[]] case :d = 0
        [eval [-:b/2*:a]] default
        [eval [(-:b+sqt:d)/(2*:a)
        (-:b-sqt:d)/(2*:a)]]

1: print solve.quad 1 7 12
[-3 -4]
1: print solve.quad 1 (-8) 16
[4]
```

log *number*

Remarks

Returns the logarithm (log to the base 10) of *number*.

number must be positive and greater than 0.

Examples

```
1: log 0.5
-0.30102999566398

1: log 20
1.30102999566398
```

lowercase *nwl*

Remarks

Returns the word or list made by converting every alphabetic character in its input to lower case. There is no effect on a number.

Example

```
1: say lowercase [LOGO system]
logo system
```

Associated Primitive

`uppercase`

made

Remarks

Returns a list of names of all global variables and variables known to this process.

Example

```
1: say made
side angle number
```

Associated Primitives

`make, unmake`

make *word nwl*
<-

Remarks

Creates a variable called *word* and gives it the contents of *nwl*.

You can get at the contents of the variable using `value` or a colon (`:`).

If `make` is used in a procedure and *word* doesn't already exist, then the resulting variable is global. It will remain in Logo's workspace unless you use `unmake` on the *word*.

Examples

```
1: make "angle 90
1: say :angle
90

1: "side <-100
1: say value :side
100
```

Associated Primitives

`local, made, unmake, value, valueq`

moan

Remarks

Reproduces the last error. All processes are stopped.

You can catch `'error` when `moan` is run; if you do then no message is printed and the process that caught the error continues after the catch.

Associated Primitives

`grievance`

multiple

Remarks

Turns parallel processing back on after `single` has been used. `multiple` is the default state.

Associated Primitive

`single`

multiply *number1 number2*
mul
*

Remarks

Returns the product of its inputs.

Examples

```
1: print multiply 20 8
160

1: make 'x 13
1: say multiply :x 3
39

1: say 10.5 * 10
105
```

near

Remarks

Returns a list of the turtles which are within eight screen units of the centre of the current turtle.

If your process is addressing more than one turtle, `near` refers to the lowest numbered one (normally the earliest one created).

If any turtle is sensing and `throw "touchturtle` occurs, then `near` can be used to find out which turtle(s) caused the `throw`.

Example

```
1: cs
1: tell [1 2 3]
1: print near
[2 3]
```

nodes

Remarks

Returns a number indicating the amount of unused memory in Logo's workspace.

For a more accurate measure of free work space, use `gc` before `nodes`.

Logo stores procedures and variables using complex rules. It is very difficult to give an easy method to find out how much workspace is occcupied by any application.

Associated Primitive

`gc`

nodribble

Remarks

When the *dribble* primitive is used, everything you type in is written to a command file. *nodribble* closes the command file and your typing stops being recorded. You can replay the command file using `consult` or `replay`.

Associated Primitives

`consult, dribble, dribbleq, replay`

nodriver

Remarks

Unloads any floor turtle driver which is present in Logo.

Associated Primitive

`driver`

nofence

Remarks

When the `fence` primitive has been used, an error is reported if the turtle hits the edge of the screen.

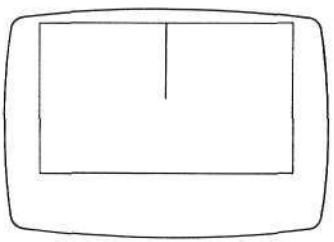
`nofence` removes this error reporting and lets the turtle move off the screen.

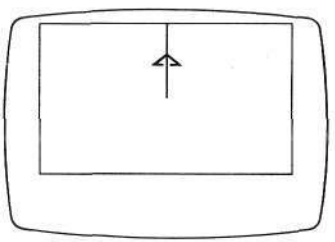
Examples

```
1: cs
1: fence
1: forward 1000
```

produces an error message on the screen because the turtle has hit the 'fence' around the screen. However, `nofence` will let the turtle out of the visible screen area.

```
1: cs nofence
1: fd 1000
1: back 950
```





Associated Primitives

`fence, fenceq, nowrap, wrap, wrapq`

nosense

Remarks

Switches off the sensing for all turtles controlled by a particular process.

When turtles are sensing, a signal is thrown if they go near another turtle, go into a different background colour or hit the edge of the screen. Sensing is time consuming, and it is advisable to turn sensing off with `nosense` whenever you don't need it.

Associated Primitive

`sense`

not *a*

~

Remarks

Returns `'true` if its input is `'false` and `'false` if its input is `'true`.

Examples

```
1: print not (1 = 1)
false

1: print not (1 = 2)
true

1: print ~ 'false
'true
```

Associated Primitives

`both, either, xor`

nowrap
nofence

Remarks

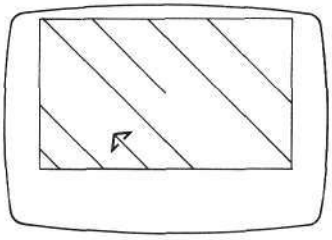
Allows the turtle to move off the screen without appearing on the opposite edge.

If the turtle moves off the screen following a `wrap` command, then it reappears at the opposite side of the screen. `nowrap` lets the turtle continue to move off the screen without reappearing on the opposite side of the screen.

`nofence` can be used interchangeably with `nowrap`.

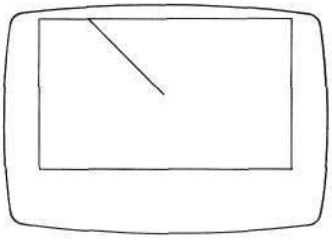
Examples

```
1: cs
1: wrap
1: left 45
1: forward 1000
```



shows the turtle wrapping around the screen and the following shows the turtle doing the same action without wrapping.

```
1: cs
1: nowrap
1: left 45
1: forward 1000
```



Associated Primitives

`fence, fenceq, wrap, wrapq`

numberq *nwl*

Remarks

Returns `'true` if *nwl* is a number and `'false` otherwise.

Example

The following procedure tests if its input is a number:

```
1: build check.no

check.no 'object
1: say putfirst :object if numberq :object
        [[is a number]]
        [[is not a number]]

1: make 'item1 1234
1: checkno :item1
1234 is a number

1: make 'item2 [1 2 3 4]
1: checkno :item2
[1 2 3 4] is not a number
```

objects

Remarks

Returns a list of all the Logo words which have had properties assigned to them using `assert`.

Examples

```
1: assert 'whiskers 'species 'cat
1: assert 'patch 'species 'dog
1: assert 'patch 'colour [black and white]
1: objects
[patch whiskers]
```

Associated Primitives

`assert, asserted, assertedq, assertions, classified, deny`

outfile *filename*

Remarks

Creates the file *filename* and opens it for output. Any existing file with the same name is deleted.

Returns `'true` if the file is successfully opened and `'false` otherwise.

Example

The following command tries to open `datafile` as an output file. If it is successful, the program continues, otherwise it prints the message `'output file already open'` and stops.

```
if not outfile 'datafile [say [output file
        already open] stop]
```

Associated Primitives

`appfile, closefile, outfiles, writefilec, writefiled, writefilel`

outfiles

Remarks

Returns a list of the names of files currently open for output.

Example

```
1: print outfiles
[data1.dat data2.dat]
```

Associated Primitives

```
closefile, outfile
```

parallel *list*

Remarks

The commands given in the list of lists are all started as separate processes. The process that issued the `parallel` command is suspended until they have all finished.

Unless `catch` catches it, an error occuring in any process will terminate all other processes running at that time.

Example

```
parallel [[monitor.keyboard] [move.robot]]
```

A further example of parallel processing is provided in the file `cage.def` on the RM Logo disk.

Associated Primitives

`begin, run`

pc

Remarks

Returns the colour of the turtle's pen. If more than one turtle is active, then the pen colour of the first turtle created is returned.

Examples

```
1: print pc
1
1: setpc 2
1: print pc
2
```

Associated Primitive

setpc

pennormal

Remarks

Lowers the turtle's pen and changes the drawing style so that existing lines are overdrawn when the turtle moves.

`pennormal` is used to cancel the effect of a `penreverse`. The `penreverse` primitive makes the turtle erase existing lines to appear to be drawing over them (known as XOR drawing).

Associated Primitives

`penreverse, reverseq`

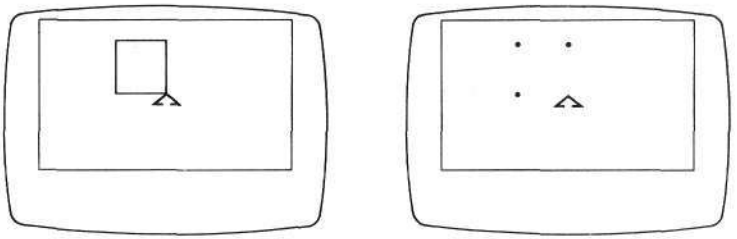
penreverse
px

Remarks

Lowers the turtle's pen. When the turtle moves, it draws lines where there are none, and erases existing lines in the same colour that it crosses over.

Example

```
1: cs
1: repeat 4 [fd 40 lt 90]
1: penreverse
1: repeat 4 [fd 40 lt 90]
```



The corner dots remain because each corner is plotted twice: once when the turtle arrives; once when it leaves.

Associated Primitives

`pennormal, reverseq`

pi

Remarks

Returns the value of pi (3.142...) accurate to the full precision of the internal arithmetic.

Examples

```
1: print pi
3.1459265358979

1: print 22/7 -pi
0.00126448926735
```

The following procedure can be used to calculate the area of a circle, given its diameter:

```
1: build 'area.circle

area.circle 'radius
result pi * (:radius ↑2)
```

pick *number*

Remarks:

Returns a pseudo-random integer in the range 1 to *number* inclusive.

Example

The following example simulates the throwing of a die:

```
1: print pick 6
5
1: print pick 6
3
```

Associated Primitives

`random`

po *wl*

Remarks

Prints out the definition of the procedures named in *wl* on the screen.

A library procedure `copy` is provided on your RM logo disk which will list them to a printer.

Example

```
1: po titles
```

lists the definition of all procedures known to Logo which are neither buried nor primitives.

point *list*

Remarks

Returns a number which tells you which colour the point *list* is currently painted. *list* is a coordinate pair.
The significance of colour numbers is described in Chapter 2.

Examples

```
1: setpoint [40 40] 3
1: say point [40 40]
3
```

Associated Primitive

```
setpoint
```

power *number1 number2*

Remarks

Raises *number1* to the power of *number2*.

Examples

```
1: power 2 3
8
1: power 3 2
9
```

print *nwl*

Remarks

Prints the contents of *nwl* on the screen.

Lists are printed with their outermost brackets. Elements of a list are separated by a space.

Logo special characters whose ASCII codes are within the (printable) range 20–7E hexadecimal are printed prefixed by the escape character \ (characters whose ASCII codes are in the range 01 to 1F and 7F to FF hexadecimal are printed as two digits prefixed by \).

Examples

```
1: print [Logo rules ok]
[Logo rules ok]
1: print 'elephant
'elephant

1: print '\1b\E
'\1b\0e

1: print '\21 ; this is not special to Logo
'!

1: print '\23 ; this is special to Logo
'\#
```

Associated Primitives

say, type

putfirst *nwl list*
pf
+>

Remarks

Returns the list which is produced by putting *nwl* at the front of *list*.

Examples

```
1: print putfirst 2 [3 4]
[2 3 4]

1: print putfirst [the owl] [and the pussycat]
[[the owl] and the pussycat]

1: print 'x +> [y z]
[x y z]
```

Associated Primitives

`putlast, join, sentence`

putlast *list nwl*
pl
<+

Remarks

Returns the list which is produced by putting *nwl* at the end of *list*.

Examples

```
1: print putlast [3 4] 5
[3 4 5]

1: print putlast [the owl and] [the pussycat]
[the owl and [the pussycat]]

1: print [x y] <+ 'z
[x y z]
```

Associated Primitives

`putfirst, join, sentence`

random

Remarks

Returns a random decimal fraction between 0 and 1.

Examples

```
1: print random
0.050603
```

The following routine returns a pseudo-random number drawn from a Gaussian distribution of `mean` and standard deviation (`sd`).

```
1: build 'gauss

gauss 'mean 'sd
local 't
't <-0
repeat 12 ['t <-:t + random]
result :mean + (:t -6) * :sd

1: print gauss 100 16
98.654...
```

readfilec *filename*
rfc

Remarks

Reads the next character from the named file and returns a one-character word. The file must have been opened for input.

If there is no more data in the file, `readfilec` throws `'endfile`. If this is not caught, it returns the word `'endfile` as its result.

Examples

```
1: make 'data readfilec 'myfile.dat
```

will read the next character from the file `myfile.dat`

An example procedure `listfile`, which prints the contents of a file on the screen, is provided on the RM Logo disk as `listfile.def`.

Associated Primitives

`closefile, infile, infiles, readfiled, readfilel`

readfiled *filename*
rfd

Remarks

Reads the next Logo data item (a number word or list) from the named file and returns it. The file must have been opened for input.

If there is no more data in the file, `readfiled` throws `'endfile`. If this is not caught, it returns the word `'endfile` as its result.

Example

```
1: make 'data readfiled 'myfile.dat
```

Associated Primitives

`closefile`, `infile`, `infiles`, `readfilec`, `readfilel`

readfilel *filename*
rfl

Remarks

Reads the next line from the named file and returns a Logo list. The file must have been opened for input.

If there is no more data in the file, `readfilel` throws `endfile`. If this is not caught, it returns the word `'endfile` as its result.

Example

```
1: make 'data readfilel 'myfile.dat
```

Associated Primitives

`closefile`, `infile`, `infiles`, `readfilec`, `readfiled`

readlist
rl

Remarks

Reads information typed at the keyboard until you press <ENTER>, then returns what you typed as a Logo list.

Examples

```
1: build 'question

question
say [name one colour in a traffic light]
if amongq readlist [[red] [yellow] [amber]
        [green]] [say [correct!] stop]
        [say [no, try again]]
question

1: question
name one colour in a traffic light
purple
no, try again
name one colour in a traffic light
green
correct!
```

Associated Primitives

`ask, key, keyq`

remainder *number number*

rem

%

Remarks

Returns the remainder that results from dividing the first input by the second (the quotient can be found by using the primitive `share`).

Examples

```
1: print remainder 13 2
1
1: print remainder 12 2
0
1: print 41 % 10
1
```

Associated Primitive

`share`

rename *word1 word2*

Remarks

Renames the procedure *word1*, giving it the new name *word2*. The old name is lost. You can't rename a primitive or a buried procedure.

Example

```
1: rename 'polygon 'six.sided.figure
```

Associated Primitive

`alias`

renamefile *filename1 filename2*

Remarks

Renames the file *filename1* to *filename2*. Returns `'true` if the operation succeeded and `'false` otherwise.

Example

```
unless renamefile 'turtles1 'turtles2 [say [cannot
        rename your file] escape]
```

Associated Primitives

`directory, erasefile`

repeat *integer list*

Remarks

The list of commands *list* is repeated *integer* times. *integer* must be zero or positive. If *integer* is zero then the list is not run.

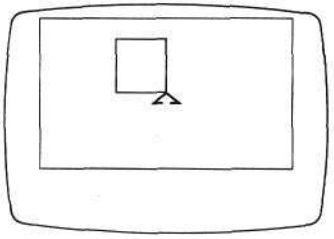
Examples

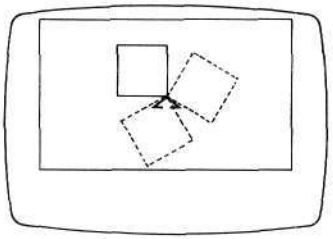
The following primitives draw a square:

```
1: cs
1: repeat 4 [forward 50 left 90]
```

The following example draws some 'spinning squares'. They use the following procedure:

```
1: build 'square

square
repeat 4 [fd 50 lt 90]

1: cs
1: repeat 3 [square lt 120]
```





Associated Primitives

`do, forever, unless, while`

replay *filename*

Remarks

Replays a sequence of commands held in the file *filename*. The commands are displayed on the screen and the prompt changes from `1:` to `1<`.

Even if an error occurs in the file, the commands continue to be replayed.

Example

```
1: replay 'turtles1.lgc
```

Associated Primitives

`consult, dribble, dribbleq`

result *nwl*

Remarks

Stops the procedure in which it occurs and passes control back to the procedure or command which called it. It returns the value *nwl* to this procedure or command.

Examples

The following procedure calculates the average of two numbers:

```
1: build 'average

average 'number1 'number2
result (:number1 + :number2)/2

1: say average 5 15
10

1: say average 10 15
12.5
```

reverseq

Remarks

Returns `'true` if the turtle's pen is plotting in reverse mode. `penreverse` or `px` cause the turtle pen to plot in reverse or exclusive OR mode.

If the pen is lifted or is plotting normally, it returns `'false`.

Associated Primitives

`pennormal, penreverse`

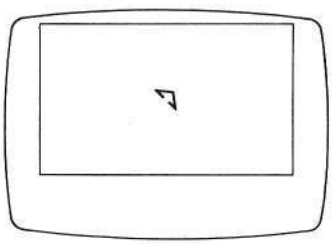
right *angle*
rt

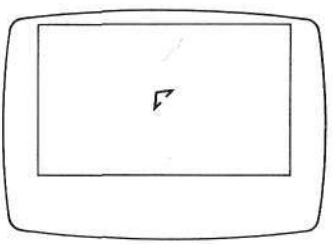
Remarks

Turns the turtle right (clockwise) by *angle* degrees. If *angle* has a negative value, the turtle turns to the left (anticlockwise).

Examples

```
1: cs
1: right 45
1: cs
1: right -45
```





Associated Primitive

`left`

rubber
eraser

Remarks

Selects the pen colour which matches the background. The turtle rubs out any lines it crosses.

The line-drawing algorithm used means that `forward` and `backward` do not necessarily trace exactly the same screen path. For perfect rubbing out, you should start in the same place and retrace your steps exactly.

`rubber` is equivalent to `setpc bg`.

Example

```
1: cs
1: forward 20
1: centre
1: rubber
1: forward 20
```

run *list*

Remarks

Executes the commands in *list*. If any command in *list* returns a value, this value is also returned by `run`, and commands after that are not executed.

Examples

The following procedure executes whatever you type at the keyboard providing it produces a result. `forward 50` will give an error because it doesn't produce a result!

```
1: build run.input

run.input
say run readlist
run.input

1: run.input
?10 + 5
15

?5 * 4
20

?10 = 5 * 3
false
```

say *nwl*

Remarks

Prints the contents of *nwl* on the screen, followed by a carriage return.

`say` prints lists without their outermost brackets.

`say` also sends special characters to the screen.

Examples

```
1: say 1 + 3
1 + 3

1: say [hello there]
hello there
```

The following command appears equivalent to `ts` and will reset and clear the screen:

```
1: say '\1bc
```

However, you should use such calls with care and, if it exists, use the equivalent in the Logo language.

Associated Primitives

`print, type`

scrap *wl*

Remarks

Procedures named by *wl* are deleted from memory. If *wl* is a list and one of the procedures named does not exist, none will be scrapped.

The following is a special case which scraps everything that isn't buried:

```
scrap titles
```

Examples

```
1: scrap 'triangle

1: scrap [square triangle]
```

sense

Remarks

Makes the turtle sensitive to touching other parts of the graphics screen.

If the turtle is about to move onto a different background colour, it will stop and do a `throw 'touch`.

If the turtle goes near another turtle or hits the edge of the screen, it will stop and do a `throw 'touchturtle`.

`fence` and `catch 'fence` work independently of the turtle's sensing or not sensing. The `throw` will direct control to the process which gave the current turtle the most recent `sense` command. See Chapters 5 and 12 for details of how to use `throw`.

`sense` is time consuming, you should cancel it with `nosense` when you don't want it in use any more.

Associated Primitives

`fence, nofence, nosense`

sentence *nwl1 nwl2*

se

&&

Remarks

Makes a list out of *nwl1* and *nwl2*. If they are both lists, `sentence` returns a list made by joining them. If they are both words, it puts them both into a list. If only one is a list, `sentence` includes the other *nwl* in it.

Examples

```
1: print sentence 'cats [are great]
[cats are great]

1: print se [cats dogs] [rabbits hamsters]
[cats dogs rabbits hamsters]

1: print 'butter && 'flies
[butter flies]
```

Associated Primitives

`join, putfirst, putlast`

setbg *number*

Remarks

Changes the screen background colour to *number*.

The command will take effect from the next `clean` or `clearscreen` command, and will be ignored if the screen is in text only mode. The numbers associated with colours are listed under the description of the `colour` primitive.

Example

```
1: setbg 3
1: cs
```

Associated Primitive

bg

setc *number*

Remarks

Changes the colour in which the turtle is painted on the screen to *number*.

The numbers associated with colours are listed under the description of `colour`.

This colour is not necessarily the same as the turtle's pen colour. Nor will the turtle always appear as the colour you specify it to be. The turtle is plotted in exclusive OR mode (mixing colours where they coincide on screen) so you must consider the background colour when choosing the turtle colour. A black background however, will guarantee an accurate turtle colour.

Examples

```
1: print colour
3
1: setc 1
1: print colour
1
```

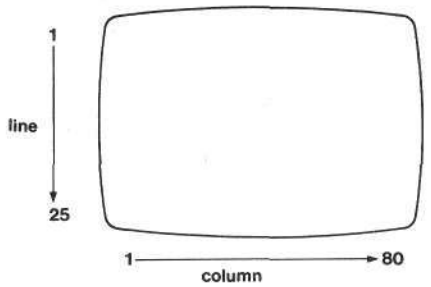
Associated Primitive

`colour`

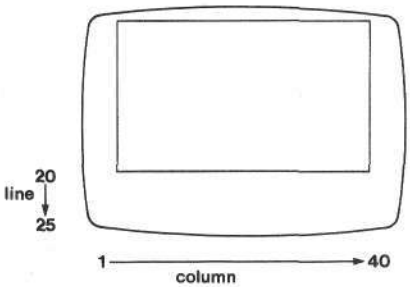
setcursor *list*

Remarks

Moves the cursor to the position given by *list*. The first element of *list* is the screen line and the second, the screen column, using the following convention:



text mode



graphics mode

Out-of-range values produce incorrect displays on screen.

Example

```
1: setcursor [20 16]
```

Associated Primitive

```
cursor
```

setdir *angle*

Remarks

Sets the turtle's movement heading to *angle*. The drawing heading is not affected.

Examples

The following commands move the turtle at an angle of 45 degrees while keeping it pointing northwards:

```
1: cs
1: tell 1
1: setdir 45
1: setspeed 20
```

Associated Primitive

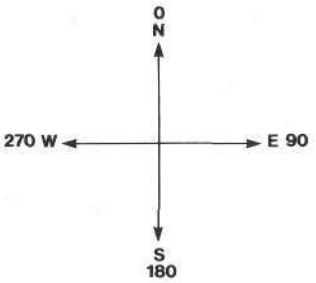
`dir`

seth *angle*

Remarks

Sets the turtle's drawing heading to *angle* degrees. The turtle shape turns to show its new direction, but the movement heading is not affected.

Headings increase clockwise from 0, as shown below. Negative values of *angle* make the turtle turn in an anticlockwise direction.



Examples

```
1: cs
1: seth 120
1: cs
1: seth -120
```

Associated Primitive

`heading`

setpc *number*

Remarks

Sets the turtle's pen colour to *number*. The values of *number* and their related colours, are given in Chapter 2 and listed under the primitive `colour`.

The pen colour can be different to the turtle's colour. The colour may also be applied in either a 'true' or 'reversed' mode depending upon whether or not `penreverse` has been used.

Example

```
1: setpc 2
```

Associated Primitive

`pc`

setpoint *list number*

Remarks

Plots a point in colour *number* at the position given in *list*.

list is a coordinate pair.

Example

```
1: cs
1: setpoint [50 0] 3
```

Associated Primitive

```
point
```

setpos *list*

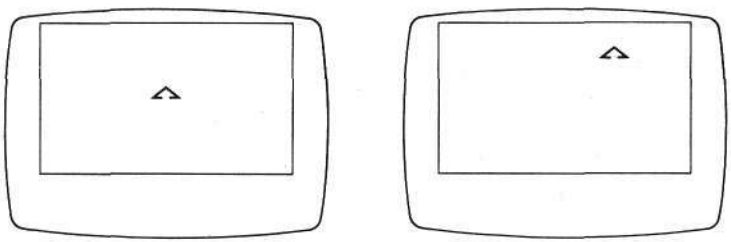
Remarks

Sets the turtle's x and y coordinates to the values given by *list*. If this moves the turtle outside the graphics area of the screen, and `fence` has been used, a `throw "fence` occurs.

The turtle does not draw as it moves.

Examples

```
1: cs
1: setpos [50 50]
```



Associated Primitives

`pos, setx, sety, xcor, ycor`

setshape *word*

Remarks

The turtle assumes the shape named by *word*. The shape must have previously been defined in a `defineshape` command.

`shapes` can be used to find out the defined turtle shapes.

Example

The shape of an arrow can be assumed by the turtle by first defining the arrow and then assigning the arrow shape to the turtle.

```
1: say shapes

1: defineshape [arrow [-8 -8][0 0][8 -8]
        lift [0 0][0-12]]
1: setshape 'arrow
```

Associated Primitives

`bdefineshape, shape, shapedef, shapes`

setspeed *number*

Remarks

Changes the turtle's movement speed to *number*.

If *number* exceeds 100, there is no change to the turtle's speed. *number* may be negative (down to –100).

The turtle moves continuously in the direction given by its movement heading but does not draw as it moves. (Remember the movement heading is different from the drawing heading.)

Example

```
1: setspeed 5
```

Associated Primitives

`dir, setdir, speed`

setx *number*

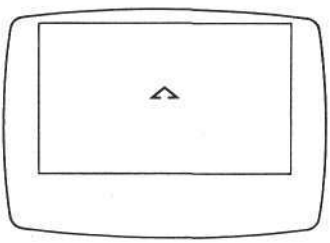
Remarks

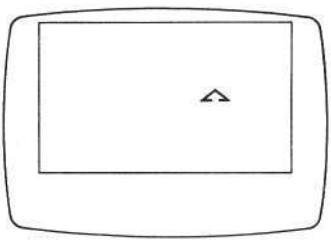
Sets the turtle's x coordinate to *number*. If this moves the turtle outside the graphics area of the screen, and fence has been used, a `throw "fence` occurs.

The turtle does not draw as it moves.

Example

```
1: cs
1: setx 50
```





Associated Primitives

`setpos, sety, xcor, ycor`

sety *number*

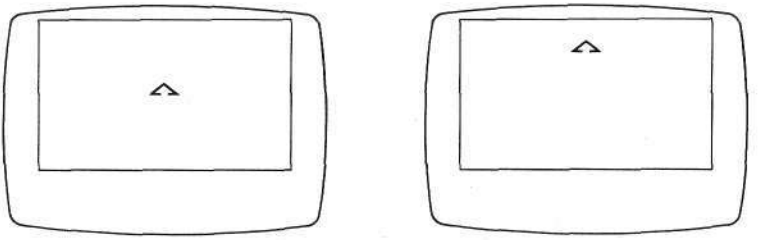
Remarks

Sets the turtle's y coordinate to *number*. If this moves the turtle outside the graphics area of the screen, and `fence` has been used, a `throw "fence` occurs.

The turtle does not draw as it moves.

Example

```
1: cs
1: sety 50
```



Associated Primitives

`setpos, setx, xcor, ycor`

shape

Remarks

Returns the name of the turtle's shape. If more than one turtle is receiving commands from this process, the result refers to the most recently created turtle.

If the turtle's shape is the default shape, `shape` returns the empty word.

Example

```
1: cs
1: print shape

1: setshape 'bicycle
1: print shape
'bicycle
```

Associated Primitives

`setshape, shapedef, shapes`

shapedef *word*

Remarks

Returns the definition of the shape *word*.

Example

```
1: dsh [box[-8 -8][8 -8][8 8][-8 8][-8 -8]]
1: print shapedef 'box
[box[-8 -8][8 -8][8 8][-8 8][-8 -8]]
```

Associated Primitives

`setshape, shape, shapes`

shapes

Remarks

Returns a list of the shapes that Logo knows about at that time. This doesn't include the default shape.

Examples

If you have started Logo with the files supplied by Research Machines (including the start-up file `start.lgc`) then:

```
1: print shapes
```

produces:

```
[bicycle car train coach]
```

Associated Primitives

`setshape, shape, shapedef`

share *number1 number2*
//

Remarks

Returns the integer quotient after *number1* has been divided by *number2*. The remainder can be found using `remainder`.

Examples

```
1: print share 6 3
2

1: print share 6 4
1

1: print 11 //2
5

1: print eval ['answer share 6 4
       'remainder rem 6 4]
[answer 1 remainder 2]
```

Associated Primitive

`remainder`

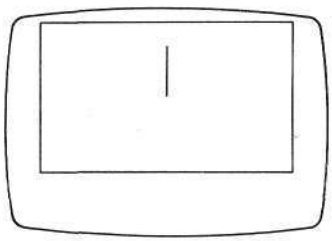
showturtle
st

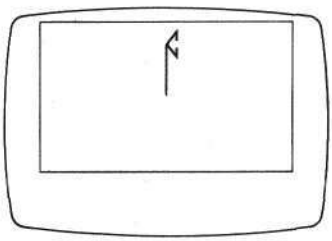
Remarks

Makes the turtle visible. This is the opposite of `hideturtle`.

Examples

```
1: cs
1: ht
1: forward 50
1: left 90
1: st
```





Associated Primitive

`hideturtle`

sin *angle*

Remarks

Returns the sine of *angle*, which is specified in degrees.

Examples

```
1: print sin 30
0.5
```

Associated Primitives

`cos, tan`

single

Remarks

The process which issues `single` turns off parallel processing until it either issues a `multiple` command or ends.

Example

```
single
'x <-:x + 1
multiple
```

Where more than one process could be altering `'x`, `single` ensures that no other process reads `'x` and alters it at the same time — otherwise, you would get a wrong result.

Associated Primitive

`multiple`

speed

Remarks

Returns the turtle's current speed.

Examples

```
if speed = 0 [setspeed 10]
```

The following example will change the speed of the turtle smoothly; either accelerating or decelerating it:

```
1: build 'change.speed

change.speed 's
branch :s = speed [stop]
        case :s > speed [setspeed speed +1]
        case :s < speed [setspeed speed -1]
change.speed :s

1: cs
1: tell 1
1: change.speed 40
```

Associated Primitive

```
setspeed
```

sqt *number*

Remarks

Returns the square root of *number*.

Examples

```
1: print sqt 16
4

1: print sqt 169
13

1: print sqt 5
2.23606797749979
```

stamp

Remarks

Prints a copy of the current turtle shape onto the graphics screen at the turtle's current position. The copy will have the current turtle's body colour.

Examples

The following example stamps a ring of turtle shapes around the centre of the drawing position.

```
1: build 'stamp.circle

stamp.circle
cs
lift
fd 90
lt 90
repeat 36 [fd 10 lt 10 stamp]
drop

1: stamp.circle
```

This example draws a row of parked cars.

```
1: cs
1: tell 1
1: setshape 'car
1: setx -100
1: lift
1: repeat 6 [stamp rt 90 fd 30 lt 90
        setc pick 15]
```

stop

Remarks

Logo stops executing the procedure in which `stop` occurs and either continues running the procedure which called it or returns to the prompt `1:`.

Example

The following procedure draws a 'spiral square', stopping when the side of length 90 steps:

```
1: build 'spiral.square

spiral.square 'side
if :side > 90 [stop]
fd :side
lt 90
spiral.square :side +5

1: spiral.square 7
```

Associated Primitives

`end, escape`

subtract *number1 number2*
sub
-

Remarks

Returns the result of subtracting *number1* from *number2*.

The infix symbol - can be used between *number1* and *number2*.

Examples

```
1: print 7 -5
2
1: print subtract 5 7
-2
1: print 5 --5
10
```

Associated Primitives

`add, divide, multiply, power`

tan *angle*

Remarks

Returns the tangent of *angle* when *angle* is specified in degrees.

Example

```
1: print tan 45
1
```

Associated Primitives

`cos, sin`

tell *nwl*

Remarks

Tells Logo which turtles you want it to 'talk' to. If a turtle with the given number or name does not exist then it will be created. Up to eight turtles can exist at any time.

If you create an even number of turtles in the same colour, and the same position you cannot see the turtles. This is because they are drawn on screen in exclusive OR mode.

The special case `tell []` means that the invoking process ceases to address any turtles.

Following a change into graphics mode after text mode, one turtle is present on screen.It is turtle number 1, also known as `seymour`.

Examples

```
1: tell 2

1: tell 'eric

1: tell [1 2 3]
```

Associated Primitives

`told, toldq, turtles, vanish`

text *word*

Remarks

Returns the definition of a procedure as a list.

The text returned on the screen is in the same form as the input to `define`.

Example

```
1: print text  'house
[[house] [square] [lt 60] [triangle]
        [rt 60] [back 100]]
```

Associated Primitives

`define, po`

textscreen
ts

Remarks

Allows the whole screen to be used for text.

Associated Primitive

`clearscreen`

throw *word*

Remarks

Used with the `catch` primitive. See Chapters 5 and 12 for fuller explanations of both `throw` and `catch`.

Associated Primitive

`catch`

titles

Remarks

Returns a list of all the procedures in memory except those which are buried.

Example

```
1: say titles

square triangle hexagon
```

Associated Primitive

po

told

Remarks

Returns the names or numbers of any turtles that the current process is 'talking' to.

If a turtle has a name, `told` will return it, otherwise it returns the number.

Examples

```
1: tell 'eric
1: tell 'sidney
1: print told
[sidney]

; eric still exists but
; this process is not talking
; to him.

1: tell [eric sidney]
1: print told
[eric sidney]
```

Associated Primitives

`tell, turtles, vanish`

touch

Remarks

Returns the colour of the background directly beneath the pen of the first turtle created.

Examples

```
1: cs
1: say touch
0
1: forward 5
1: say touch
15
```

Associated Primitives

`setpoint, point`

towards *list*

Remarks

Returns a list of two numbers.

list represents a point on screen as a coordinate pair.

The first number is the distance in steps towards the point *list*. The second is the clockwise direction in degrees from the turtle's heading to the point *list*.

Examples

```
1: cs
1: print towards [20 0]
[20 90]
```

To move the turtle to the point [20 0] type:

```
1: seth last towards [20 0]
1: forward first towards [20 0]
1: print pos
[20 0]
```

trace *wl*

Remarks

Tells Logo to give a message every time the procedures named by *wl* are called.

The message ends `...?` and waits for you to press:

- <ENTER> to continue
- <ESC> to stop
- <F10> key to stop tracing but resume processing.

See Chapter 12 for a more detailed description.

Examples

```
1: trace "explore
1: trace [explore report]
```

Associated Primitives

`bug, unbug, untrace, walk, unwalk`

turtles

Remarks

Checks which turtles have been created and returns their names and numbers in the form of a list.

Examples

```
1: print turtles
[1 2 3 4]
```

One useful trick to get rid of all the turtles is

```
1: tell turtles vanish
```

Associated Primitives

`tell, told, vanish`

type *nwl*

Remarks

This gives the same output as `say`, except that it is not followed by a carriage return.

Lists are printed without their outermost brackets and with no spaces before punctuation marks.

`type` is useful for sending escape sequences or for multiple output on a single line.

Examples

```
1: make  'x 42
1: make  'y 43
1: type [x = \20] type :x type [\20and y = \20]
        type :y type '\0a type '\0d

x = 42 and y = 43
```

unbload *wl*

Remarks

Removes the Logo extensions that were loaded from the file(s) named in *wl*.

Example

```
1: unbload 'stuff.lgx
```

removes the extensions that were loaded from the file `stuff.lgx`.

Associated Primitives

`bload, bloaded`

unbug *wl*

Remarks

Cancels the effect of **bug**. The named variable need not exist.

Examples

```
1:unbug  'x

1:unbug [x y]
```

Associated Primitives

`bug, trace, untrace, unwalk, walk`

unequalq *nwl1 nwl2*
ueq
~=

Remarks

Returns the value `'true` if *nwl1* and *nwl2* are unequal, and `'false` if they are equal.

- Two numbers are considered equal if they differ by 1/2000000 or less.
- Words are considered unequal unless they contain the same order of letters, irrespective of case.
- Lists are unequal unless their elements are equal and in the same order.

Example

```
1: unless :x ueq 10 [say [something is wrong]]
```

Associated Primitives

`equalq, greaterequalq, lessequalq`

unless *a list*

Remarks

Executes *list* unless the expression *a* is `'true`.

This primitive is equivalent to:

`if not` *a list*

Example

```
unless :number > 0 [error.handler]
```

Associated Primitives

`do, if, while`

unmake *wl*

Remarks

If the input is a word, the variable named by the word is erased.

If the input is a list, each variable named in the list is erased.

`unmake made` will remove all variables.

You will get an error if you try to `unmake` a non-existent variable.

Examples

```
1: unmake "x

1: unmake [x y z]
```

Associated Primitives

`made, make`

untrace *wl*

Remarks

Cancels the effect of `trace` on the procedures named by *wl*. You cannot `untrace` the procedures named by *wl* unless they exist.

Examples

```
1: untrace 'explore
1: untrace [explore report]
```

Associated Primitive

`trace`

unwalk *wl*

Remarks

Cancels the effect of `walk` for the procedures named by *wl*. You cannot `unwalk` procedures that do not exist.

Examples

```
1: unwalk 'explore
1: unwalk [square triangle]
```

Associated Primitive

`walk`

uppercase *nwl*

Remarks

Converts every alphabetic character of *nwl* into a capital letter and returns the changed *nwl*.

Example

```
1: say uppercase [LOGO system]
LOGO SYSTEM
```

Associated Primitive

`lowercase`

upq

Remarks

Returns `'true` if the turtle's pen is up and `'false` if it is down.

Associated Primitives

`drop, lift`

value *word*

Remarks

Returns the value associated with the name *word*. `value` has the same effect as putting a colon (:) before a name: it returns the contents. Unlike the colon, it can be recursive.

Examples

```
1: make "x 12
1: say value "x
12

1: say :x
12

1: build "decrement

decrement "v.name
make :v.name (value :v.name)-1

1: make "x 42
1: print :x
42
1: decrement "x
1: print :x
41
```

Associated Primitive

`valueq`

valueq *word*

Remarks

Returns `'true` if *word* is the name of a variable, otherwise it returns `'false`.

Associated Primitive

`value`

vanish

Remarks

Turtles controlled by the process disappear. They are no longer maintained by Logo.

Associated Primitives

`tell, told, toldq, turtles`

walk *wl*

Remarks

When any of the procedures named by *wl* are called, Logo prints each line before executing it and waits for you to press a key before continuing. This is useful in debugging.

The keys you can press, and their significance, are described in Chapter 12.

The procedures must exist when you give the command `walk`.

Examples

```
1: walk "explore
1: walk [square circle]
```

Associated Primitive

`unwalk`

whenever *a list*

Remarks

`whenever` is intended for parallel processing. The expression *a* is evaluated continuously and, when it becomes `true`, Logo executes the command *list*.

Logo waits for *a* to become `true` again before running *list* again.

Example

```
whenever :x = 0 [make 'x 100]
```

Associated Primitive

`await`

while *a list*

Remarks

As long as the expression *a* is `'true`, the command *list* is repeatedly executed.

`while` is similar to `do...until` except that *while* may never execute the command *list*, but `do...until` always executes it at least once.

Example

```
1: make 'number 12
1: while :number > 0 [print :number
        make :number :number-1]
```

Associated Primitives

`do, if, unless`

wordq *nwl*

Remarks

Returns `'true` if *nwl* is a word and `'false` otherwise.

Examples

The following procedure tests if its input is a word:

```
1: build 'checkword

checkword 'object
if wordq :object [say 'word] [say [not word]]

1: make 'item1 'patsy
1: checkword :item1
word

1: make 'item2 [1 2 3 4]
1: checkword : item2
not word

1: checkword 3
not word
```

wrap

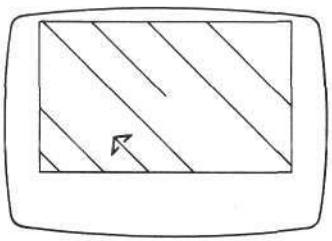
Remarks

Makes the turtle wrap around the screen.

If it is sent off the right hand edge of its field, it reappears on the left; if it is sent off the top, it reappears at the bottom. The turtle's drawing heading always remains unchanged.

Examples

```
1: cs
1: wrap
1: left 45
1: forward 1000
```



Associated Primitives

`fence, fenceq, nowrap, wrapq`

wrapq

Remarks

Returns `'true` if the `wrap` primitive has been used, otherwise it returns `'false`.

Associated Primitives

`fence, fenceq, nowrap`

writefilec *nw filename*
wfc

Remarks

Writes the first character of *nw* to the named file, which must have been opened for output.

The character is joined with the previous character output.

wfc returns `'true` if the character is successfully written and `'false` otherwise.

Example

```
1: unless writefilec 'b 'myfile.dat
        [say [not written]]
```

Associated Primitives

`closefile, outfile, outfiles, writefiled, writefilel`

writefiled *nwl filename*
wfd

Remarks

Writes *nwl* to the named file, which must already be open for output. The data is written in the format used by `print`.

`'true` is returned if the data is successfully written and `'false` otherwise.

Example

```
1: unless writefiled 'rhubarb 'myfile.dat
       [say [not written]]
```

Associated Primitives

`closefile, outfile, outfiles, writefilec, writefilel`

writefilel *list filename*
wfl

Remarks

Writes *list* to the named file, which must already be open for output. The data is written in the format used by `say` and is followed by a carriage return.

The value `'true` is returned if *list* is successfully written, and `'false` otherwise.

Example

```
1: unless writefilel [rhubarb] 'myfile.dat
        [say [not written]]
```

Associated Primitives

`closefile, outfile, outfiles, writefilec, writefiled`

xcor

Remarks

Returns the turtle's current x coordinate.

Examples

```
1: cs
1: print xcor
0

1: setpos [100 50]
1: print xcor
100
```

Associated Primitives

`pos, setpos, setx, sety, ycor`

xor *a b*

||

Remarks

Stands for exclusive or. Returns `'true` if either *a* or *b* is true and the other false; otherwise it returns `'false`. The table below shows `xor` working with *a* and *b* values:

a	*b*	`xor` *a b*
`'false`	`'false`	`'false`
`'false`	`'true`	`'true`
`'true`	`'false`	`'true`
`'true`	`'true`	`'false`

Example

This procedure acts like a simple two-way light switch.

```
1: build 'lights

lights 's1 's2
result if xor (:s1 = 'up) not (:s2 = 'up)
       [[lights on]] [[lights off]]
```

If both switches are up or down then the lights are off. If they are in different positions the lights are on.

```
1: say lights 'up 'up
lights off
1: say lights 'up 'down
lights on
```

Associated Primitives

`both, either, not`

ycor

Remarks

Returns the turtle's current y coordinate.

Examples

```
1: cs
1: print ycor
0

1: setpos [100 50]
1: print ycor
150
```

Associated Primitives

`pos, setpos, setx, sety, xcor`

# #

list # number

Remarks

Returns the *number* th element of *list*.

can also be used after `make` to change the value of an element of a list.

Examples

```
1: make 'notice [NO SMOKING]
1: say :notice # 2
SMOKING
1: say :notice # 1
NO

1: make 'notice # 2 'eating
1: say :notice
NO eating
```

The following example uses # recursively to handle simple structured data.

```
1: make 'marks [[Maths 8] [Chemistry 5]
       [English 7]]
1: print :marks #3 #2
7
```

Logo Keywords

Logo recognises special words which are not primitives.

These *keywords* are:

`case`
`default`
These are part of the `branch` primitive.

`until`
This is part of the `do...until` primitive.

`'true`
`'false`
These are the boolean data types recognised by Logo.

Logo Signals

The Logo system throws signals which you need to catch within your program otherwise your program may come to a stop.

The system signals are:

```
cancel, endfile, error, fence, touch,
touchturtle, escape
```

and can be caught in your program using `catch`. For example:

```
catch 'endfile [say rfc 'myfile]
```

More information on the system signals can be found in chapters 5 and 12.

Special Logo Characters

Logo recognises various characters as having a special effect. Their uses are detailed in the associated primitive descriptions. The characters are:

`:` *colon*
Precedes a variable name and returns its contents.

`;` *semi-colon*
Precedes a comment which is not part of the program.

`:-` *tag*
Receives control from a `goto` command. See `goto`.

`[ ]` *square brackets*
The contents inside the brackets are a list.

`'` *quote mark*
Indicates that what follows is either a name or a word.

`( )` *round brackets*
Ensures a specific order of evaluation.

`\` *backslash*
The subsequent character is to be treated as an ordinary text character or a hexadecimal value.

`*` *asterisk*
For multiplication. See `multiply`.

`+` *plus*
For addition. See `add` and `join`.

`-` *minus*
For subtraction. See `subtract`.

/ *slash*
For division. See `divide` and `share`.

| *vertical bar*
`either` and `xor`.

↑ *up arrow*
For exponentiation. See `power`.

~ *tilde*
Symbol for 'not'. See `not` and `unequalq`.

$ *dollar*
Specifies Logo's definition of the following name or word is to be used. See chapter 15.

hash
Counts the elements in a list. See the # primitive which is the last primitive described.

= *equals*
Tests for equality.

< *less than*
Tests for one value being less than another.

> *greater than*
Tests for one value being greater than another.

<- *greater than dash*
Gives contents to a variable. See the `make`.

% *percentage*
Returns the remainder of a division. See `remainder`

& *ampersand*
Joins expressions, words and lists. See `both` and `sentence`.

Index

RESEARCH MACHINES plc
Mill Street, Oxford OX2 0BW Telephone (0865) 791234
Facsimile (0865) 796279 Telex 837203