Logo

on the BBC Microcomputer and Acorn Electron

BARRY MORRELL

ACORNSOFT

Acknowledgements

The Acornsoft Logo program was written in BCPL by Chris Jobson and John Richards with assistance and advice from many other people. We would like to thank Richard Noss of the Advisory Unit for Computer Based Education, in particular, for his advice.

ISBN 0 907876 96 X

First published in 1984 by Acornsoft Limited

Note: Within this publication the term 'BBC' is used as an abbreviation for 'British Broadcasting Corporation'.

FIRST EDITION

Acornsoft Limited, Betjeman House, 104 Hills Road,
Cambridge CB2 1LQ, England. Telephone (0223) 316039

Contents

How to use this book

If you are new to programming and to Logo you should start by reading the companion book, *Introduction to Logo on the BBC Microcomputer and Acorn Electron*. This reference manual is intended for those who are familiar with programming but not with Logo.

A general description of Logo appears in chapter 1 and, in addition, most chapters begin with some background information and a summary of the primitives. If you want to give yourself a quick briefing on Logo you may find it helpful to glance through these areas first.

To aid easy reference, this book has Logo primitives arranged by chapters into distinct groupings. This means that if you want a primitive to perform a specific task you can find the appropriate group by looking in the Contents list. For details of a particular primitive you should look it up in the Index.

Once you are more familiar with Logo you will probably find that you use the accompanying *Logo Reference Card* most of the time and refer to this book only occasionally.

1 Introduction to Logo for experienced programmers

1.1 Acornsoft Logo

Acornsoft Logo is a full, new and accurate version of this attractive educational language and it conforms closely to the implementations developed at the Massachusetts Institute of Technology. It can thus accept most published programs in Logo and can be used with the most popular books on the language.

Acornsoft Logo is also faithful to the BBC Microcomputer and the Acorn Electron. It allows full use of their powerful graphics features and has integrated these into Logo as new pen characteristics. It also supports the 6502 Second Processor and the sound, analogue, joystick and VDU functions.

Since the world of Logo is constantly developing with new hardware devices and new ideas for microworlds, this implementation has been designed to accept 'extensions' to the language, and an initial range of such extensions is provided.

The turtle graphics facilities allow the use of words or shapes as turtles and support up to 32 screen turtles, as well as a range of floor turtles. A flexible set of trace and debugging functions is also provided.

Logo's traditional list processing commands can also be applied to words in all cases. Additional list processing commands are provided to access and change individual items within a list.

Many commands which take a single word as an input can also take a list and apply to each word in the list. This provides a simple introduction to list processing.

You can easily tailor Acornsoft Logo to your own, individual requirements. For example, the floating point number system can be restricted to integers and primitives can be redefined.

Finally, a completely new set of Logo example programs illustrates the use of the language in a wide variety of different activities such as conversation, data systems, mapping, maze following, logical language and natural language. We hope and expect that once you have enjoyed the attractions of turtle graphics these examples will encourage further exploration of the extensive possibilities of Logo.

1.2 Starting up Logo

The accompanying leaflet will have shown you how to install your Logo ROMs (in the BBC Microcomputer) or ROM cartridge (in the Acorn Electron Plus 1). To start up Logo you merely need to switch on your machine and the following message will be displayed:

```
Welcome to Logo
```

If you do not get this message, you can still get into Logo by typing `*LOGO`.

When you first enter Logo you are in 'graphics mode' and you can use any of the turtle graphics primitives described in this book. You can get into 'text mode' by typing `TS` then pressing RETURN. To get back to graphics mode again, type `DRAW` then press RETURN, or use any of the graphics primitives such as `CLEAN`.

If you are using turtle graphics, Logo allocates six lines at the bottom of the screen for text; the rest is devoted to graphics. You can vary the number of text lines between one and 20:

```
(DRAW 20)
(DRAW 8)
```

Some of the more advanced and little used parts of Logo are held on the tape or disc which accompanies your Logo package; they are known as Logo 'extensions'. An accompanying booklet describes the contents of these extensions and how to load them.

1.3 Use of typefaces in this book

Dialogue between yourself and the computer is printed in this book in a different typeface from the normal one; it resembles more closely the sort of typeface you will see upon your screen, for example:

```
PRINT [HELLO WORLD]
HELLO WORLD
```

1.4 Typing direct commands at the keyboard

1.4.1 Primitives

When Logo is expecting you to type a 'command line' at the keyboard, it displays the ? prompt. It is then said to be in 'command mode'.

The commands you can type include the following:

`DRAW` This initiates turtle graphics and puts the turtle at the centre of the screen (the 'home position') pointing upwards.

`CLEAN`	This clears whatever graphics are on the screen but leaves the turtle wherever it was.
`HOME`	This returns the turtle to its home position and leaves it pointing upwards.
`FORWARD`	This moves the turtle forward by a number of steps.
`LEFT`	This turns the turtle left (anticlockwise) by a specified angle (in degrees).
`RIGHT`	This turns the turtle right (clockwise) by a specified angle (in degrees).

These commands are examples of 'primitives'. Primitives are words which are built into Logo; when you switch on your computer they are already there.

Some of these primitives do not need any other information (for example, `CLEAN` and `HOME`); you can type them in and they will perform a unique action. Others need 'inputs' which you can vary, for example:

```
FORWARD 100
FORWARD 300
RIGHT 60
LEFT 90
```

1.4.2 Error handling

When you type in a primitive you get Logo to take action upon it by pressing RETURN. If you make a mistake before you press this you can correct it using DELETE. If you don't do this, Logo will reply with an error message telling you what is wrong. For example, if you type `FORWRAD 100` you will get the message:

```
Logo doesn't know how to FORWRAD
```

You can then type the correct information.

1.4.3 Using primitives

When you type in a line at the keyboard, Logo searches it from left to right for the name of a primitive or a procedure (procedures are described later in this chapter). Then, if Logo expects inputs, it looks for these and evaluates them. Finally, it does something with the result (such as moving the turtle).

Logo allows you to type a number of primitives separated by spaces on the same line, although they will not be executed until you press RETURN. If you type more than will fit onto one line, the rest of your command line will run onto the next line but will still be valid. In this book, long command lines are shown with

the continuation lines printed white on black where they would appear in inverse video in the Logo editor (which is described in chapter 4).

Logo also allows you to type in short forms of some primitives, for example, `FORWARD` can be replaced by `FD`, `LEFT` by `LT` and `RIGHT` by `RT`. The short forms are given with the description of each primitive in the following chapters, and they follow its name.

1.5 Procedures

You can teach Logo new words in terms of those words it already knows, and these are called 'procedures'. You define a procedure using the `TO` primitive. The following commands form two procedures which draw a square and a triangle respectively:

```
TO SQUARE
REPEAT 4 [FORWARD 200 LEFT 90]
END

TO TRIANGLE
REPEAT 3 [FORWARD 200 LEFT 120]
END
```

The first line of these procedures is called the 'title line'. It tells Logo that you want to define a procedure named `SQUARE` or `TRIANGLE`. You can then type in commands which are stored in memory for later execution. When you are defining a procedure, Logo displays the prompt >. When you are back in command mode, the ? prompt appears.

You can abandon the procedure definition at any time by pressing ESCAPE.

The `END` primitive tells Logo that you have finished defining your procedure and it returns control to command mode. You can then run your procedures by typing:

```
SQUARE
DRAW
TRIANGLE
```

If you want to modify your procedures you cannot do so using the `TO` primitive. Instead, you must use the Logo editor described in chapter 4, 'Using the editor'. You can also define a new procedure using the editor, if you wish, and this can sometimes be more convenient than using `TO`.

You can call other procedures from within one procedure. For example:

```
TO HOUSE
SQUARE
FORWARD 200
LEFT 30
TRIANGLE
PENUP
HOME
HIDETURTLE
END
```

HOUSE

1.5.1 Inputs to procedures

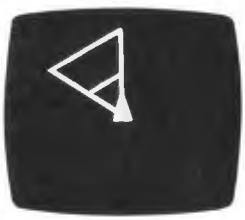
Your procedures can have inputs, just like some of the primitives mentioned above. For example, the following changes would make `SQUARE` and `TRIANGLE` draw shapes of varying size:

```
TO SQUARE :SIDE
REPEAT 4 [FORWARD :SIDE LEFT 90]
END
```

```
TO TRIANGLE :SIDE
REPEAT 3 [FORWARD :SIDE LEFT 120]
END
```

SQUARE 100

SQUARE 300



TRIANGLE 100
TRIANGLE 300

In the title line for `SQUARE`, you are creating a 'box' and giving it the name `SIDE`. Within the procedure you want to perform an action using the *contents* of the box called `SIDE`, and the colon (`:`), known in Logo as 'dots', indicates that you are referring to these contents.

Now, when you type in a command line like `SQUARE 200`, the value 200 is put into this box and the primitive `FORWARD :SIDE` uses the contents of the box to produce:

```
FORWARD 200
```

Your procedures can have more than one input. When they do, the inputs must be separated by spaces, for example:

```
TO RECTANGLE :SIDE1 :SIDE2
REPEAT 2 [FORWARD :SIDE1 LEFT 90 FORWARD :SIDE2
LEFT 90]
END
```

1.5.2 Outputs from procedures

As well as having inputs, your procedures can output values using the OUTPUT primitive. Look at the following procedure, for example:

```
TO NUMBER.SQUARE :NUMBER
OUTPUT :NUMBER * :NUMBER
END
```

Note the use of the full stop in the title line to make the procedure name NUMBER.SQUARE more legible. You can't use a space here as the name would become NUMBER.

In this example, the procedure NUMBER.SQUARE outputs the square of its input and this can be displayed using the PRINT primitive:

```
PRINT NUMBER.SQUARE 10
100
```

1.6 The use of CTRL and ESCAPE

You can use the ESCAPE key to interrupt the execution of a command line or a procedure. If you do this whilst the command line:

```
REPEAT 1000 [FD 200 LT 90]
```

is executing, the turtle will stop moving and the line that was being executed will be printed in the text area, together with the [] ? prompt:

```
Interrupted
REPEAT 1000 [FD 200 LT 90]
[] ?
```

You can continue execution by typing CO or CONTINUE.

If you press ESCAPE whilst a procedure is executing, the turtle will stop and the line that was being executed will be printed, together with the procedure name and the ? prompt. For example:

```
Interrupted, in SQUARE
REPEAT 1000 [FD 200 LT 90]
SQUARE ?
```

Again, you can continue execution by typing `CONTINUE`, or just `CO`.

If you want to 'break out' of the procedure or command line completely, you can do so by holding down the CTRL key then pressing ESCAPE. The line that was being executed will be printed, together with the procedure name. For example:

```
Stopped, in SQUARE
REPEAT 1000 [FD 200 LT 90]
?
```

In this case you cannot continue execution.

1.7 Greedy primitives

Some primitives have optional inputs; these are called 'greedy primitives'. For example:

```
DRAW
(DRAW 10)
```

When you want to use the inputs that are optional, you must surround the entire primitive with round brackets (as above); otherwise, you should omit the optional inputs and the brackets.

In the following chapters, greedy primitives can be identified by the first lines of their definition, for example:

```
SUM <number1> <number2>
(SUM <number1> <number2> ... <numbern>)
```

The first of these lines contains the default number of inputs.

Check this against the definition of `SUM` in chapter 8, 'Arithmetic'.

Greedy primitives can appear at the end of a line with less than their default number of inputs; in this case, they do not need brackets. For example:

```
PRINT SUM 2 3
5
PRINT (SUM 2 3 4 5)
14
PRINT (SUM 1)
1
PRINT SUM 1
1
```

1.8 Logo objects

In Logo, there are two types of 'object': words and lists.

Logo words are similar to words in the English language: they consist of groups of characters. You indicate that something in Logo is a word by preceding it with quotes, as in the following:

```
PRINT "HELLO
HELLO
```

You can break words into smaller words or combine them to form long ones. A word with no elements is indicated by " and is called an 'empty word'.

Numbers are a type of Logo word and you can perform arithmetic on them. They are slightly different from normal words in that you do not need to precede them by quotes, for example:

```
PRINT 25
25
```

A list is made up of Logo objects, and these can be words or other lists. You indicate that something is a list by surrounding it with square brackets, for example (as part of a line):

```
[JAMAICA HAWAII 5 [CATS DOGS]]
```

A list with no elements is called an 'empty list' and is indicated by `[]`.

You can manipulate lists in a similar way to words: breaking them into smaller lists or combining them to form longer ones. You can also use them with primitives such as `PRINT`:

```
PRINT [JAMAICA HAWAII 5 [CATS DOGS]]
JAMAICA HAWAII 5 [CATS DOGS]
```

Words and lists are described in chapter 9, 'Words and lists'.

1.9 Naming things

Names can consist of letters, numbers and punctuation. Logo does not care if a name is in lower or upper case. For example, the following are regarded as being the same:

```
FORWARD 100
forward 100
```

When you define procedures you give names to a number of things: the procedure itself and its inputs (if it has any).

You can also give names to data, or variables, using the MAKE command. For example:

```
MAKE "NUMBER 10
PRINT :NUMBER
10
```

The first input to the MAKE command is the name of a 'box' and the second is the thing you are going to put into it: its contents. In the example above, you are giving the value 10 to the name NUMBER. In the PRINT command you are looking at the contents of the box (note the dots in front of NUMBER).

Another way of getting the contents of a box is to use the primitive THING. This is equivalent to dots (:) in the following, simple example:

```
PRINT THING "NUMBER
10
```

However, it is more flexible than dots because it can be part of a more general expression, for example:

```
MAKE "PLACE [HAWAII HONOLULU]
PRINT THING FIRST [PLACE STREET]
HAWAII HONOLULU
```

Here, PLACE is defined as a list using MAKE and the next line prints the contents of the first element of the list [PLACE STREET]. FIRST is a primitive which outputs the first element of the list (in this case, "PLACE). You could not use dots with this example, because dots can only be used before a name.

The names that you use as inputs to procedures are 'private' or 'local' to the procedures themselves. As a result, you don't need to worry about Logo getting mixed up between inputs to different procedures.

1.10 Arithmetic

In Logo, numbers are words made up of digits. They can contain a sign, a decimal point and an E or N. They are described in more detail in chapter 8, 'Arithmetic'.

Logo allows you to perform arithmetic operations using the normal operators: + (plus), – (minus), * (multiplication) and / (division).

When you use – (minus) as an operation, it must be followed by a space.

```
PRINT 3 - 1
2
PRINT 3 -1
3
Logo doesn't know what to do with -1
```

There are also a number of arithmetic primitives such as SQRT, SUM and COS and they are all described in chapter 8, 'Arithmetic'.

1.11 Order of evaluation

The order in which the various operators and primitives are evaluated when they occur together is as follows:

1. Multiplication and division.

2. Addition and subtraction.

3. Most primitives; where a number of these occur together they are evaluated from left to right.

4. The operators >, < and =.

5. The primitives ALLOF, ANYOF, IF, LOCAL, MAKE, NOT, OUTPUT, PRINT, SHOW, TEST, TITLE and TYPE. Where a number of these occur together they are evaluated from left to right.

If you want to have Logo perform a particular calculation before any other, you can make it do so by surrounding the calculation with brackets. For example:

```
PRINT 20 * (20 - 15)
100
```

Here, the subtraction will be done before the multiplication.

1.12 Special characters

1.12.1 Quotes, or "

When used before a word, these indicate that whatever follows is to be used as a word, not the name of a procedure or the contents of a variable ('box').

1.12.2 Dots, or :

When used before a word, these indicate that you are referring to the contents of the variable ('box') named.

Dots at the end of a word indicate that the word is a label which is used with the GO primitive, for example:

```
HERE:
```

Labels can only be used at the start of a line or list.

1.12.3 Square brackets, or []

These surround a list.

1.12.4 Round brackets, parentheses, or ()

These are used to group items into the order in which you want Logo to interpret them, or to identify greedy primitives (see section 1.7).

1.12.5 Up arrow, or ^

This tells Logo to interpret the next character literally, rather than as a character which has a special meaning in Logo. It is used before the following characters:

space ^ ( ) [ ] * / - \ < > =

For example:

```
PRINT "3 + 5
8
PRINT "3^+ 5
3+5
```

1.12.6 Backslash, or \

This tells Logo that the text after it is to be treated as a 'comment'. In other words it is to be used to clarify the logic of your procedures rather than to be acted upon by Logo. For example:

```
TO CIRCLE \ This procedure draws a circle
REPEAT 360 [FD 3 LT 1]
END
```

1.12.7 Star, or *

If this is the first character in a line or list, it tells Logo that the line is an operating system command. For example, *TAPE and *LOGO.

You can use an operating system command inside a procedure, but it must be the only thing on a line or in a list.

1.13 Creating a Logo environment

You may want to restrict the facilities which Logo offers or extend them in some way. For example, you might want to:

1. Restrict the precision of numbers.

2. Redefine primitives such that `FORWARD 10` moves the turtle by 100 steps instead of 10.

3. Change the initial screen mode and start up colours.

4. Have certain of your procedures treated as primitives in that they cannot be edited by users.

5. Rename primitives for use with other languages.

You can do any of these things by creating a 'Logo environment'. The actions needed are described in chapter 16, 'Creating a Logo environment'.

1.14 Extensions to Logo

One of the main benefits of Acornsoft Logo is the fact that it supports 'extensions' which can be loaded from disc and tape. This means that its usefulness is not limited by future improvements to either Logo or your computer.

Where a primitive is included in an extension, we will mention this fact under the description of the primitive. Descriptions of the extensions themselves are given in an accompanying booklet. Future extensions will be described in the documentation supporting them.

Extensions are loaded using the primitive `LOAD`. For example, the following command loads the multiple turtles extension `MULT`:

```
LOAD "MULT
```

1.15 Syntax of primitives

In the descriptions of primitives throughout the following chapters, wherever inputs are required these are given as text in angled brackets, for example:

```
OUTPUT <object>
```

This means that the primitive `OUTPUT` needs one input which is a Logo object.

The words which we use in describing the inputs to Logo primitives are explained here:

`<a,b>` An expression which is either `"TRUE` or `"FALSE`.

<byte> A unit of data used by the computer. Integer between 0 and 255.

<character> Letters of the alphabet, numbers, etc. See Appendix C, 'ASCII code table'.

<degrees> An angle in degrees. (See <number>.)

<distance> (See <number>.)

<filename> Any valid filename for the current filing system.

<item> An object which is part of another object such as a list in another list.

<list> Either the empty list [] or one or more items enclosed by square brackets.

<n> A number in the range –32768 to 32767.

<name> A word of between 1 and 63 characters, which may be used for a Logo procedure or variables. It must not start with a numeric character or Logo punctuation, ie ( ) [ ] * / \ < > = space ^ .

<number> On input, <number> is a word without spaces containing an optional sign, a decimal number with an optional decimal point, and an optional exponent. Positive exponents are introduced by E and negative exponents by N. On output, numbers are shown in decimal format if zero or in the range 0.01 to 99999999.

<object> Any Logo word or list (characters and numbers are particular kinds of words).

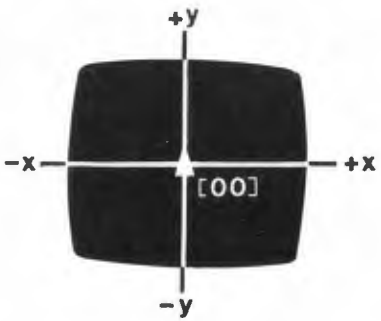
<property name> A name used as a property description.
or <pr>

<word> Contains any set of characters preceded by a ". Any Logo punctuation in a word must be preceded by ".

2 Turtle graphics

When you type DRAW, Logo displays the turtle graphics screen. You can then type in the turtle primitives described in this chapter or run procedures which use them.

Most of the primitives which move the turtle produce movements relative to the turtle's current position. However, some primitives such as SETX and SETY produce absolute movements using the following system of coordinates:



The use of colour is not described in this chapter. You can produce some extremely attractive effects when you combine turtle graphics with colour and the range of possibilities is large. For this reason, colour is treated as a separate topic and has a chapter to itself: chapter 15, 'Screen modes and the use of colour'.

Primitives which redefine the turtle's shape and handle multiple turtles are covered in chapter 12, 'Turtle shapes and multiple turtles'. However, some of the turtle graphics primitives which can be used with one turtle have a special effect when used with multiple turtles. Where this is the case, it will be mentioned in the present chapter under the descriptions of the primitives concerned.

2.1 Summary of primitives

Primitive	Effect
BACK	Moves turtle backwards
BG	Returns background colour
CLEAN	Erases graphics area without moving turtle
CS	Erases graphics area and homes turtle
CT	Clears text area
DISTANCE	Returns distance from turtle to a specific point
DOT	Returns colour of a dot
DRAW	Erases graphics area, homes turtle and resets some screen functions
FENCE	Sets fence round graphics area
FORWARD	Moves turtle forwards
HEADING	Returns turtle's heading
HIDETURTLE	Hides turtle from view
HOME	Moves turtle to centre of screen
LEFT	Turns turtle to left
MODE	Returns the screen mode
PAL	Sets logical colour
PC	Returns the pen colour
PE	Puts turtle's eraser down
PEN	Returns current pen parameters
PENDOWN	Puts turtle's pen down
PENRESET	Resets turtle's pen to initial state
PENUP	Lifts turtle's pen
PENUPQ	Reports whether pen is up
POS	Returns turtle's position as `[x y]`
PX	Makes turtle draw new lines but erases existing ones
RIGHT	Turns turtle to right
SCR	Returns aspect ratio of screen
SECT	Draws sectors of a circle
SETBG	Changes background colour
SETDOT	Puts a dot at a specific position
SETHEADING	Turns turtle to point to a specific heading
SETMODE	Changes the screen mode
SETNIB	Selects graphics option of BASIC `PLOT` statement
SETPC	Changes the pen colour
SETPEN	Set pen parameters

Primitive	Effect
SETPOS	Moves turtle to [x y]
SETPT	Defines the use of colour on the screen
SETSCR	Changes aspect ratio of screen
SETSH	Changes turtle's shape
SETX	Moves turtle horizontally
SETY	Moves turtle vertically
SH	Returns turtle's shape
SHOWTURTLE	Makes turtle visible
STAMP	Stamps the turtle's shape on the screen
TITLE	Prints text in graphics area
TOWARDS	Returns heading that would point the turtle at position [x y]
WINDOW	Removes restrictions imposed by FENCE and WRAP
WRAP	Arranges for turtle to 'wrap' from one side of screen to the other when it hits fence
XPOS	Returns turtle's x-coordinate
YPOS	Returns turtle's y-coordinate

2.2 Primitives

BACK(BK)

```
BACK <distance>
```

Moves the turtle backwards by <distance> steps. The turtle's heading does not change.

BACK 100

BG

Returns an integer which represents the logical background colour (see chapter 15, 'Screen modes and the use of colour').

CLEAN

Clears (erases) whatever graphics are on the screen, but does not move the turtle or change its state.

```
CLEAN
```

CS

Clears (erases) whatever graphics are on the screen and returns the turtle to its home position with a heading of zero.

```
CS
```

CT

Clears the text area and puts the cursor on the first line of the text area.

DISTANCE

```
DISTANCE <list>
```

Returns the distance from the current turtle position to a point on the screen addressed by `<list>`.

Example

```
HOME
PRINT DISTANCE [100 100]
141.42136
```

DOT

```
DOT <list>
```

Returns the colour of the dot at a position in the form `[x y]` specified by `<list>`. The numbers returned correspond to the colours shown in table 15.2.

If the position is off the screen, the value 255 is returned.

DRAW

```
DRAW
(DRAW <n>)
```

This primitive does the following things:

1. Sets the background colour to 0 (normally black).

2. Sets the screen to wrap mode and clears the graphics screen and text area.

3. Destroys all turtles except turtle 0 if multiple turtles are in use (see chapter 12, 'Turtle shapes and multiple turtles').

4. Returns turtle 0 to the 'home position' (the centre of the graphics screen) and makes it visible. Resets the turtle's shape to a triangle.

5. Sets the pen colour to 7 (normally white) and puts the pen down. Selects the default `SETNIB` and `PENSTATE` options.

If `<n>` is specified, this number of lines will be reserved at the bottom of the screen for text (up to 20 text lines are allowed). You can reset the text area to its default size (6 lines) by omitting `<n>`:

DRAW

FENCE

Sets a fence around the graphics area. An error will occur if the turtle hits this fence.

The fence is created immediately in logical colour 7, which is normally white (logical colours are described in chapter 15, 'Screen modes and the use of colour').

See also `WINDOW` and `WRAP`.

Example

```
FENCE
FORWARD 2000
```

gives the error message `Turtle hit fence.`

FORWARD(FD)

```
FORWARD <distance>
```

Moves the turtle forward by `<distance>` steps. Its heading (see `HEADING`, below) does not change.

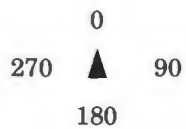
Example

`FORWARD 100`

HEADING

```
HEADING
(HEADING <n>)
```

Returns the turtle's heading, or the heading of turtle `<n>`. The heading is the direction in which the turtle is pointing, in degrees, using the following system:



Example

```
TO TURN
RIGHT 15
IF HEADING = 90 [PRINT [YOU ARE HEADED EAST]]
END

REPEAT 6 [TURN]
YOU ARE HEADED EAST
```

HIDETURTLE(HT)

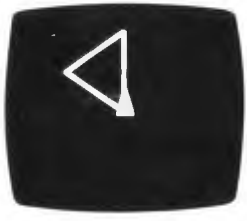
Hides the turtle from view until the next occurrence of the SHOWTURTLE primitive.

Example

HIDETURTLE

HOME

Moves the turtle to the centre of the screen (the 'home position') and leaves it pointing upwards. The screen is not cleared and if the pen is down, the track to the centre is drawn.



HOME

LEFT(LT)

```
LEFT <degrees>
```

This primitive turns the turtle left (anticlockwise) through an angle specified by `<degrees>`.

If `<degrees>` is negative, the turtle will turn in a clockwise direction.

Examples

```
LEFT -45
```

MODE

Returns the screen mode. Screen modes are described in chapter 15, 'Screen modes and the use of colour'.

PAL

```
PAL <logicalcolour> <physicalcolour>
```

Stands for PALette. Sets one of the logical colours of the BBC Microcomputer or Acorn Electron to a specific physical colour. See chapter 15, 'Screen modes and the use of colour', for a full description.

PC

```
(PC <n>)
```

Returns the current pen colour. If you are using multiple turtles you can find the pen colour for turtle `<n>` using the 'greedy' form of the primitive shown above. Otherwise, it needs no inputs.

Examples

The following example shows how you can use `PC` with one turtle:

```
PRINT PC
7
```

The next example shows how it can be used with a number of turtles:

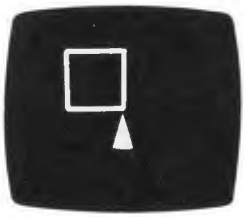
```
PRINT (PC 2)
7
```

This shows the colour of turtle number 2.

PE

Puts the turtle's 'eraser' in the down position. When the turtle moves it will then erase lines over which it passes. To lift the eraser you must use PENDOWN, PENUP, PX or PENRESET.

Example



```
PE
FORWARD 200
```

PEN

```
PEN
(PEN <n>)
```

This operation returns the current pen parameters in the form of a list. If you are using multiple turtles you can find the pen parameters for turtle <n> using the 'greedy' form of the primitive shown above. Otherwise, it needs no inputs.

The elements of the list returned are as follows:

```
[PENSTATE SHOWN COLOUR NIB PENTYPE]
```

- PENTYPE — Defines use of colour (see the SETPT primitive in this chapter).
- NIB — Graphics option (see the SETNIB primitive in this chapter).
- COLOUR — Pen colour as defined in table 15.2 in section 15.2.
- SHOWN — Whether or not turtle is visible (TRUE or FALSE).
- PENSTATE — One of the values PU, PD, PE or PX.

Example

```
PRINT PEN
PD TRUE 7 8 0
```

PENDOWN(PD)

Puts the turtle's 'pen' down. When the turtle is moved it will then draw lines in the current pen colour. `DRAW` resets the pen to the down position.

Example

```
PENUP
```

```
FORWARD 150
```

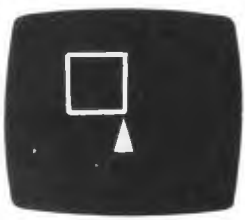
```
PENDOWN
FORWARD 150
```

PENRESET

Resets the pen to the state it was in when it was first used. The colour will be set to logical colour 7 (normally white), the nib to 8 and the pen type to 0. The pen will be put down and the turtle will be shown.

PENUP(PU)

Lifts the turtle's pen. When the turtle subsequently moves it will not draw lines.



```
PENUP
FORWARD 200
```

PENUPQ

Returns `TRUE` if the pen is up and `FALSE` if it is down.

POS

(POS <n>)

Returns the turtle's position (in the form of x-, y-coordinates) as a list [x y]. After you type DRAW, the turtle will be at [0 0], the home position.

If you are using multiple turtles you can find the position of turtle <n> using the 'greedy' form of the primitive shown above. Otherwise, it needs no inputs.

Example

This example assumes the turtle is away from the home position. The primitives below draw a circle at the home position then move the turtle back to wherever it was on the screen:

```
TO CIRCLE
REPEAT 360 [FORWARD 3 LEFT 1]
END

TO HOMECIRCLE
MAKE "SAVEPOSITION POS
PENUP
HOME
PENDOWN
CIRCLE
PENUP
SETPOS :SAVEPOSITION
END
```

HOMECIRCLE

The following example shows how POS can be used with multiple turtles:

```
PRINT (POS 2)
100 100
```

PX

Sets a reversing pen. When you use this primitive and then move the turtle, the pen will draw new lines, but erase existing ones.

Example

The following procedure draws spinning squares without any 'spokes' and then removes them:

```
TO SPIN.WIPE
PX
SETMODE 5
SETPC 0
SETBG 2
REPEAT 2 [SPIN WAIT 2]
END

TO SPIN
REPEAT 24 [LT 15 SQUARE]
END

TO SQUARE
REPEAT 4 [FD 200 LT 90]
END
```

RIGHT(RT)

```
RIGHT <degrees>
```

Turns the turtle right (clockwise) through an angle specified by `<degrees>`. If `<degrees>` is negative, the turtle will turn in an anticlockwise direction.

Examples

RIGHT 45 RIGHT −45

The following procedure will draw triangles of varying sizes:

```
TO VARIABLE.TRIANGLE :SIDE
REPEAT 3 [FORWARD :SIDE RIGHT 120]
END
```

SCR

Returns the aspect ratio of the screen (see SETSCR).

Example

```
PRINT SCR
1
```

SECT

```
SECT <radius> <angle> <width>
```

Draws a sector through the specified <angle>. <radius> is the distance from the turtle to the centre of curvature (a positive radius means that the centre is to the right of the turtle). <width> specifies the separation of the two lines of the arc (a positive width means that the second line is to the right of the turtle). If <angle> is positive, the turtle moves forward; if negative, it moves backwards.

The turtle finishes at the other end of the line from its starting point.

If the nib has been set to 80, the space between the lines is filled.

Examples of the use of SECT are given in the booklet which describes the extensions.

SETBG

```
SETBG <n>
```

Changes the logical background colour (initially black, or 0) to colour <n>. See chapter 15, 'Screen modes and the use of colour', for a full description. Logo also performs an immediate CLEAN using the new colour.

SETDOT

```
SETDOT <list>
```

Puts a dot at the position given by <list>, using the turtle's current pen colour. The turtle is not moved during this process. <list> is in the form of x-, y-coordinates.

If the position is off the screen, an error is generated in WRAP or FENCE modes and the command is ignored in WINDOW mode.

Example

```
TO CIRCLE
MAKE "ANGLE 0
REPEAT 360 [PLOT MAKE "ANGLE :ANGLE + 1]
```

```
END

TO PLOT
SETDOT LIST (100 * SIN :ANGLE) (100 * COS :ANGLE)
END
```

HOME

CIRCLE

SETHEADING(SETH)

`SETHEADING <degrees>`

Turns the turtle so that it is pointing in the direction (ie has the heading) given by `<degrees>`.

Examples

HOME

SETHEADING −45

SETHEADING 45

SETMODE

`SETMODE <n>`

Selects screen mode `<n>`. For a full description of `SETMODE`, see chapter 15, 'Screen modes and the use of colour'.

SETNIB

`SETNIB <n>`

Attractive effects can be produced with this primitive, which allows you to select the graphics option of the BBC BASIC `PLOT` statement. The available

values of <n> are summarised in the following table, together with their effects (the default value is 8). Unlike BBC BASIC (where the intervening values have different effects) Logo does not use the least significant three bits of <n> so that the values of <n> may at first seem strange. For example, if 18 is used, it will have the same effect as 16.

<n>	Effect
0	Draws line including last point
8	Draws line but omits last point
16	Draws dotted line
24	Draws dotted line but omits last point
32	Reserved for Graphics Extension ROM
64	Plots a single point only
72	Reserved
80	Plots and fills a triangular area between the current turtle position and the last two points visited
88–255	Reserved for future expansion

Examples

The range of effects you can produce is quite large and some of the effects are spectacular. Below are some examples:

```
SETNIB 16
SQUARE 300
```

```
SETNIB 80
SQUARE 300
```

```
SETNIB 64
SQUARE 300
```

SETPC

`SETPC <n>`

Changes the logical pen colour to the value represented by `<n>` (see chapter 15, 'Screen modes and the use of colour').

SETPEN

`SETPEN <list>`

Sets the current pen parameters from the values held in `<list>`. The elements of `<list>` are as follows:

`[PENSTATE SHOWN COLOUR NIB PENTYPE]`

- `PENTYPE` — Defines use of colour (see the `SETPT` primitive in this chapter).
- `NIB` — Graphics option (see the `SETNIB` primitive in this chapter).
- `COLOUR` — Pen colour as defined in table 15.2 in section 15.2.
- `SHOWN` — Whether or not turtle is visible (`TRUE` or `FALSE`).
- `PENSTATE` — One of the values `PD`, `PU`, `PE` or `PX`.

Example

```
HOME
SETPEN [PD FALSE 1 80 0]
```

```
FORWARD 200
LEFT 90
FORWARD 200
```

SETPOS

`SETPOS <list>`

Moves the turtle to the position given by `<list>`, where `<list>` is in the form of x-, y-coordinates. The position must be on the screen, unless window

mode is in use, when x and y must be in the range –10000 to 10000.

After any movement the turtle's heading is unchanged. If the pen is down a line will be drawn.

Example

HOME

SETPOS [0 150]

SETPOS [150 0]

SETPT

```
SETPT <n>
```

Defines the way that colour is to be plotted upon the screen (see chapter 15, 'Screen modes and the use of colour', for a full description).

SETSCR

```
SETSCR <number>
```

Allows you to change the aspect ratio of the screen (the ratio vertical step/horizontal step). It is intended to be used when squares appear like rectangles on your display. `<number>` can take any positive value other than zero.

Examples

The following command line makes each vertical step half the length of a horizontal one:

```
SETSCR .5
```

The following command line returns the aspect ratio to normal:

```
SETSCR 1
```

SETSH

```
SETSH <object>
(SETSH <object1> <object2>...)
```

Defines the shape of the turtle. See chapter 12, 'Turtle shapes and multiple

turtles', for full details.

SETX

`SETX <number>`

Moves the turtle horizontally to the point with the x-coordinate `<number>` and leaves the y-coordinate of the turtle unchanged. The point must be on the screen, unless window mode is in use, when `<number>` must be in the range –10000 to 10000.

If the pen is down a line will be drawn. The turtle's heading will be left unchanged.

Example

HOME

SETX 200

FD 150

SETX 0

SETY

`SETY <number>`

Moves the turtle vertically to the point with the y-coordinate `<number>` and leaves the x-coordinate of the turtle unchanged. The point must be on the screen, unless window mode is in use, when `<number>` must be in the range −10000 to 10000.

If the pen is down, a line will be drawn. The turtle's heading will be left unchanged.

Example

```
HOME
RT 45
```

```
SETY 100
```

```
FD 50
```

```
SETY 50
```

SH

Returns a list which defines the turtle's shape. See chapter 12, 'Turtle shapes and multiple turtles', for full details.

SHOWTURTLE(ST)

Makes the turtle visible (see `HIDETURTLE` also).

Example

```
HOME HIDETURTLE
FD 200  LT 120
FD 200
```

```
SHOWTURTLE
```

STAMP

Stamps the turtle's shape on the screen.

Example

The following procedure stamps a series of turtle shapes in a circle around the home position:

```
TO STAMP.CIRCLE
PENUP
SETPOS [180 -40]
REPEAT 12 [FORWARD 100 LEFT 30 STAMP]
HOME
END
```

TITLE

```
TITLE <object>
(TITLE <object1> <object2>...)
```

Prints text on the graphics screen in the turtle's pen colour and at the turtle's position; it does not output a RETURN. If the greedy form of the primitive is used, no spaces will be put between the objects.

The turtle does not move during the operation.

Example


```
TITLE [HELLO]
```

TOWARDS

```
TOWARDS <list>
```

Returns a heading that would make the turtle face the position given by `<list>`. `<list>` is in the form `[x y]` and the heading returned is between zero degrees and 360 degrees.

WINDOW

Removes any restrictions on the turtle's 'territory' that may have been imposed

using FENCE and WRAP.

The screen becomes a window that shows only part of the field in which the turtle can move. If the turtle moves outside this window you can still make it move and turn, but you cannot see it.

The new field has a measurement of 10000 steps from the home position along both the x and y axes and is about 20 times the size of the screen. Movements past the new boundary will wrap around.

Example

```
WINDOW
CS
FORWARD 600
RIGHT 170
FORWARD 300
```

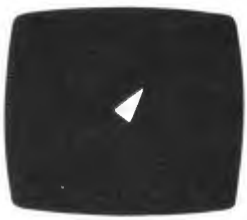
WRAP

Places a fence around the screen, but arranges that when the turtle hits the fence it reappears on the opposite side of the screen.

Example

```
WRAP
CS
RT 30
FD 2000
```

DRAW
WRAP



RT 30

FD 2000

XPOS

(XPOS <n>)

Returns the x-coordinate of the current turtle position. If you are using multiple turtles you can find the x-coordinate of turtle <n> using the 'greedy' form of the primitive shown above. Otherwise, it needs no inputs.

Example

```
SETPOS [100 200]
PRINT XPOS
100
```

YPOS

```
(YPOS <n>)
```

This operation returns the y-coordinate of the current turtle position. If you are using multiple turtles you can find the y-coordinate of turtle `<n>` using the 'greedy' form of the primitive shown above. Otherwise, it needs no inputs.

Example

```
SETPOS [100 200]
PRINT YPOS
200
```

3 Flow of control

The first three sections of this chapter describe the ways in which you can change the flow of control in Logo procedures. The last section describes in detail the primitives that help you do this.

3.1 Repetition

If you want to execute a list of instructions a number of times, you can do this using the REPEAT primitive:

```
TO SPIN
REPEAT 12 [LEFT 30 SQUARE]
END

TO SQUARE
REPEAT 4 [FORWARD 200 LEFT 90]
END
```

The second lines of both SPIN and SQUARE tell the computer to execute the primitives inside the lists 12 and 4 times respectively.

If you don't know how many times you want a sequence repeated you can use the DOFOREVER primitive:

```
TO POLY :ANGLE
DOFOREVER [FORWARD 200 LEFT :ANGLE]
END
```

This repeats the primitives inside the brackets indefinitely to draw a closed figure. :ANGLE determines the type of figure drawn: it is the angle of turn (the external angle). The table on page 38 shows some of the types of closed figure that can be drawn.

:ANGLE	Number of sides	Type of figure
45	8	Octagon
60	6	Hexagon
72	5	Pentagon
90	4	Square
120	3	Triangle

If you want to stop POLY executing you should hold down the CTRL key and then press ESCAPE.

3.2 Conditionals

If you want a number of actions to be performed only if a certain condition is true, you can use 'conditionals' to do this. Look at the following procedure, for example:

```
TO CHECK.SIGN "NUMBER
IF :NUMBER < 0 [OUTPUT "NEGATIVE] [OUTPUT "POSITIVE]
END

PRINT CHECK.SIGN 25
POSITIVE
PRINT CHECK.SIGN -25
NEGATIVE
```

The second line checks the number that is input. If a particular condition is true (in this case, if NUMBER is negative), the contents of the first square brackets are executed; if it is false (NUMBER is positive), the contents of the second brackets are. The IF primitive is called a 'conditional'. It tests the truth of a condition.

The above procedure could also look like this:

```
TO CHECK.SIGN :NUMBER
IF :NUMBER < 0 [OUTPUT "NEGATIVE]
OUTPUT "POSITIVE
END
```

The second list after IF can be omitted if the IF primitive is placed last on the line, as this example shows. Alternatively, you could use a null list [] in place of the second list.

Conditionals give a way of breaking out of DOFOREVER loops. For example:

```
TO SPIRAL
MAKE "SIDE 100
DOFOREVER [FD :SIDE LT 120 MAKE "SIDE :SIDE + 20 IF
:SIDE > 400 [STOP]]
END
```

This procedure draws spiral triangles until SIDE is greater than 400, then the STOP primitive in the square brackets is obeyed and returns control to the caller of the procedure.

3.3 Recursion

This is another way of repeating a series of actions when you do not know how many repetitions will be necessary. The procedure SPIRAL in the last section could now be designed in the following, more elegant way:

```
TO SPIRAL :SIDE
FD :SIDE
LT 120
SPIRAL :SIDE + 20
END
```

Here, SPIRAL calls itself on the last line but one, the 'recursive line', and draws lines indefinitely. You can make it stop by putting a conditional expression in it as follows:

```
TO SPIRAL :SIDE
IF :SIDE > 400 [STOP]
FD :SIDE
LT 120
SPIRAL :SIDE + 20
END
```

3.4 Summary of primitives

Primitive	Effect
ALLOF	Returns TRUE if all its inputs are true, otherwise it returns FALSE
ANYOF	Returns TRUE if at least one of its inputs is true, otherwise it returns FALSE
BREAK	Breaks out of REPEAT or DOFOREVER
CATCH	Runs a list. If a THROW is called during its execution, control returns to the command after CATCH
DOFOREVER	Repeats a list of actions forever, or until an interruption occurs
GO	Transfers control to the command following a label
IF	Executes one of two lists of primitives, depending upon the truth of a condition
IFFALSE	Executes a list if most recent TEST was false
IFTRUE	Executes a list if most recent TEST was true
LOOP	Returns to beginning of REPEAT/DOFOREVER list, incrementing the repeat count if REPEAT is used
NOT	Returns TRUE if input is false and FALSE if input is true
OUTPUT	Returns a value to the calling environment
REPEAT	Repeats a list of primitives
RUN	Runs a list of primitives
STOP	Stops procedure and returns control to calling environment
TEST	Notes if an expression is true or false
THROW	(See CATCH, above)

3.5 Primitives

ALLOF

```
ALLOF <expression1> <expression2>
(ALLOF <expression1> <expression2> <expression3> ...)
```

Returns TRUE if both <expression1> and <expression2> are true, otherwise it returns FALSE.

Examples

```
PRINT ALLOF (2 = 3) (4 = 4)
FALSE
PRINT ALLOF (2 = 2) (4 = 4)
TRUE
```

ANYOF

```
ANYOF <expression1> <expression2>
(ANYOF <expression1> <expression2> <expression3> ...)
```

Returns `TRUE` if at least one of `<expression1>` and `<expression2>` is true, otherwise it returns `FALSE`.

Examples

```
PRINT ANYOF (2 = 3) (4 = 4)
TRUE
PRINT ANYOF (2 = 3) (4 = 5)
FALSE
PRINT ANYOF (2 = 2) (4 = 4)
TRUE
```

BREAK

Breaks out of `REPEAT` or `DOFOREVER` loop.

Example

The following procedure prints a word on the screen continuously until you press the 'A' key. It uses the `RC` primitive described in chapter 6, 'Input/output'.

```
TO READ.UNTIL.A
DOFOREVER [IF NOT KEYQ [BREAK] [IF RC = "A [PRINT
"CUSTARD! STOP]]]
PRINT "RHUBARB
READ.UNTIL.A
END
```

CATCH

```
CATCH <name> <list>
```

This runs `<list>`. If `THROW <name>` is called during its execution, control returns to the command after the `CATCH` primitive. If `CATCH "TRUE` is used, this will catch any `THROW`.

CATCH "ERROR catches an error which would otherwise print an error message and return to command level. When errors are caught, the error message that would normally have been printed is suppressed and you can use the primitive ERROR to return information to your procedures.

CATCH "ESCAPE allows you to control the use of the ESCAPE key.

See chapter 10, 'Handling keyboard errors and debugging' for a complete description and an example.

DOFOREVER

```
DOFOREVER <list>
```

Repeats <list> forever, or until one of the following occurs:

1. A BREAK, LOOP, OUTPUT or STOP is encountered.

2. An error occurs.

3. A THROW or GO is executed and moves control out of the list.

4. ESCAPE is pressed.

Example

```
DOFOREVER [LT 15 SQUARE]
```

GO

```
GO <name>
```

Transfers control to the instruction following the label <name> in the same procedure. <name> is normally a quoted word and can be any valid name. Labels are declared in the form:

```
LABEL1:
        ...
        ...
LABEL2:
        ...
        ...
```

IF

```
IF <expression> <list1>
IF <expression> <list1> <list2>
```

In the first form of IF, if <expression> is TRUE, <list1> will be executed; if <expression> is FALSE the next command will be executed. IF must be the last command on the line.

In the second form of IF, if <expression> is TRUE, <list1> will be executed; if <expression> is FALSE, <list2> will be executed.

In both cases, if <list1> or <list2> generated an output, the value output will be passed back to the calling statement.

Examples

The following procedure tests for the letter 'A' being input. Three different forms of the procedure are used.

IF used to control execution:

```
TO DECISION :TEXT
IF :TEXT = "A [PRINT "YES STOP]
PRINT "NO
END

DECISION "B
NO
DECISION "A
YES
```

A different method for the same result:

```
TO DECISION :TEXT
IF :TEXT = "A [PRINT "YES] [PRINT "NO]
END

DECISION "B
NO
DECISION "A
YES
```

IF used to return a result:

```
TO DECISION :TEXT
OUTPUT IF :TEXT = "A ["YES] ["NO]
END

PRINT DECISION "B
NO
PRINT DECISION "A
YES
```

IFFALSE

```
IFFALSE <list>
```

If the result of the most recent TEST in the current procedure was FALSE, this primitive executes <list>, otherwise it does nothing (see also IFTRUE).

Example

The following procedure tests if a number input is positive or negative:

```
TO SIGN :NUMBER
TEST :NUMBER < 0
IFTRUE [OUTPUT "NEGATIVE]
IFFALSE [OUTPUT "POSITIVE]
END

PRINT SIGN 25
POSITIVE

PRINT SIGN 100 - 330
NEGATIVE
```

IFTRUE

```
IFTRUE <list>
```

If the result of the most recent TEST in the current procedure was TRUE, this primitive executes <list>, otherwise it does nothing (see also IFFALSE).

LOOP

This returns control to the beginning of the REPEAT or DOFOREVER list. In the case of REPEAT, it increments the repeat count.

Example

The following procedure reads ten characters. If a capital A is typed it will print:

```
CAPITAL A TYPED
```

If a small A is typed it will print:

```
SMALL A TYPED
```

```
TO A.LOOP
REPEAT 10 [LOCAL "CH RC IF :CH = "A [TYPE "CAPITAL]
[IF :CH = "a [TYPE "SMALL]] [LOOP]] PRINT " ^ A^ TYPED]
END
```

NOT

```
NOT <expression>
```

Returns TRUE if <expression> is false and FALSE if it is true.

Examples

```
PRINT NOT (2 = 2)
FALSE
PRINT NOT (2 = 4)
TRUE
```

OUTPUT(OP)

```
OUTPUT <object>
```

This is meaningful only when it is within a procedure. It makes <object> the output of the procedure and passes control back to the environment (procedure or command line) that called it.

Example

```
TO AVERAGE :NUMBER1 :NUMBER2
OUTPUT (SUM :NUMBER1 :NUMBER2)/2
END

PRINT AVERAGE 10 20
15
PRINT AVERAGE 20 25
22.5
```

REPEAT

```
REPEAT <n> <list>
```

This primitive runs <list> <n> times, unless one of the following occurs:

1. A BREAK, LOOP, OUTPUT or STOP is encountered.
2. An error occurs.
3. A THROW or GO is executed and moves control out of the list.
4. ESCAPE is pressed.

<n> can be zero, but not negative.

Example

```
REPEAT 12 [LT 30 SQUARE]
```

RUN

```
RUN <list>
```

Executes a list of primitives.

Example

```
TO CALCULATE
PRINT RUN READLIST
CALCULATE
END

CALCULATE
5 + 2
7
12 = 4 * 4
FALSE
12 = 3 * 4
TRUE
```

STOP

This is only allowed within a procedure. It stops the procedure and returns control to the point at which it was called.

Example

```
TO COUNTDOWN :NUMBER
IF :NUMBER = 0 [STOP]
PRINT :NUMBER
COUNTDOWN :NUMBER - 1
END

COUNTDOWN 4
4
3
2
1
```

TEST

```
TEST <expression>
```

This tests whether `<expression>` is `TRUE` or `FALSE` and remembers the result in case there is a subsequent call to `IFTRUE` or `IFFALSE`. Each use of `TEST` is local to the procedure in which it is used.

Example

```
TO QUIZ
PRINT [WHAT IS THE CAPITAL OF FRANCE?]
TEST READLIST = [PARIS]
IFTRUE [PRINT [THAT'S RIGHT!] STOP]
IFFALSE [PRINT [SORRY! TRY AGAIN]]
QUIZ
END
```

```
QUIZ
WHAT IS THE CAPITAL OF FRANCE?
MARSEILLES
SORRY! TRY AGAIN
WHAT IS THE CAPITAL OF FRANCE?
PARIS
THAT'S RIGHT!
```

THROW

`THROW <name>`

This has meaning only when used with the `CATCH` primitive. Its use is described in section 10.2.

`THROW "TOPLEVEL` returns control to the highest command level.
`THROW "LEVEL` returns control to the most recent command level.

4 Using the editor

Acornsoft Logo contains an interactive editor which allows you to modify your procedures and variables in a very straightforward way. You can also define a procedure using the editor, instead of via `TO`. Sometimes this is more convenient; if you make a mistake when using `TO` you have to use the editor to correct it anyway.

4.1 Editing procedures

The editor can operate on one procedure at a time, or a group of procedures. To edit one or more procedures you use `EDIT`, to edit all of your procedures you use `EDPS`.

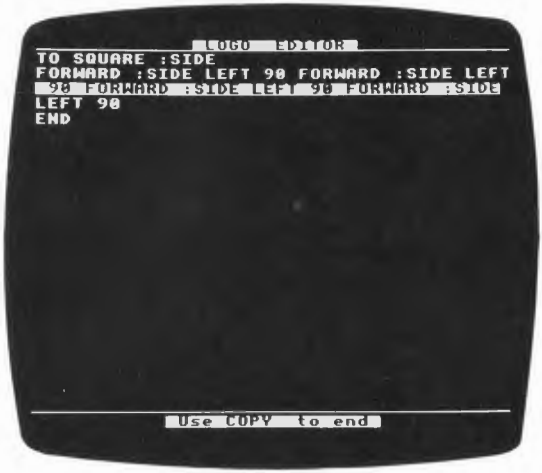
As an example, to edit the procedure `SQUARE` (which we will assume has already been defined) you type:

```
EDIT "SQUARE
```

and the following would be displayed.

You can move around this text using the arrow keys at the right-hand side of your keyboard, and you can change the text using the small number of keys described in section 4.5. Characters are inserted at the cursor position.

If any of the lines overflow the width of the screen, they will be continued in reverse video on the second and subsequent lines. For example:



Unless you have a 6502 Second Processor, the editor will always change from whatever text mode you are using to mode 6, the 40 character text mode; it will also leave you in mode 6 on exit.

As you add or delete lines, the last line of text moves down, or up, respectively. When the line at the bottom of the screen is reached, the text scrolls upwards. If you then try to move the cursor up off the top of the screen, the text will scroll downwards.

To get out of the editor and preserve the changes you have made you should press COPY. To get out of it and leave the procedure as it was before you started, you should press ESCAPE.

When you leave the editor, the text is still left in the 'edit buffer' and you can re-enter the editor by typing:

```
EDIT
```

The edit buffer is preserved until one of the following situations occurs:

1. You start to use turtle graphics.

2. You change the screen mode to something other than mode 6 or 7.

When you leave the editor the screen will be left in mode 6, unless you are using a 6502 Second Processor, and the `TS` primitive will subsequently use this mode. If you are using a 6502 Second Processor, the screen mode will be left unchanged from before you used the editor.

If you exit from the editor using COPY and you are not editing or defining a procedure, any primitives in the edit buffer will be executed immediately.

If you want to edit several procedures together you must put their names into a list as follows:

```
EDIT [PROCA PROCB]
```

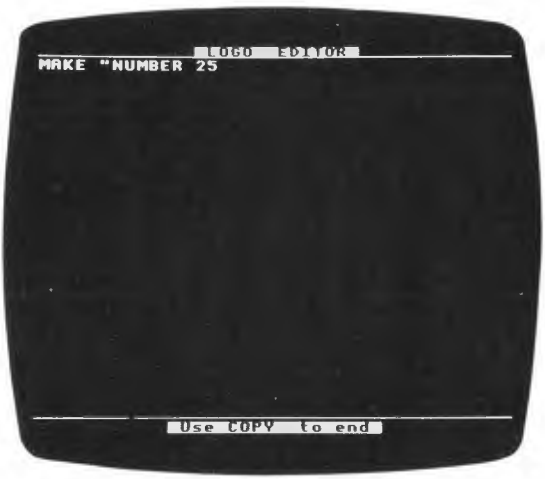
4.2 Editing variables

The editor can operate upon one name (variable) at a time or a number of names. To edit one or more names you use **EDN**, to edit all of your names you use **EDNS**.

If you want to edit the name **NUMBER** (which might have the value 25, say), you would type:

```
EDN "NUMBER
```

and the edit screen will be displayed. This looks similar to the screen described in the last section, but instead of a procedure being displayed you will see a **MAKE** primitive with **NUMBER** as its input:



You can now edit the name, just as you would a procedure.

4.3 Summary of primitives

Primitive	Effect
EDALL	Edits all procedures and names in workspace
EDIT	Edits a procedure or list of procedures
EDN	Edits one or more names
EDNS	Edits all names
EDPS	Edits all procedures

4.4 Primitives

EDALL

Puts all names and procedures into the edit buffer and allows you to edit them using the keys described in section 4.5.

EDIT(ED)

```
EDIT <object>
```

Puts the procedure or procedures specified by <object> into the edit buffer and allows you to edit it/them using the keys described in section 4.5. <object> can be a word or a list. If <object> is absent, the current contents of the edit buffer will be displayed.

If <object> does not currently exist, the edit screen will be displayed and a title line for <object> will be inserted; you can then create a new procedure <object> using the editor.

Examples

```
EDIT "CIRCLE
EDIT [SQUARE CIRCLE TRIANGLE]
```

EDN

```
EDN <object>
```

Puts the variable or variables specified by <object> into the edit buffer and allows you to edit it/them using the keys described in section 4.5. <object> can be a word or a list. If <object> is absent, the current contents of the edit buffer will be displayed.

If <object> does not currently exist, the edit screen will be displayed and a

MAKE primitive for <object> will be inserted. If you exit from the editor using COPY, this primitive and any other ones you put into the edit buffer will be executed immediately, unless one of them defines a procedure.

Examples

```
EDN "SIDE
EDN [SIDE ANGLE]
```

EDNS

This primitive is similar to EDN, but it allows you to edit all names.

EDPS

This primitive is similar to EDIT, but it allows you to edit all procedures.

4.5 Editing keys

Note that some functions use the FUNC key on the Electron or the CTRL key on the BBC Microcomputer. This is indicated below by CTRL/FUNC.

Function	Actions necessary
Move cursor to left	Press the ← key
Move cursor to right	Press the → key
Move cursor up one row	Press the ↑ key
Move cursor down one row	Press the ↓ key
Move cursor to start of Logo line	Hold the CTRL/FUNC key down then press the ← key
Move cursor to end of Logo line	Hold the CTRL/FUNC key down then press the → key
Move cursor to top of text	Hold the CTRL/FUNC key down then press the ↑ key
Move cursor to bottom of text	Hold the CTRL/FUNC key down then press the ↓ key
Insert line	Move the cursor to any point on the Logo line above the one you want to insert then hold the CTRL/FUNC key down and press N simultaneously, or move the cursor to the end of the previous line and press RETURN

Function	Actions necessary
Delete character at cursor position	Hold the CTRL/FUNC key down then press the D key
Delete character before cursor	Press the DELETE key
Delete from cursor to end of line	Hold the CTRL/FUNC key down and press L simultaneously
Delete line	Move the cursor to any point on the line, then hold down CTRL/FUNC and press U
Close up lines	Put the cursor at the start of the empty line then press DELETE
Escape from the editor without altering the original procedure(s)/ name(s)	Press the ESCAPE key
Exit from the editor and preserve the edited procedure(s)/name(s)	Press the COPY key

5 Workspace management

5.1 Introduction

When you define a procedure it is stored in the computer's memory (the 'workspace') until the machine is switched off; it is then destroyed. If you want to keep a set of procedures for future use, you must save them into disc or tape 'files', using the `SAVE` command. You can then load them back into the computer later using the `LOAD` command.

You can examine the variables and procedures in your workspace using various primitives described in this section. You can also erase them from the workspace or modify them using other primitives.

When you are listing procedures and variables the computer is normally in 'scroll mode'. In this state, the text will scroll up when more lines are displayed than would fit onto the screen. You can get it to display a 'page' at a time by holding down CTRL and pressing N. Subsequent pages can be displayed by pressing SHIFT. You can restore scroll mode by holding down CTRL and pressing O.

Some of the primitives in this chapter print procedures and variables on the text screen. If you wish, you can send this output to a printer, as well, by holding down the CTRL key then pressing B. When you have finished, you can disable printer output by holding down the CTRL key and pressing C.

As you add more procedures and data to the workspace, it fills up. When Logo finds that there is no more room in the workspace, it tries to make room by a process known as 'garbage collection'. During this process, procedures or data items which are no longer needed are erased from the workspace. Garbage collection can be observed as a short pause at intervals while your program is running. If such a pause would be inconvenient during a particular operation, you can force garbage collection beforehand using the primitive `TIDY`.

You can check the state of the workspace at any time using the primitive `WS`. This returns a list of two numbers, the first being the total number of free bytes in the workspace, the second the maximum workspace available for any one item such as a list.

5.2 Summary of primitives

Primitive	Effect
CAT	Catalogues disc or tape
ERALL	Erases all procedures and variables from workspace
ERASE	Erases one or more procedures from workspace
ERFILE	Erases a file
ERN	Erases one or more variables from workspace
ERNS	Erases all variables (names) from workspace
ERPS	Erases all procedures from workspace
LOAD	Loads the contents of a file into workspace
PO	Prints definition of one or more procedures
POALL	Prints definition of all procedures and contents of all variables in workspace
PONS	Prints name and value of every variable in workspace
POPS	Prints definition of every procedure in workspace
POTS	Prints title line of every procedure in workspace
READPICT	Reads a picture from a file
SAVE	Saves all or some procedures and variables in workspace into a file
SAVEPICT	Saves a picture in a file
TIDY	Performs a garbage collection
WS	Reports the state of the workspace

5.3 Primitives

CAT

```
CAT <word>
```

Prints the catalogue of the drive specified by `<word>`. `CAT` by itself prints the catalogue of the current drive or tape.

Example

The following command line catalogues drive 1:

```
CAT 1
```

ERALL

Erases all procedures and variables currently in the workspace. Property lists will not be erased by this command; they will be erased by ERPLISTS as described in chapter 14, 'Property lists'.

ERASE(ER)

```
ERASE <object>
```

Erases one or more procedures from the workspace.

Example

The following command line erases the SQUARE procedure:

```
ERASE "SQUARE
```

The following line erases the procedures SQUARE and TRIANGLE:

```
ERASE [SQUARE TRIANGLE]
```

ERFILE

```
ERFILE <filename>
```

Erases the file <filename>.

Example

```
ERFILE "MYFILE
```

ERN

```
ERN <object>
```

Erases one or more variables from the workspace. <object> can be a word or a list.

Example

The following command line erases the variables SIDE and ANGLE:

```
ERN [SIDE ANGLE]
```

Note that only the most recent occurrence of the name is erased:

```
TO FRED :N
PRINT :N
ERN "N
PRINT :N
END
```

```
MAKE "N 4
FRED 6
6
4
```

ERNS

This erases all variables from the workspace. Note that `ERN` erases only the most recent occurrence of a particular variable whereas `ERNS` erases all occurrences of all variables.

ERPS

This erases all procedures from the workspace.

LOAD

`LOAD <filename>`

Loads the contents of the file `<filename>` into the workspace. The file to be loaded must be either an extension or have been saved using the `SAVE` command. Note that loading a file may redefine existing procedures and variables.

If the file loaded includes the procedure `LOADINIT` (as a procedure that takes no parameters) then `LOADINIT` will be executed just before `LOAD` returns. The circumstances in which this could be useful are described under `SAVE`, in this chapter.

Example

```
LOAD "TURPROG
```

PO

`PO <object>`

Prints the definition of one or more procedures on the screen. `<object>` can be a word or a list.

Example

```
PO "TRIANGLE
TO TRIANGLE
REPEAT 3 [FD 200 LT 120]
END
```

POALL

Prints the definition of every procedure and the contents of every variable currently in the workspace. Pressing ESCAPE will abandon `POALL`.

PONS

Prints the name and value of every variable currently held in the workspace. Pressing ESCAPE abandons `PONS`.

Example

```
PONS
"CITY is "DURHAM
"NUMBER is 337
"SHIP is [TRAMP LINER TUG FERRY]
```

POPS

Prints out the definition of every procedure in the workspace. Pressing ESCAPE abandons `POPS`.

POTS

Prints out the title line of every procedure in the workspace. Pressing ESCAPE abandons `POTS`.

Example

```
POTS
TO SQUARE :SIDE
TO RECTANGLE :SIDE1 :SIDE2
TO COUNTDOWN :NUMBER
```

READPICT

`READPICT <filename>`

Copies the picture in the file `<filename>` onto the screen. The file to be loaded will usually have been saved using the `SAVEPICT` command.

Note that this primitive might change the screen mode, number of text lines, palette and screen type (fence, wrap or window).

Example

```
READPICT "PICTURE
```

SAVE

`SAVE <filename> <object>`

Creates the file `<filename>` and saves procedures and variables into it. `<object>` can be a word or a list.

If `<object>` is omitted, all procedures and variables will be saved into `<filename>`. If `<object>` is present, all variables will be saved into `<filename>` but the only procedures which will be saved are the ones specified by `<object>`.

If a procedure called `LOADINIT` is saved, then when `<filename>` is loaded again, `LOADINIT` will be executed automatically. This could be used to:

1. Set up a particular environment for the procedures (see chapter 16, 'Creating a Logo environment').

2. Run the procedures automatically on loading.

Note that `LOADINIT` must have no inputs for it to be executed automatically.

See also the `LOAD` command.

Example

```
SAVE "TURPROC
SAVE "MYFILE [SQUARE TRIANGLE]
```

SAVEPICT

`SAVEPICT <filename>`

Creates the file `<filename>` and saves into it the screen picture. See also the `READPICT` command.

Example

```
SAVEPICT "PICTURE
```

TIDY

When Logo runs short of workspace it automatically clears out early versions of variables which have been changed and procedures which have been deleted. If you want to do this yourself before a time-dependent activity, you can do so using `TIDY`.

WS

Reports on the state of the workspace. `WS` returns a list of two integers. The first is the total number of bytes free in the workspace. The second is the maximum workspace available for one individual item such as a list. If space is running short, workspace may be freed automatically by garbage collection or by calling `TIDY`.

6 Input/output

This chapter describes the primitives that you can use for communication between the computer and the outside world through the keyboard, screen, RS423 channel, printer port, A-D channels, loudspeaker, etc.

6.1 Summary of primitives

Primitive	Effect
ADVAL	Accesses analogue to digital converter channels
BEEP	Generates brief sound from loudspeaker
BUTTONQ	Tells you if a joystick is pressed
CI	Clears keyboard input buffer
CT	Clears text area of screen
CURSOR	Returns text cursor position [x y]
ENVELOPE	Controls pitch and volume of sound
INKEY	Inputs key value if key is pressed within a given time
KEYQ	Tests if key is pressed but does not wait
PRINT	Prints object(s) in text area followed by a new line
PRSCREEN	Copies screen to printer
RC	Reads next character from keyboard
READLIST	Returns line from keyboard as list
READWORD	Returns a word from the keyboard
SETCURSOR	Places text cursor at a given position
SHOW	Prints object on screen followed by carriage return and brackets, if it is a list
SOUND	Generates sounds from the speaker
TIME	Returns time since computer switched on or last TIMERESET
TIMERESET	Resets time counter to zero
TS	Reserves entire screen for text
TYPE	Prints object(s) in text area without adding a new line
VDU	Sends codes to VDU driver
WAIT	Stops execution for a given time

6.2 Primitives

ADVAL

```
ADVAL <n>
```

This primitive is equivalent to the ADVAL operation of BBC BASIC: it allows you to access the analogue to digital converter channels of the computer.

If <n> is in the range one to four, ADVAL returns the value of that channel as an integer between 0 and 4095. If <n> is anything else then it is equivalent to the BBC BASIC ADVAL.

Full details of ADVAL are given in the User Guide for your computer.

Examples

The following command line will print the number of free spaces in the printer buffer:

```
PRINT ADVAL -4
```

The following procedure allows you to control the turtle using a joystick:

```
TO DRIVE
RIGHT (2048 - ADVAL 1) / 64
FORWARD (ADVAL 2) / 128
DRIVE
END
```

BEEP

Generates a brief sound from the computer's loudspeaker.

BUTTONQ

```
BUTTON <n>
```

Returns the value TRUE if the button on the appropriate joystick is pressed, otherwise it returns the value FALSE. The joystick is identified by <n> and this has the following significance:

<n>	**Meaning**
1	Button on joystick 1
2	Button on joystick 2

Any other value of <n> is treated as an error.

CI

Clears the keyboard input buffer. Any keys pressed before `CI` is issued will be forgotten.

CT

Clears the text area of the screen and puts the cursor at its upper left hand corner. The graphics area is not cleared.

CURSOR

Returns the text cursor position as a list of its x- and y-coordinates.

You can set the cursor to a specific text position using `SETCURSOR`.

ENVELOPE

This primitive is identical to the `ENVELOPE` operation of BBC BASIC: it is used with the `SOUND` operation to control the volume and pitch of a sound and it has 14 parameters. Full details are given in the User Guide for your computer.

Examples

```
ENVELOPE 1 1 4 -4 4 10 20 10 127 0 0 -5 126 126
```

The following commands give a warbling 'ray gun' noise:

```
ENVELOPE 2 1 96 0 0 100 100 100 127 -2 -1 -1 126 0
REPEAT 5 [SOUND 2 2 1 25]
```

INKEY

`INKEY <n>`

If `<n>` is in the range:

`0 <= <n> <= 3276`

`INKEY` waits for that number of tenths of seconds or until a key is pressed. If no key is pressed, the empty word is returned; if a key is pressed, the one-character word `CHAR <code>` is returned, where `<code>` is the ASCII value of the key. If `<n>` is greater than 3276 an error is generated.

If `<n>` is negative, a specific key is tested and the value `TRUE` is returned if that key is currently pressed; otherwise, the value `FALSE` is returned.

Example

The following command line waits for up to a second and puts the value of any key pressed into CHARACTER:

```
MAKE "CHARACTER INKEY 10
```

KEYQ

This primitive returns the value TRUE if a key has been pressed and its value has not been used by RC, READWORD or READLINE; otherwise, it returns the value FALSE.

Example

The following procedures allow you to control the movement of the turtle using only four keys; this could be suitable for use by small children.

```
TO TURTLEMOVE
GETKEY
TURTLEMOVE
END

TO GETKEY
MAKE "KEY TESTKEY
CI
IF :KEY = "L [LEFT 15]
IF :KEY = "R [RIGHT 15]
IF :KEY = "F [FORWARD 20]
IF :KEY = "B [BACK 20]
IF :KEY = "D [DRAW]
END

TO TESTKEY
IF KEYQ [OUTPUT RC]
OUTPUT "
END
```

Here, the procedure TESTKEY checks to see if a key has been pressed and returns the empty word or the value of the key pressed.

PRINT(PR)

```
PRINT <object>
(PRINT <object1> <object2> ...)
```

This takes one or more words or lists and outputs them at the text cursor position; it then outputs a RETURN. A space is output between successive items.

PRINT is similar to TYPE but it inserts spaces between items and it ends the text with a RETURN.

Examples

```
PRINT SENTENCE [THIS IS] [A LONG LIST]
THIS IS A LONG LIST
PRINT SENTENCE "THIS [IS ANOTHER LIST]
THIS IS ANOTHER LIST
(PRINT "SO "IS "THIS)
SO IS THIS
```

PRSCREEN

This primitive copies the contents of the screen to the printer. Both the graphics and text areas are copied. If the screen is in modes 3, 6 or 7, this primitive does nothing. If ESCAPE is pressed the printout is abandoned.

PRSCREEN is supplied as part of each printer extension, for example, EPSON. You must load the appropriate extension for your printer.

Sample of PRSCREEN output

RC

This primitive reads the next character from the keyboard; if none is available, it waits until something is typed. The character is not shown on the screen.

For an example of its use, see the description of the primitive KEYQ in this chapter.

READLIST(RL)

This operation returns a line that is read from the keyboard in the form of a list. The line is shown on the screen.

You can use the normal BBC Microcomputer or Acorn Electron line editing facilities as you are inputting the line.

Example

```
TO WATER
PRINT [WHAT SEPARATES BRITAIN FROM FRANCE?]
IF RL = [ENGLISH CHANNEL] [PRINT [CORRECT!] STOP]
PRINT [NO, TRY AGAIN]
WATER
END
```

```
WATER
WHAT SEPARATES BRITAIN FROM FRANCE?
THE COMMON MARKET
NO, TRY AGAIN
WHAT SEPARATES BRITAIN FROM FRANCE?
ENGLISH CHANNEL
CORRECT!
```

READWORD(RW)

This operation returns the first word of a line that is read from the keyboard. The line is shown on the screen. If the line has no characters when RETURN is pressed, an empty word is returned.

You can use the normal BBC Microcomputer or Acorn Electron line editing facilities as you are inputting the line.

SETCURSOR

`SETCURSOR <list>`

This primitive places the text cursor at the position given by `<list>`. `<list>` has the column number as the first element and the line number as the second. Line and column numbering on the screen depend upon the screen mode, but the top left-hand corner of the text area is always `[0 0]`.

Example

```
SETCURSOR [20 12]
```

SHOW

```
SHOW <object>
```

Prints the contents of `<object>` on the screen, followed by a carriage return. If `<object>` is a list, the list brackets are printed around it.

Examples

```
SHOW SENTENCE [THIS IS] [A LONG LIST]
[THIS IS A LONG LIST]
```

```
SHOW "THIS
THIS
```

SOUND

```
SOUND <channel> <amplitude> <pitch> <duration>
```

This is equivalent to the `SOUND` command in BBC BASIC. It is used to make the BBC Microcomputer or Acorn Electron generate sounds from the internal loudspeaker. Full details are given in the User Guide for your computer.

Example

The following command line will play a note on sound channel 1 with a loudness of −15 (maximum volume), a pitch value of middle C and a duration of one second (20 twentieths of a second):

```
SOUND 1 -15 53 20
```

TIME

Returns the time (in tenths of a second) since one of the following events occurred:

1. The computer was switched on.

2. BREAK was pressed.

3. The `TIMERSET` operation (see below) was last used.

The time returned is accurate to within one tenth of a second. The time 'wraps round' to zero at 26214 (after 43 minutes, 41.44 seconds).

Examples

```
PRINT TIME
5312
```

```
DRAW
REPEAT 10 [FORWARD 32 TITLE TIME]
```

TIMERESET

This primitive resets the time counter to zero.

Example

```
PRINT TIME
5312
TIMERESET
PRINT TIME
10
```

The time count returned would have been zero if we could type a little faster!

TS

Reserves the entire screen for text and clears the entire screen. It may change the screen mode.

TYPE

```
TYPE <object>
(TYPE <object1> <object2> ...)
```

This primitive takes one or more words or lists and outputs their contents at the text cursor position; it does not output a RETURN. The text cursor moves to the end of the printed text.

TYPE is similar to PRINT, but it does not insert spaces between items or end the text with a RETURN.

Example

The following procedure types a message followed by a space. It then moves the turtle forward a distance specified by the user. Note the presence of the ^ character; this tells Logo that a special character (in this case the space) follows.

```
TO MOVETURTLE
TYPE [HOW MANY STEPS SHOULD I TAKE?]
TYPE " ^ / (space)
FORWARD RW
MOVETURTLE
END
```

```
MOVETURTLE
HOW MANY STEPS SHOULD I TAKE? 100
HOW MANY STEPS SHOULD I TAKE? 50
HOW MANY STEPS SHOULD I TAKE? -150
```

VDU

```
VDU <object>
(VDU <object1> <object2>...)
```

This is equivalent to the `VDU` command of BBC BASIC: it allows you to send codes to the VDU driver of your computer.

Each input can be:

1. A number.

2. A list, each of whose items is either a number or `;`.

3. The word `";`.

The input `";` can occur only after an input which is a number.

Specifying a list is exactly like specifying each item of the list as a separate input.

Examples

The following command line turns the printer on and copies all text typed subsequently to it:

```
VDU 2
```

This one turns it off:

```
VDU 3
```

The following procedure changes the turtle's shape to a 'pencil':

```
TO DEFPEN
VDU [23 224 0 0 0 0 0 1 2 2]
VDU [23 225 0 0 0 0 0 128 64 32]
VDU [23 226 5 4 8 8 16 17 33 34]
VDU [23 227 32 192 64 128 128 0 0 0]
VDU [23 228 66 68 228 248 240 224 192 128]
SETSH [11 228 11 8 226 227 11 8 8 224 225]
END
```

WAIT

```
WAIT <n>
```

This command stops the program running for `<n>` tenths of a second or until you press ESCAPE.

Example

The following procedure draws a hexagon. After the procedure draws each side it makes the loudspeaker beep and waits for two and a half seconds before it draws the next side.

```
TO HEXAGON :SIDE
REPEAT 6 [FORWARD :SIDE LEFT 60 BEEP WAIT 25]
END
```

7 Procedures and variables

A variable can be regarded as a 'box' containing a word or a list. You can put information into variables in two ways: by making them inputs (or parameters) to procedures (see section 1.5) or by assigning values to them using the MAKE and LOCAL primitives.

This chapter describes the primitives used to put information into these variables. Primitives connected with procedures are also described here.

7.1 Summary of primitives

Primitive	Effect
COPYDEF	Copies a procedure definition and renames it
DEFINE	Allows you to write procedures that define other procedures
DEFINEDQ	Tests if a word is a procedure name
END	Defines the end of a procedure
LOCAL	Makes a variable local to procedure within which LOCAL is used
MAKE	Assigns a value to a variable
PRIMITIVEQ	Tests if a command is a primitive
TEXT	Extracts text of a procedure in the form of a list
THING	Returns contents of a variable
THINGQ	Checks if a name is a variable
TO	Defines a procedure and names its inputs

7.2 Primitives

COPYDEF

```
COPYDEF <newname> <name>
```

Copies the definition of the procedure <name> and gives it the name <newname>. If <name> is a procedure rather than a primitive, <newname> is part of the workspace and can be erased or redefined. If it is a primitive, <newname> is also a primitive and cannot be saved, erased or redefined. One use of COPYDEF might be to shorten the name of a primitive.

COPYDEF cannot be used to copy the definitions of operators such as + and -.

Example

The following command line gives the procedure F the same definition as the primitive FORWARD:

```
COPYDEF "F "FORWARD
```

DEFINE

```
DEFINE <name> <list>
```

Allows you to write procedures that define other procedures. <name> is the procedure to be defined; <list> helps with the definition and it consists of the following sub-lists:

List	Contents
First sub-list	Inputs to the procedure. If there are no inputs this must be the empty list
Second sub-list	First line of the procedure
Third sub-list	Second line of the procedure
Fourth sub-list	Third line of the procedure
etc.	

The primitive TEXT returns the contents of a procedure definition in the above form.

Example

The following procedure defines another procedure which draws spirals. Note that there are no quotes or colons on the inputs and there is no END primitive.

```
TO DEFSPI
DEFINE "SPI [[SIDE ANGLE] [FD :SIDE RT :ANGLE] [SPI
:SIDE + 10 :ANGLE]]
END
```

DEFINEDQ

```
DEFINEDQ <word>
```

Returns the value TRUE if <word> is the name of a procedure or primitive, otherwise it returns the value FALSE.

END

Tells Logo that the definition of a procedure which began using `TO` is complete. `END` must be on a line by itself.

LOCAL

```
LOCAL <name> <item>
```

This primitive hides any previous occurrence (if one exists) of `<name>` from the current procedure or list (for example, `REPEAT`) and establishes a new one containing `<item>`. The previous value is restored in the following circumstances:

1. On leaving the procedure or list in the normal way.

2. When a `THROW` transfers control to a procedure at a higher level.

3. When `ERN` is used to erase the name.

4. When CTRL and ESCAPE are pressed together.

`LOCAL` should always be used for data that is local to a procedure; it makes writing large programs much easier.

MAKE

```
MAKE <name> <object>
```

Assigns the value `<object>` to `<name>`.

Examples

```
MAKE "PET "DOG
PRINT :PET
DOG

MAKE "NUMBERS [12 24 36 48]
PRINT :NUMBERS
12 24 36 48
```

PRIMITIVEQ

```
PRIMITIVEQ <object>
```

Returns `TRUE` if `<object>` is the name of a primitive, otherwise it returns `FALSE`.

Examples

```
PRINT PRIMITIVEQ "FORWARD
TRUE

PRINT PRIMITIVEQ "SQUARE
FALSE
```

TEXT

`TEXT <name>`

Returns the definition of `<name>` as a list of lists. The output is suitable for input to `DEFINE`.

Example

In this example we assume that the procedure `SPIRAL` is defined as follows:

```
TO SPIRAL :SIDE
FD :SIDE
LT 90
SPIRAL :SIDE + 20
END

TO DEFSPI
DEFINE "SPI TEXT "SPIRAL
END

DEFSPI
```

will result in `SPI` having the same definition as `SPIRAL`.

THING

`THING <item>`

Returns the contents of `<item>`. `THING` is similar to dots (`:`) but can be applied to expressions whereas dots can only be applied to names.

Example

```
MAKE "SHIP "TRAWLER
PRINT :SHIP
TRAWLER

PRINT THING "SHIP
TRAWLER

PRINT THING (WORD "SH "IP)
TRAWLER
```

THINGQ

```
THINGQ <item>
```

Returns the value TRUE if <item> has some value, otherwise it returns the value FALSE.

Example

```
PRINT THINGQ "ARTIST
FALSE
MAKE "ARTIST "RENOIR
PRINT THINGQ "ARTIST
TRUE
```

TO

```
TO <name>
TO <name> <name1> <name2> ... <namen>
```

Tells Logo that a procedure called <name> is being defined with the inputs <name1>, etc (if present). All subsequent lines up to and including the END line will be stored in the workspace for later execution.

The prompt changes from ? to > to show that Logo is 'learning' a new procedure rather than obeying commands.

The procedure definition can be abandoned by pressing ESCAPE.

8 Arithmetic

Logo gives you a set of facilities which let you add, subtract, multiply and divide numbers. It also provides primitives for trigonometric and other functions.

Numbers are treated as a special category of words. They are handled like text unless the context forces them to be treated as values, for example:

```
PRINT 1000
1000
PRINT 1000 + 500
1500
```

The numbers themselves can be whole numbers or decimal numbers. Acceptable number formats include 9999, 99.99, 0.9, .99, 9.99E33 and 9.99N9. Numbers are rounded to eight significant digits and numbers too large to be stored (typically, larger than 1.0E38) are not accepted.

2E3 means two multiplied by (10 to the power of 3), or 2000. 2N3 means two multiplied by (10 to the power of minus 3), or 0.002. Other examples of the E and N form of numbers are:

Number	Equivalent
6E2	600
2.4E2	240
6N2	0.06
2.4N2	0.024
6E6	6 000 000

Numbers output by primitives do not use the E or N format unless they are larger than 99,999,999 or smaller than 0.1 (but see the `SETDECS` primitive described in this chapter). Leading and trailing zeros are omitted, except that values less than one have a leading zero. Positive numbers are not shown with a + sign.

The biggest number that can be held is 1.7014118E38. The smallest non-zero value is 1.469368N39.

Some primitives require numeric inputs in the form of integers within the more

restricted range of −32768 to 32767. Such primitives round fractions to the nearest integer, for example:

```
REPEAT 4.6 [...]
```

will execute the list five times. The primitives which allow the full range of numbers (apart from the operators +, -, etc) are: ATN, BACK, COS, EXP, EXPLORE, FORWARD, INT, LEFT, LN, PRODUCT, QUOTIENT, REMAINDER, RIGHT, ROUND, SETH, SIN, SQRT (positive numbers only), SUM and TAN.

8.1 Summary of primitives

Primitive	Effect
ASN	Returns arcsine
ATN	Returns arctangent
COS	Returns cosine
DECS	Returns a value which defines the number of decimal places being used
EXP	Returns exponential function
INT	Returns integer part
LN	Returns natural logarithm
NUMBERQ	Tests if an object is numeric
PI	Returns the value of pi
PRODUCT	Returns product of its inputs
QUOTIENT	Returns integer part of a/b
RANDOM	Returns random integer
REMAINDER	Returns remainder of a/b
RERANDOM	Seeds the random number generator
ROUND	Rounds number to nearest integer
SETDECS	Controls the handling of numbers
SIN	Returns sine
SQRT	Returns square root of its input
SUM	Returns the sum of its inputs
TAN	Returns tangent
+	Adds numbers on either side and returns result
-	Subtracts number on right from number on left
*	Returns product of numbers on either side
/	Divides number on left by number on right and returns result
>	Return the result of comparing
<	the numbers on each side (TRUE
=	or FALSE)

ASN

```
ASN <number>
```

Returns the arcsine (inverse sine) of `<number>`. The arcsine is the angle in degrees corresponding to a given sine value. It is in the range −90 to 90.

The arccosine of a number can be calculated by first evaluating its arcsine and then subtracting the result from 90. This gives a value between 0 and 180.

This primitive is in the extension `CALC`.

Examples

```
PRINT ASN 1
90
PRINT ASN 0.8660254
60
```

ATN

```
ATN <number>
```

Returns the arctangent (inverse tangent) of `<number>`. The arctangent is the angle in degrees corresponding to a given tangent value. It is in the range −90 to 90.

Examples

```
PRINT ATN 1
45
PRINT ATN 1.7320508
60
```

COS

```
COS <degrees>
```

Returns the cosine of `<degrees>`.

Examples

```
PRINT COS 30
0.8660254
PRINT COS 60
0.5
```

DECS

Returns a value which indicates the number of decimal places used in calculations (see SETDECS).

EXP

```
EXP <number>
```

Returns the exponential function of <number>, in other words 2.7182818 to the power of <number>.

This primitive is in the extension CALC.

Example

```
PRINT EXP 1
2.7182818
```

INT

```
INT <number>
```

Returns the integer part of <number>; any decimal part is stripped off. No rounding occurs when INT is used (contrast this with the ROUND operation described later in this chapter).

Examples

```
PRINT INT 9.123
9
PRINT INT -9.123
-10
PRINT INT 9.5
9
PRINT INT 9.999
9
```

LN

```
LN <number>
```

Returns the natural logarithm of <number>. This primitive is in the extension CALC.

Example

```
PRINT LN 2.7182819
1
```

NUMBERQ

```
NUMBERQ <object>
```

Returns the value TRUE if the <object> is a number, otherwise it returns the value FALSE.

Examples

```
PRINT NUMBERQ 23
TRUE

PRINT NUMBERQ [23]
FALSE

PRINT NUMBERQ 3.33E20
TRUE

PRINT NUMBERQ "
FALSE

PRINT NUMBERQ []
FALSE
```

PI

Returns the value of pi (ie 3.141592654). This primitive is in the extension CALC.

PRODUCT

```
PRODUCT <number1> <number2>
(PRODUCT <number1> <number2> ... <numbern>)
```

Returns the product of its inputs. It has the same effect as the * operator except that, being a greedy primitive, it can take more than two inputs.

Examples

```
PRINT PRODUCT 10 10
100
PRINT (PRODUCT 1 2 3)
6
```

QUOTIENT

```
QUOTIENT <number1> <number2>
```

Divides <number1> by <number2> and returns the integer part unrounded. If <number2> is zero, an error will be generated.

QUOTIENT and REMAINDER are intended for use with integer arithmetic. However, they also have uses in other cases, for example:

```
MAKE "HEAD REMAINDER :ANGLE 360
```

Examples

```
PRINT QUOTIENT 10 2
5
PRINT QUOTIENT 10 3
3
PRINT QUOTIENT 10 13
0
PRINT QUOTIENT 14 3
4
PRINT QUOTIENT 14 -3
-5
PRINT QUOTIENT -14 3
-5
PRINT QUOTIENT -14 -3
4
```

RANDOM

```
RANDOM <n>
```

Returns a random non-negative integer less than <n>. See also RERANDOM.

You do not get the same sequence of numbers each time the computer is switched on.

Example

RANDOM 2 could output 0 or 1 and could be used to simulate tossing a coin:

```
TO COIN
IF RANDOM 2 = 0 [PRINT "HEADS STOP]
PRINT "TAILS
COIN
END
```

```
COIN
TAILS
TAILS
HEADS
```

The exact number of TAILS will vary.

REMAINDER

`REMAINDER <number1> <number2>`

Divides `<number1>` by `<number2>` and returns the remainder. If `<number2>` is zero an error will be generated. Contrast with `QUOTIENT`.

Examples

```
PRINT REMAINDER 10 2
0
PRINT REMAINDER 10 3
1
PRINT REMAINDER 10 13
10
PRINT REMAINDER -10 3
2
PRINT REMAINDER 14 3
2
PRINT REMAINDER 14 -3
-1
PRINT REMAINDER -14 3
1
PRINT REMAINDER -14 -3
-2
```

RERANDOM

`RERANDOM <n>`
`RERANDOM`

This primitive 'seeds' the random number generator so that it produces a repeatable sequence of random numbers when `RANDOM` is called. Different values of `<n>` give different sequences.

If `<n>` is not specified, this will seed the random number generator with a random number. You can use this to break out of a repeatable sequence.

Example

`RANDOM 2` could output `0` or `1` and could be used to simulate tossing a coin. In the following example, the same sequence would be obtained each time the program is run:

```
TO COIN
IF RANDOM 2 = 0 [PRINT "HEADS STOP]
PRINT "TAILS
COIN
END

TO REP.COIN
RERANDOM 17
COIN
END

REP.COIN
TAILS
HEADS
```

ROUND

```
ROUND <number>
```

This primitive rounds `<number>` to the nearest integer. Contrast the examples given below with those for `INT`.

```
PRINT ROUND 9.123
9
PRINT ROUND 9.5
10
PRINT ROUND 9.999
10
PRINT ROUND -9.5
-10
```

SETDECS

```
SETDECS <n>
```

Controls the handling of numbers. If `<n>` is in the range 0 to 7, the N format of numbers will not be output. Instead, numbers will be rounded to the given number of decimal places. If `<n>` is 8, normal output will be restored.

`SETDECS 0` provides for integer arithmetic.

Example

```
PRINT 1/3
0.33333333
SETDECS 2
PRINT 1/3
0.33
```

Note that SETDECS also affects calculations. For example:

```
SETDECS 2
PRINT (1234/1000) * 100
123
```

SIN

```
SIN <degrees>
```

This primitive returns the sine of <degrees>.

Examples

```
PRINT SIN 30
0.5
PRINT SIN 60
0.8660254
```

SQRT

```
SQRT <number>
```

This primitive returns the square root of <number>. If <number> is negative an error will be generated.

Examples

```
PRINT SQRT 4
2
PRINT SQRT 2
1.4142136
```

SUM

```
SUM <number1> <number2>
(SUM <number1> <number2> ... <numbern>)
```

This primitive returns the sum of its inputs.

Examples

```
PRINT SUM 5 10
15
PRINT (SUM 1 2 3)
6
PRINT (SUM 2.5 5)
7.5
PRINT (SUM 1 2 -4)
-1
```

TAN

```
TAN <degrees>
```

This primitive returns the tangent of `<degrees>`. It is in the extension `CALC`.

Examples

```
PRINT TAN 45
1
PRINT TAN Ø
Ø
```

The + Operator

```
<number1> + <number2>
+<number>
```

This operation returns the sum of its inputs.

Examples

```
PRINT 5 + 1Ø
15
```

The – Operator

```
<number1> - <number2>
-<number>
```

– is treated as the binary form unless one of the following situations occurs:

1. It follows a space and is followed by a number without a space between the two, for example: `(PRINT 3 -2)`

2. The context is such that it must be unary, for example: `PRINT - 2`

Examples

```
PRINT 1Ø  - 3
7
```

```
(PRINT 3 -2)
3 -2
```

The * Operator

```
<number1> * <number2>
```

This operator returns the product of its inputs.

Example

```
PRINT 2 * 3
6
```

The / Operator

`<number1> / <number2>`

This operator returns a value equal to `<number1>` divided by `<number2>`. It returns an error if `<number2>` is zero.

Examples

```
PRINT 10/2
5
```

```
PRINT 10/3
3.3333333
```

The > Operator

`<number1> > <number2>`

This returns the value `TRUE` if the number on its left is greater than the one on its right, otherwise it returns the value `FALSE`.

Examples

```
PRINT 10 > 4
TRUE
```

```
PRINT 10 > 20
FALSE
```

The < Operator

`<number1> < <number2>`

This returns the value `TRUE` if the number on its left is less than the one on its right, otherwise it returns the value `FALSE`.

Examples

```
PRINT 10 < 20
TRUE
```

```
PRINT 10 < 4
FALSE
```

The = Operator

```
<number1> = <number2>
```

This returns the value `TRUE` if the object on its left is equal to the one on its right, otherwise it returns the value `FALSE`.

If both objects are numeric it does a numeric comparison, otherwise it does a textual one. To force a textual comparison, you should place the items in lists.

Examples

```
PRINT 5 = 5
TRUE

PRINT 5 = 10
FALSE

PRINT "1E4 = "10E3
TRUE

PRINT (LIST "1E4) = (LIST "10E3)
FALSE

PRINT "1E4A = "10E3A
FALSE
```

9 Words and lists

Logo has two types of object: 'words' and 'lists'. It also has operations which allow you to join objects together, break them into distinct parts or examine them. The first section of this chapter tells you more about words and lists; we then go on to describe the primitives that you can use with them.

9.1 Introduction

9.1.1 Words

In English and many other languages, groups of letters with an accepted meaning are termed 'words'. In Logo, a similar system applies. You indicate that a Logo object is a word by preceding it with quotes, as in:

```
PRINT "HELLO
HELLO
PRINT "A
A
PRINT "WHAT?
WHAT?
```

The quotes are not part of the word and they must only appear at the start of it. If they are put around the word, as they are in ordinary punctuation, the last quotes will be printed:

```
PRINT "HELLO"
HELLO"
```

You do not, however, have to use quotes before numbers. For example:

```
PRINT 1066
1066
PRINT 3.14159
3.14159
```

Words can be broken into smaller words using the `FIRST`, `LAST`, `ITEM`, `BUTFIRST` and `BUTLAST` primitives. For example:

```
PRINT FIRST "CATS
C
PRINT BUTFIRST "CATS
ATS
```

If you were to type the following:

```
PRINT BUTFIRST "C
```

everything but the first character (C) would be output. You would get a word containing no characters and this is called 'the empty word'. You can use the empty word in your procedures by typing quotes followed by no characters, as in:

```
PRINT "
(empty line)
```

9.1.2 Lists

A list is a sequence of 'elements' separated by spaces. Each element can be either a word or another list:

```
PRINT [THIS IS [A LIST]]
THIS IS [A LIST]
```

The items between the square brackets are a list with the elements:

```
THIS
IS
[A LIST]
```

PRINT strips off the outer square brackets and prints just the elements of the list. The square brackets are a way of identifying the sequence as a list. SHOW displays the list with the brackets intact:

```
SHOW [THIS IS [A LIST]]
[THIS IS [A LIST]]
```

The first two elements are words, the last is another list. Spaces are used only to separate the elements of the list; extra spaces are ignored. For example:

```
PRINT [LOOK                NO      SPACES]
LOOK NO SPACES
```

Words in a list do not need quotes:

```
PRINT "HELLO
HELLO
PRINT ""HELLO
"HELLO
PRINT FIRST [HELLO]
HELLO
PRINT FIRST ["HELLO]
"HELLO
```

You can manipulate lists in a similar manner to words, using the primitives FIRST, LAST, ITEM, BUTFIRST and BUTLAST. However, instead of them operating on the characters of a word to give words, they operate on the list to give words or another list:

```
PRINT FIRST [THIS IS [A LIST]]
THIS
PRINT LAST [THIS IS [A LIST]]
A LIST
```

If you were to type the line:

```
PRINT BUTFIRST [THIS]
```

everything but the word THIS would be printed. Logo would give you a list containing no words and this is known as 'the empty list'. You can use the empty list in your programs by typing [], as in the following:

```
PRINT []
```
(empty line)

9.2 Summary of primitives

Primitive	Effect
ADDITEM	Inserts a new element in a list or word
ASCII	Returns ASCII code of its input
BUTFIRST	Returns everything but first element of an object
BUTLAST	Returns everything but last element of an object
CAPS	Changes letters of an object to capitals
CHAR	Returns character whose ASCII code is its input
COUNT	Returns number of elements in a list or word
EMPTYQ	Tests if an object is the empty word or the empty list
ERITEM	Removes an element from a word or list
FIRST	Returns first element of an object
FPUT	Produces a new object by putting an object at the front of an old one
ITEM	Returns an element of a list or word
LAST	Returns last element of an object
LIST	Combines objects to form a list
LISTQ	Tests if object is a list
LPUT	Produces a new object by putting an object at the end of an old one

Primitive	Effect
MEMBER	Tests if an object is part of a word or list and returns element number, if it is
MEMBERQ	Tests if object is an element of a list or word
SENTENCE	Combines objects to form a list
SETITEM	Changes an element of a list or word
WORD	Combines words to form another word
WORDQ	Tests if an object is a word

9.3 Primitives

ADDITEM

```
ADDITEM <n> <object> <newitem>
```

Creates a new object from an old object with <newitem> added at position <n>. If <item> is a word, <newitem> must not be an empty word and only its first character will be used.

Examples

```
MAKE "CAPITALS [OTTAWA WASHINGTON LONDON]
MAKE "CAPITALS ADDITEM 3 :CAPITALS "OSLO
PRINT :CAPITALS
OTTAWA WASHINGTON OSLO LONDON

MAKE "NAME "JON
MAKE "NAME ADDITEM 3 :NAME "H
PRINT :NAME
JOHN
```

ASCII

```
ASCII <character>
```

This returns the ASCII code for <character>; if the word used as input contains more than one character, only the first will be used. Appendix C contains a list of ASCII codes and their equivalent characters.

The primitive CHAR has the reverse effect to ASCII.

Examples

```
PRINT ASCII "A
65
```

```
PRINT ASCII "a
97

PRINT ASCII 1
49
```

BUTFIRST(BF)

```
BUTFIRST <object>
```

This outputs everything but the first element of `<object>`, which can be a word or a list. If you try to use the empty word or the empty list an error will be generated.

Examples

```
PRINT BUTFIRST "CATS
ATS

PRINT BUTFIRST [TORTOISESHELL CATS ARE GREAT]
CATS ARE GREAT

TO TRIANGLE :TEXT
IF :TEXT = " [STOP]
PRINT :TEXT
TRIANGLE BUTFIRST :TEXT
END

TRIANGLE "QUEBEC
QUEBEC
UEBEC
EBEC
BEC
EC
C
```

BUTLAST(BL)

```
BUTLAST <object>
```

This outputs everything but the last element of `<object>`, which can be a word or a list. If you try to use it on the empty word or the empty list an error will be generated.

Examples

```
PRINT BUTLAST "CATS
CAT
```

```
PRINT BUTLAST [ORANGES LEMONS [CITRUS FRUIT]]
ORANGES LEMONS
```

The following procedure reverses the text typed in; it uses the operation **WORD** to combine two words. The inputs to **WORD** are each surrounded by brackets to make the example clearer, but the brackets are not necessary.

```
TO REVERSE :TEXT
IF :TEXT = " [OUTPUT "]
OUTPUT WORD (LAST :TEXT) (REVERSE BUTLAST :TEXT)
END

PRINT REVERSE "HELLO
OLLEH

PRINT REVERSE "THERE
EREHT
```

The third line of **REVERSE** outputs the last letter of **TEXT** joined to the reverse of the rest of **TEXT**, and so on, until an empty word is encountered.

CAPS

```
CAPS <object>
```

This outputs <object> with all the letters in it in upper case.

Examples

```
MAKE "NAME "susan
PRINT CAPS :NAME
SUSAN

PRINT CAPS [[FRED jim] Sheila]
[FRED JIM] SHEILA
```

CHAR

```
CHAR <n>
```

This is the reverse of **ASCII**: it returns a one character word whose ASCII code is <n>.

The ASCII codes and their corresponding characters are listed in Appendix C.

Examples

```
PRINT CHAR 65
A
PRINT CHAR 49
1
```

The following procedure converts a list of ASCII characters back into text form:

```
TO CONVERT.BACK :TEXT
IF :TEXT = [] [STOP]
TYPE CHAR (FIRST :TEXT)
CONVERT.BACK BUTFIRST :TEXT
END

CONVERT.BACK [76 79 71 79]
LOGO
```

In this procedure, the brackets around `FIRST :TEXT` are not needed; they are included merely for the purpose of clarity.

COUNT

`COUNT <object>`

This returns the number of elements in `<object>`, which can be a word or a list.

Examples

```
PRINT COUNT [TORONTO SEATTLE LONDON HARARE]
4
PRINT COUNT "TORONTO
7
PRINT COUNT [[A B C] [D E F]]
2
```

EMPTYQ

`EMPTYQ <object>`

This returns the value `TRUE` if `<object>` is the empty word or the empty list, otherwise it returns the value `FALSE`.

Examples

```
EMPTYQ 12
```
outputs `FALSE`

```
EMPTYQ [DOGS CATS]
```
outputs `FALSE`

```
EMPTYQ "
```
outputs `TRUE`

```
EMPTYQ []
```
outputs `TRUE`

ERITEM

```
ERITEM <n> <oldobject>
```

Generates an object from a given <oldobject> with the element at position <n> of <oldobject> erased.

Examples

```
MAKE "CAPITALS [OTTAWA HARARE WASHINGTON LONDON]
PRINT ERITEM 3 :CAPITALS
OTTAWA HARARE LONDON

MAKE "CONTINENT "AUSTRALASIA
MAKE "CONTINENT ERITEM 8 :CONTINENT
MAKE "CONTINENT ERITEM 8 :CONTINENT
PRINT :CONTINENT
AUSTRALIA
```

FIRST

```
FIRST <object>
```

This returns the first element of <object>, which can be a word or a list. If you try to use the empty word or the empty list, an error will be generated.

Examples

```
PRINT FIRST "NAPOLEON
N
PRINT FIRST [NAPOLEON BONAPARTE]
NAPOLEON
```

FPUT

```
FPUT <object> <oldobject>
```

This takes <oldobject> and produces a new object by putting <object> at the beginning of it. If <oldobject> is a word, <object> must not be an empty word and only its first character is used.

Examples

```
FPUT [THIS IS] [A LONG LIST]
outputs [[THIS IS] A LONG LIST]

FPUT "THIS [IS ANOTHER LIST]
outputs [THIS IS ANOTHER LIST]

FPUT "A "ANOTHER
outputs "AANOTHER
```

```
FPUT "A []
```
outputs [A]

ITEM

```
ITEM <n> <object>
```

This returns the <n>th element of <object>, which can be a word or a list. An error will be generated if <n> is greater than the number of items in <object> or if <object> is the empty list.

Examples

```
PRINT ITEM 1 [ATLANTIC PACIFIC MEDITERRANEAN AEGEAN]
ATLANTIC
PRINT ITEM 3 [ATLANTIC PACIFIC MEDITERRANEAN AEGEAN]
MEDITERRANEAN
PRINT ITEM 3 "ATLANTIC
L
```

LAST

```
LAST <object>
```

This returns the last element of <object>, which can be a word or a list. If you try to use the empty word or the empty list, an error will be generated.

Examples

```
PRINT LAST "ICELAND
D
PRINT LAST [NORTH AMERICA]
AMERICA
```

LIST

```
LIST <object1> <object2>
(LIST <object1> <object2> ...)
```

This returns a list whose elements are <object1>, <object2>, etc. Each element can be a word or another list.

Contrast this primitive with SENTENCE.

Examples

```
LIST [THIS IS] [A LONG LIST]
```
outputs [[THIS IS] [A LONG LIST]]

```
LIST "THIS [IS ANOTHER LIST]
```
outputs `[THIS [IS ANOTHER LIST]]`

```
LIST "AND "ANOTHER
```
outputs `[AND ANOTHER]`

```
LIST "A []
```
outputs `[A []]`

LISTQ

```
LISTQ <object>
```

This returns the value `TRUE` if `<object>` is a list; otherwise it returns the value `FALSE`.

Examples

```
PRINT LISTQ 25
FALSE
PRINT LISTQ [10 20 30 40]
TRUE
PRINT LISTQ []
TRUE
PRINT LISTQ "
FALSE
```

LPUT

```
LPUT <object> <oldobject>
```

This takes `<oldobject>` and produces a new object by putting `<object>` at the end of it. If `<oldobject>` is a word, `<object>` must not be an empty word and only its first character is used.

Examples

```
LPUT [THIS IS] [A LONG LIST]
```
outputs `[A LONG LIST [THIS IS]]`

```
LPUT "THIS [IS ANOTHER LIST]
```
outputs `[IS ANOTHER LIST THIS]`

```
LPUT "A "ANOTHER
```
outputs `"ANOTHERA`

```
LPUT "A []
```
outputs `[A]`

MEMBER

`MEMBER <object1> <object2>`

If `<object>` is an element of `<object2>`, it returns the element number, otherwise it returns zero.

Examples

```
PRINT MEMBER "LONG [A LONG LIST]
2
PRINT MEMBER "WORD [A LONG LIST]
0
```

MEMBERQ

`MEMBERQ <object1> <object2>`

This returns the value `TRUE` if `<object1>` is an element of `<object2>`, otherwise it returns the value `FALSE`. `<object1>` and `<object2>` can be either words or lists. If `<object1>` and `<object2>` are both words, only the first character of `<object1>` is used.

Note that the comparison takes note of upper and lower case differences. For example:

`MEMBERQ "A [a b]`
is false
`MEMBERQ "a [a b]`
is true

Examples

`MEMBERQ "AFRICA [EUROPE AMERICA AFRICA ASIA]`
outputs `TRUE`

`MEMBERQ "AMERICA [EUROPE AMERICA AFRICA ASIA]`
outputs `TRUE`

`MEMBERQ "GREENLAND [EUROPE AMERICA AFRICA ASIA]`
outputs `FALSE`

`MEMBERQ "B "AMERICA`
outputs `FALSE`

`MEMBERQ "A "AMERICA`
outputs `TRUE`

SENTENCE(SE)

```
SENTENCE <object1> <object2>
(SENTENCE <object1> <object2> <object3> ...)
```

This takes objects (which may be words or lists) as inputs and combines them to form one list.

Examples

```
PRINT SENTENCE "CAT "FISH
CAT FISH
PRINT SENTENCE [CAT] [FISH]
CAT FISH
```

```
SENTENCE "MONET [RENOIR LAUTREC WHISTLER]
```
outputs `[MONET RENOIR LAUTREC WHISTLER]`

If an input is a list, `SENTENCE` uses the members of that list, rather than the list itself:

```
(LIST [A [B C]] [D E F] "G)
```
returns `[[A[B C]][D E F]G]`, a list with three elements.

```
(SENTENCE [A [B C]] [D E F] "G)
```
returns `[A [B C] D E F G]`, a list with six elements.

SETITEM

```
SETITEM <n> <object1> <object2>
```

Returns a new object based on `<object1>` with element `<n>` of `<object1>` changed to `<object2>`. If `<object1>` and `<object2>` are both words, `<object2>` must not be an empty word and only the first character of `<object2>` is used.

Examples

```
MAKE "CAPITALS [OTTAWA HARARE WASHINGTON LONDON]
PRINT SETITEM 4 :CAPITALS "OSLO
OTTAWA HARARE WASHINGTON OSLO
```

WORD

```
WORD <word1> <word2>
(WORD <word1> <word2> <word3> ...)
```

This returns a word which is built up from the words input to it.

Examples

```
PRINT WORD "CAT "FISH
CATFISH
PRINT WORD "LOG "O
LOGO
```

WORDQ

```
WORDQ <item>
```

This returns the value TRUE if <item> is a word; otherwise it returns the value FALSE.

Examples

```
PRINT WORDQ 25
TRUE
PRINT WORDQ [10 20 30 40]
FALSE
PRINT WORDQ []
FALSE
PRINT WORDQ "
TRUE
```

10 Handling keyboard errors and debugging

The first section in this chapter describes the facilities available for correcting keyboard mistakes when you are in command mode; the second describes how you can handle errors within your procedures. The third section describes the facilities available to help you with debugging. Finally, the last two sections describe the primitives used for all these functions.

10.1 Correcting keyboard mistakes

A special group of six keys on the right-hand side of your keyboard is used to alter lines on the screen. You can also use them to repeat command lines. The keys are COPY, DELETE and the four arrow keys. Their effect is the same as for the line editor described in the User Guide for your computer.

When you press one of the arrow keys, the computer enters what is known as edit mode. It then displays two cursors: a white block termed the 'write cursor' and the flashing 'read cursor'. Moving the read cursor (using the arrow keys) to a word and then pressing the COPY key will copy the text under the cursor to the new line at the write cursor. When you are part way through copying a line you can move the read cursor to another piece of text and copy this into the new line.

You can also type in new characters and delete them (using DELETE), or delete a line using CTRL U.

Whatever appears in your new line is input to Logo when you press RETURN.

10.2 Handling errors by program

You can trap input errors by using `CATCH` and `THROW`. These can be used for other things beside error handling, but they are described in this chapter for convenience.

The `THROW` primitive can be called when an error is detected by your program. It will then return control to a `CATCH` primitive which may be in a different procedure.

There are five special cases of `CATCH` and `THROW`:

1. `CATCH "ERROR`, which catches an error which would otherwise print an error message and return to command level. When errors are caught, the error

message that would normally have been printed is suppressed and you can use the primitive ERROR to obtain the information for your procedures.

2. THROW "LEVEL, which returns control to the most recent command level.

3. THROW "TOPLEVEL, which allows you to return to command level.

4. CATCH "ESCAPE, which allows you to control the use of the ESCAPE key (but not CTRL and ESCAPE).

5. CATCH "TRUE, which catches all throws other than errors or ESCAPE.

Perhaps the easiest way of explaining the use of CATCH and THROW is with an example:

The procedure SQUARE.PRINT reads numbers from the keyboard and prints their squares. If you type something other than a number, the READNUMBER procedure prints an appropriate message then returns control to SQUARE.PRINT, which carries on working.

```
TO SQUARE.PRINT
CATCH "NOTNUMBER [CALCULATE]
SQUARE.PRINT
END

TO CALCULATE
PRINT [TYPE A NUMBER, PLEASE: ]
PRINT READNUMBER
END

TO READNUMBER
LOCAL "TEXT READLIST
IF EMPTYQ :TEXT [THROW "TOPLEVEL]
IF NOT NUMBERQ FIRST :TEXT [PRINT [NUMBERS ONLY,
PLEASE!] THROW "NOTNUMBER]
IF NOT EMPTYQ BUTFIRST :TEXT [PRINT
[ONLY ONE NUMBER, PLEASE!] THROW "NOTNUMBER]
OUTPUT (FIRST :TEXT) * (FIRST :TEXT)
END
```

10.3 Debugging your procedures

When there is a 'bug' in a program, the program frequently does not fail on the line containing the bug. In fact, sometimes it does not fail at all but runs and produces unexpected results. As well as this, if there is more than one 'bug' in a program they can combine to produce spectacular results.

With these points in mind, you can minimise the need for debugging by designing your programs as collections of 'procedures', each of which is so small that it is unlikely to contain more than one 'bug'. You can then test each procedure independently of the others.

If 'bugs' still occur, they are likely to be caused by interaction between two procedures and you can use the primitives described in the next section to control the execution of your procedures and check what is happening at each stage.

Three other facilities are available to help you check your procedures:

1. If you press the SHIFT key when using graphics commands, your procedure will pause for half a second after each move or turn.

2. If you hold down the CTRL and SHIFT keys when using printing or graphics commands your procedure will pause until you release either of them.

3. You can interrupt a procedure or list (such as a `REPEAT` list) using the ESCAPE key and use `PRINT`, for example, to find out what is happening. You can then continue running it using `CONTINUE(CO)`.

10.4 Summary of primitives

Primitive	Effect
`CATCH`	Runs a list of instructions. If a `THROW` is called during its execution, control returns to `CATCH`
`CONTINUE`	Continues running after a `PAUSE`
`ERRMSG`	Prints an error message
`ERROR`	Returns error number to your procedure
`PAUSE`	Suspends running until `CONTINUE` is typed
`SETERR`	Makes original error appear to occur at the point where you called `SETERR`
`TC`	Displays names of procedures called
`THROW`	See `CATCH`, above
`TRACE`	Controls tracing

10.5 Primitives

CATCH

`CATCH <name> <list>`

This runs `<list>`. If `THROW <name>` is called during its execution, control returns to the command after the `CATCH` primitive.

Special cases of `CATCH` are:

1. `CATCH "ERROR.`

2. `CATCH "ESCAPE.`

3. `CATCH "TRUE.`

and these are described in section 10.2, 'Handling errors by program'. An example of the use of `CATCH` is also given in this section.

CONTINUE(CO)

This resumes running after a `PAUSE` has been executed or ESCAPE has been pressed.

ERRMSG

`ERRMSG <list>`

If `<list>` is a list in the form returned by `ERROR` (see below), this prints the appropriate error message.

ERROR

This primitive returns information about an error which has occurred while a `CATCH "ERROR` is in effect. The information is in the form of a list with two items:

1. The error number (a word). Error numbers are given in Appendix B, 'Logo error messages'.

2. The two parameters of the error or empty lists, if non-existent.

`ERROR` returns this list the first time it is called after the error has occurred, providing Logo has not returned to command level. If `ERROR` is called at any other time, it returns the empty list.

PAUSE

This suspends the execution of a procedure until `CONTINUE` is typed in and tells you that the procedure is suspended. You can then type instructions to debug your procedure (for example, you might type `TRACE 7`, then `CONTINUE`, to trace the execution of part of your procedure).

During a pause you can access all local variables.

SETERR

`SETERR <list>`

If you use `CATCH "ERROR` to check errors, you might decide not to take action upon some errors. You can then use `SETERR` and this will make it appear as though the error occurred at the point where you called `SETERR`. For example:

```
CATCH "ERROR [MYPROCESS]
MAKE "NEWERROR ERROR
MAKE "ERRNO FIRST :NEWERROR
IF :ERRNO = 305 [...]
IF :ERRNO = 310 [...]
          .
          .
          .
SETERR :NEWERROR
```

This catches all errors and handles errors 305 and 310, perhaps providing diagnostic information before finishing the program. Other errors (for example, 303) are not handled and `SETERR` generates them again, causing a return to the highest command level.

TC

This primitive name stands for 'Type Calls'. It shows the chain of current procedure calls in the form:

```
(<procedure> <input> <input>...)      (<procedure>...)
(...)
```

The most recent procedure is shown first. `TC` is most useful after a procedure is interrupted using `PAUSE` or the ESCAPE key, since it shows how this point was reached.

THROW

`THROW <name>`

Special cases are `THROW "TOPLEVEL` and `THROW "LEVEL`. These return control to the top command level and the most recent command level respectively.

Otherwise, `THROW` is only used with the `CATCH` primitive described above. Its use is described in section 10.2.

TRACE

`TRACE <n>`

This primitive introduces tracing. `<n>` specifies the trace characteristics:

`TRACE 1` traces every line and gives a trace message.

`TRACE 2` traces every procedure call and gives a trace message (buried procedures are not traced).

`TRACE 4` traces every primitive and buried procedure call and gives a trace message.

`TRACE 8` stops after every trace message and waits for you to press RETURN.

These can be combined by addition to give a wide range of tracing information. For example,

`TRACE 7`

traces lines, procedures and primitives.

`TRACE 15`

stops after any line, procedure or primitive.

`TRACE 0` produces no tracing information and this is the default state. To change the trace characteristics while tracing, you must first stop the program using ESCAPE. You can then enter a new `TRACE` command.

Example

The best way to understand tracing is to run through an example and observe what happens on the screen. Try the following:

```
TO SQUARE :SIZE
REPEAT 4 [SIDE :SIZE]
END
```

```
TO SIDE :LENGTH
FD :LENGTH RT 90
STOP
END

TRACE 7
SQUARE 200
TRACE 0
```

11 Floor turtles

Extensions are available for several different floor turtles. These are held on the disc or tape which accompanies the Acornsoft Logo package.

For example, the extension for the *BBC Buggy* is loaded by typing:

```
LOAD "BUGGY
```

Once the extension is loaded, you need to tell Logo that subsequent commands apply to the floor turtle, instead of the screen turtle. This is done by typing: `FLOOR`. You can stop using the floor turtle and continue using the screen turtle by typing: `SCREEN`.

The floor turtle will respond to the primitives summarised in section 11.1. Most of these are described in chapter 2, 'Turtle graphics'. The remainder are described in section 11.2.

Note that, when using floor turtles, if you try to use graphics primitives which are not supported (for example, `SETPT`) these will usually have no effect. If you try to use screen commands which return information (for example, `POS`), an error will be generated.

11.1 Summary of primitives

Primitive	Effect
`BACK`	Moves turtle backwards (`BACK 1` moves the turtle back by about 1mm)
`EXPLORE`	Moves turtle forward until an obstacle is encountered
`FLOOR`	Activates a floor turtle
`FORWARD`	Moves turtle forwards (`FD 1` moves the turtle forward by about 1mm)
`HOOT`	Activates speaker
`LEFT`	Turns turtle to left
`PENDOWN`	Puts turtle's pen down
`PENUP`	Lifts turtle's pen
`PENUPQ`	Tests if turtle's pen is up
`RIGHT`	Turns turtle to right

Primitive	Effect
SCREEN	Restores the screen turtle
SCREENQ	Tests if screen turtle is in use
SENSE	Tests if turtle is touching anything

11.2 Primitives

EXPLORE

EXPLORE <number>

Moves the turtle forward by <number> steps. If an obstacle is encountered before this, it stops and returns the number of steps travelled. If the turtle does not have appropriate sensors, EXPLORE can be terminated by pressing the ESCAPE key on the computer.

FLOOR

Activates the floor turtle and stops the screen turtle. The driver for a floor turtle must first have been loaded from the disc or tape which accompanies the Acornsoft Logo package, otherwise an error will be generated.

HOOT

Activates the hooter on the floor turtle, if one exists, otherwise it causes a BEEP at the computer.

SCREEN

Stops subsequent commands being applied to the floor turtle and addresses them to the screen turtle.

SCREENQ

Tests if floor or screen turtle is in use. If the screen turtle is in use, SCREENQ returns TRUE, otherwise it returns FALSE.

SENSE

SENSE <n>

Returns the value TRUE if turtle sensor <n> is touching anything, otherwise it returns the value FALSE.

For sensors on some turtles, SENSE returns a number. Refer to the documentation on individual floor turtle extensions for details.

12 Turtle shapes and multiple turtles

Acornsoft Logo allows you to change the turtle's shape and drive several turtles around the screen at the same time. This chapter tells you how to do both of these things.

12.1 Changing the turtle's shape

You can change the turtle's shape using the SETSH primitive. For example, the following command changes the turtle shape to the letter A:

```
SETSH ASCII "A
```

You can also do this with the following:

```
SETSH 65
```

Each character which you type in at the keyboard has an associated 'ASCII code'. When the computer is told to use this character it looks up the code and treats it as an 8 x 8 matrix of dots. The letter A, as used above, has the ASCII code 65. Other codes are shown in Appendix C, 'ASCII code table'.

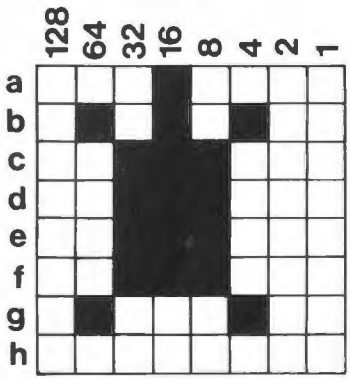
SETSH can take a list as input and this is shown by the following examples, which change the turtle's shape to the letters shown on their right:

```
SETSH [65 66]                    AB

SETSH [65 66 67 68]              ABCD

SETSH [65 66 8 8 10 67 68]       AB
                                 CD
```

In the last example, 8 is the ASCII code for 'backspace' and 10 is the code for 'linefeed'.

Certain ASCII code values have been left to be defined by the user. They include the values 224 to 255 and they can be defined using the VDU command, then used by SETSH to create more picturesque shapes.

Suppose you wanted to make a turtle pattern. You should first plan the character on an 8 x 8 square grid as shown below.



To store this shape as code number 240, type in the following:

```
VDU [23 240 16 84 56 56 56 56 68 0]
```

The numbers which follow `VDU [23 240` tell the computer the pattern of dots in each horizontal row of the grid. For example, row a has the value 16, while row c has the value 56 (32 + 16 + 8).

Once you have typed in the line shown above, you can redefine the turtle's shape by typing:

```
SETSH 240
```

The `EDSHAPE` example on the disc or tape which accompanies your Logo package provides another way of building shapes.

The following procedure changes the turtle's shape twice to give the effect of a bird flying. It swaps between two shapes with the wings in the up and down positions respectively.

```
TO FLY
VDU [23 224 129 66 36 24 24 0 0 0]
VDU [23 225 0 0 0 60 90 129 129 0]
WINDOW RT 45 PU
DOFOREVER [SETSH 224 SETSH 225 FD 20]
END
```

You can also build more complex shapes by using a number of user-defined characters with SETSH. For example, the following procedure turns the turtle into a pen shape:

```
TO DEFPEN
VDU [23 224 0 0 0 0 0 1 2 2]
VDU [23 225 0 0 0 0 0 128 64 32]
VDU [23 226 5 4 8 8 16 17 33 34]
VDU [23 227 32 192 64 128 128 0 0 0]
VDU [23 228 66 68 228 248 240 224 192 128]
SETSH [11 228 11 8 226 227 11 8 8 224 225]
END
```

You can restore the triangular turtle shape at any time by typing:

```
SETSH
```

12.2 Multiple turtles

Acornsoft Logo allows you to have up to 32 turtles on your screen at the same time. Using this facility you might, for example, direct one turtle onto a randomly moving target using the keyboard. Or you could reverse this game by letting a number of turtles home in upon one turtle that is controlled via the keyboard or a joystick.

Turtles are 'hatched' using HATCH, and they are created at the current turtle position. You 'talk' to one or more of them using TELL. When they are newly hatched, they are invisible; you must use TELL and then SHOWTURTLE before you can see them. When you are finished with one or more turtles, you can remove them using FORGET.

The special primitives which control multiple turtles are described later in this chapter. Otherwise, the primitives you can use are the normal turtle graphics primitives. Most of these are no different whether you are using one or more turtles. However, a few of them (for example, POS) need information about one turtle and so cannot apply to a list of turtles. If the last TELL command contained a list of turtles, these primitives will use the first turtle in the list. All primitives of this kind are greedy for a turtle number; thus, (POS 2) returns information about turtle 2, regardless of which turtles are currently selected.

The accompanying book, *Introduction to Logo on the BBC Microcomputer and Acorn Electron*, shows a number of ways that you can use multiple turtles with turtle shapes redefined as the letters of the alphabet. In the present chapter we will show you two more sophisticated uses of multiple turtles.

First of all, you need to load the multiple turtle extension from the disc or tape which accompanies your Logo package. You do this by typing:

```
LOAD "MULT
```

Now type the following:

```
TO FLAGS
DRAW
MAKE "N 1
REPEAT 11 [HATCH :N TELL :N RT 30 ST MAKE "N :N + 1]
TELL TURTLES
FD 200
SETNIB 80
REPEAT 4 [RT 90 FD 50]
HT
END
```

This example creates twelve turtles. It then produces some attractive moving effects using a simple drawing pattern together with `SETNIB`. To work out what it is doing, and why it is called `FLAGS`, remove the `SETNIB` command and put a few `PAUSE` commands into it at appropriate points.

The next example uses multiple turtles to show how a number of words can be sorted into alphabetical order using a 'bubble sort'. It allows you to input five words and displays them above one another on the screen. They are then compared, two at a time. The pair of words being compared is highlighted in yellow and, if they are in order, they are changed back to white. If they are out of order, they are highlighted in green and one of them moves out to the right. One then moves down while the other moves upwards, and they are returned to the list. Finally, the colours are reset to white.

```
TO CODEL :W
OUTPUT IF EMPTYQ :W [[]] [FPUT ASCII :W CODEL BF :W]
END

TO ORDEREDQ :A :B
IF EMPTYQ :A [OUTPUT "TRUE]
IF EMPTYQ :B [OUTPUT "FALSE]
IF FIRST :A = FIRST :B [OUTPUT ORDEREDQ BF :A BF :B]
OUTPUT ASCII :A < ASCII :B
END
```

```
TO SWAP
SETPC 1 TELL :P
FD 400 SETH 180
TELL :N SETH 0
TELL SE :P :N FORWARD 100 SETH 90
TELL :P BACK 400
TELL SE :P :Q
END

TO COMPARE :X :Y
MAKE "P ITEM :X :ORD
MAKE "N ITEM :R :ORD
TELL SE :P :Q
SETPC 2 WAIT 20
TEST ORDEREDQ THING WORD "W :P THING WORD "W :Q
IFFALSE [SWAP MAKE "SWAP "TRUE MAKE "ORD ADDITEM :X
ERITEM :R :ORD ITEM :R :ORD]
SETPC 3
END

TO SORT
MAKE "J 4
BACK:
MAKE "I 1
MAKE "SWAP "FALSE
REPEAT :J [COMPARE :I :I + 1 MAKE "I :I + 1]
IF :SWAP [MAKE "J :J  - 1 GO "BACK]
END

TO START
DRAW
SETMODE 5
PAL 1 2
PU SETPOS [-500 -350] SETPC 2 TITLE [The Bubble Sort]
SETPC 7
HATCH [1 2 3 4]
MAKE "I 0
REPEAT 5 [TELL :I PU SETX -300 SETR 300  - 100 * :I
SETH 90 ST MAKE "I :I + 1]
PR [Input 5 words]
MAKE "I 0
REPEAT 5 [TYPE "> MAKE WORD "W :I RW TELL :I
```

```
SETSH CODEL THING WORD "W :I MAKE "I :I +1]
MAKE "ORD [0 1 2 3 4]
SORT
END
```

In the examples of multiple turtles where TELL has been applied to a number of turtles at once, each subsequent command will be applied to all turtles at the same time. If, instead, you want to apply a list of commands first to one turtle, then to another, and so on, you can do so using the following procedure, EACH:

```
TO EACH :LST
MAKE "$ WHO \ an unusual name
MAKE "$$ WHO
CATCH "ERROR [DOFOREVER [IF EMPTYQ :$ [TELL :$$ STOP]
[] TELL FIRST :$ RUN :LST MAKE "$ BUTFIRST :$]]
TELL :$$
SETERR ERROR
END
```

12.3 Summary of primitives

Primitive	Effect
ALIVEQ	Tests if a given turtle is alive
FORGET	Destroys the named turtle(s)
HATCH	Creates one or more turtles
SETSH	Redefines the turtle's shape
SH	Returns the turtle's shape as a list
TELL	Applies subsequent commands to the turtles specified
TURTLES	Returns list of all 'live' turtles
WHO	Returns the numbers of turtles currently being 'talked to'

12.4 Primitives

ALIVEQ

ALIVEQ <n>

Returns TRUE if turtle <n> is alive, otherwise it returns FALSE.

```
PRINT ALIVEQ 0
TRUE

PRINT ALIVEQ 10
FALSE
```

FORGET

```
FORGET <object>
```

Destroys the turtle or turtles specified by `<object>` and removes it/them from the screen. `<object>` can be an integer or a list of integers.

Turtle 0 cannot be destroyed. If you try to `FORGET` it, the command will be ignored.

Examples

```
FORGET 1

FORGET [1 2 4]
```

HATCH

```
HATCH <object>
(HATCH <object> <shape>)
```

Creates the turtle or turtles specified by `<object>`, which can be an integer or list of integers. The new turtles are invisible and can be selected using `TELL`, then made visible using `SHOWTURTLE`. The turtle you started with is called turtle 0.

In the greedy form of `HATCH` shown above, `<shape>` consists of the inputs used by `SETSH`. Otherwise, the turtle takes the shape and other characteristics of the current turtle(s).

Examples

```
HATCH 1

HATCH [1 2 3 4]
```

The following example creates turtle 1 and makes it look like the letters `AB`:

```
(HATCH 1 [65 66])
TELL 1
SHOWTURTLE
```

SETSH

```
SETSH <object>
(SETSH <object>)
```

Changes the turtle's shape to the value given by `<object>`, which can be a word or a list. If `<object>` is omitted then the turtle returns to the triangular shape used initially.

In the greedy form of `SETSH` shown above, `<object>` corresponds to the parameters of the BBC BASIC `VDU` command. It can consist of:

1. A number.

2. A list, each of whose items is either a number or `;`.

3. The word `";`.

Or any combination of these.

Examples

```
SETSH 65
```

```
SETSH [65 66]
```

The following converts the turtle to a line, drawn from the current position (an 'elastic band' effect):

```
(SETSH 25 5 FIRST POS "; LAST POS "; )
REPEAT 50 [FD 10 RT 5]
```

SH

```
SH
(SH <n>)
```

Returns the current turtle shape in the form of a list. If the greedy form of the primitive is used, the shape returned will be that of turtle `<n>`.

Example

```
SETSH [65 66 8 8 10 67 68]
PRINT SH
65 66 8 8 10 67 68
```

TELL

```
TELL <object>
```

Tells Logo which turtle(s) you want to 'talk' to. Turtle commands will be applied to turtle 0 unless you tell Logo otherwise.

Example

The following procedures put four turtles at the main points of the compass and apply subsequent commands to them:

```
TO COMPASS
HATCH [1 2 3]
START 0 START 1 START 2 START 3
TELL [0 1 2 3]
END

TO START :NO
TELL :NO
SHOWTURTLE
RIGHT :NO * 90
END
```

TURTLES

Checks which turtles have been created and returns their numbers in the form of a list.

Example

```
HATCH [1 2 3]
PRINT TURTLES
0 1 2 3
TELL TURTLES
```

WHO

Checks which turtles are currently being 'talked to' using `TELL` and returns their numbers in the form of a list.

Example

```
HATCH [1 2 3 4]
TELL [1 2]
PRINT TURTLES
0 1 2 3 4
PRINT WHO
1 2
```

13 Interface to machine functions

The BBC Microcomputer and Acorn Electron have a wide range of machine functions which can be accessed using 'OSBYTE calls'. They are described and listed in detail in the User Guide for your computer and will not be repeated here.

You can access these functions from Logo using the OSBYTE, CALL, DATAAREA, DEPOSIT, HIBYTE, LOBYTE and EXAMINE primitives. You can also access machine code routines from Logo using the CALL command.

13.1 Summary of primitives

Primitive	Effect
CALL	Calls a machine code routine
DASIZE	Returns the size of the data area in bytes
DATAAREA	Returns byte address of a data area
DEPOSIT	Changes contents of memory
EXAMINE	Examines contents of memory
HEX	Returns decimal value of a hexadecimal number
HIBYTE	Returns the most significant byte of a two byte value
LOBYTE	Returns the least significant byte of a two byte value
OSBYTE	Performs an OSBYTE call

13.2 Primitives

CALL

```
CALL <n>
```

This primitive calls a machine code routine. <n> must be a signed 16-bit integer (it is often convenient to use HEX to specify this address. Thus, to call OSWORD (hex FFF1), either of the following could be used:

```
CALL -15
CALL HEX "FFF1
```

but not:

```
CALL 65521
```

since this is an *unsigned* 16-bit integer.

On entry to the machine code, the A, X and Y registers are set up from bytes 0, 1 and 2, respectively, of **DATAAREA**.

On return, bytes 0 to 3 are set up from the A, X, Y and P registers respectively. If a MOS fault has occurred, byte 3 of **DATAAREA** is set to 255 (the P register can never have this value at any other time) and byte 1 contains the fault number. In this case, bytes 2 and 3 of **DATAAREA** are undefined. If a fault has occurred, any ESCAPE condition will have been acknowledged.

Note that incorrect use of **CALL** can crash the system or cause random errors.

CALL is in the extension **MOS**.

DASIZE

Returns the size of the data area in bytes.

DASIZE is in the extension **MOS**.

DATAAREA

```
DATAAREA
DATAAREA <n>
```

This returns the byte address of a data area for use by your Logo program. The address is a signed integer in the range 0 to 32767 and its position is allocated by Logo. If <n> is specified, an area of size <n> bytes is allocated. If you ask for an area for which there is insufficient memory available, an error will be generated.

Logo can allocate only one such area at a time. If you need space for two or more purposes, you should obtain sufficient space for all purposes and allocate its use within your program.

If the data area is to be used by **CALL** with a command such as the following:

```
CALL :OSWORD
```

bytes 0 to 3 of the data area will be used by **CALL**.

DATAAREA is in the extension **MOS**.

DEPOSIT

```
DEPOSIT <n> <n>
```

This command allows you to change the contents of the computer's memory.

The first input is a byte address and it must be a signed 16-bit integer, as for `CALL`. The second input is the value to be deposited. If this is greater than 255, the least significant eight bits are used.

Note that incorrect use of `DEPOSIT` can crash the system or cause random errors.

EXAMINE

```
EXAMINE <n>
```

This command allows you to look at the contents of the computer's memory.

`<n>` is a byte address and it must be a signed 16-bit integer, as for `CALL`. It can take the form of an 'absolute' address, for example:

```
PRINT EXAMINE 16132
```

or a 'relative' address:

```
PRINT EXAMINE DATAAREA + 23
```

HEX

```
HEX <word>
```

Returns the decimal value of `<word>` as a signed integer. `HEX` is in the extension `MOS`.

Example

```
PRINT HEX "FFF4
-12
```

HIBYTE

```
HIBYTE <n>
```

Returns the most significant byte of a two byte value given by `<n>`.

Example

```
PRINT HIBYTE 1000
3
```

LOBYTE

```
LOBYTE <n>
```

Returns the least significant byte of a two byte value given by <n>.

Example

```
PRINT LOBYTE 1000
232
```

OSBYTE

```
OSBYTE <integer>
OSBYTE <integer> <integer> <integer>
```

Calls the operating system OSBYTE routine. The inputs are, respectively, the A, X and Y registers and the X, Y registers can be omitted.

The value returned is an integer which is made up of the contents of the X and Y registers. The X register contents are in the low byte of the call and can be accessed using LOBYTE. The Y register contents are in the high byte of the call and can be accessed using HIBYTE.

For further details of OSBYTE calls, refer to the User Guide for your computer.

14 Property lists

Logo allows you to build up a simple 'filing system' for yourself using 'property lists'.

Before you can use the primitives which manipulate property lists you must load them from the extension PROP. This is held on the tape or disc which is part of your Logo package. To load it, type:

```
LOAD "PROP
```

You should then type TS to get into text mode.

Suppose you want to build up a record of peoples' names, telephone numbers and other such things. You could start this off as follows:

```
PPROP "JOHN "TELEPHONE [Ø734 55555]
PPROP "JOHN "AGE 12
PPROP "JOHN "HOBBY "FISHING
```

PPROP stands for Put PROPerty. It creates a property list which is connected to the name JOHN. A property list consists of an even number of elements. Each pair of elements consists of the name of a property (for example TELEPHONE, AGE and HOBBY) and its value ([Ø734 55555], 12 and FISHING, respectively). You can look at the entire property list using the PLIST primitive:

```
PRINT PLIST "JOHN
TELEPHONE [Ø734 55555] AGE 12 HOBBY FISHING
```

You can now build up your filing system by adding other entries, for example:

```
PPROP "ANN "TELEPHONE [91 44444]
PPROP "ANN "AGE 13
PPROP "ANN "HOBBY "READING
```

If you want to look at the value of a specific property for one person you can do so using GPROP (Get PROPerty):

```
PRINT GPROP "ANN "AGE
13

PRINT GPROP "JOHN "HOBBY
FISHING
```

You can look at all the properties for one person using PPS:

```
PPS "JOHN
JOHN's TELEPHONE is [0734 55555]
JOHN's AGE is 12
JOHN's HOBBY is FISHING
```

Note the difference between this and the output from PLIST above.

You can also look at all the properties and their values using PPALL:

```
PPALL
ANN's TELEPHONE is [91 44444]
ANN's AGE is 13
ANN's HOBBY is READING
JOHN's TELEPHONE is [0734 55555]
JOHN's AGE is 12
JOHN's HOBBY is FISHING
```

Once you have a filing system you can add new properties to it using PPROP, for example:

```
PPROP "JOHN "HAIR "BROWN
PPROP "ANN "HAIR [ASH BLONDE]

PPALL
ANN's TELEPHONE is [91 44444]
ANN's AGE is 13
ANN's HOBBY is READING
ANN's HAIR is [ASH BLONDE]
JOHN's TELEPHONE is [0734 55555]
JOHN's AGE is 12
JOHN's HOBBY is FISHING
JOHN's HAIR is BROWN
```

You can also change existing properties with PPROP:

```
PPROP "JOHN "HAIR "BLACK

PRINT GPROP "JOHN "HAIR
BLACK
```

With a little bit of effort, you can list a given property for a number of people, together with their names:

```
TO LIST.PROPS :LST :PROP
MAKE "N 1
DOFOREVER [IF :N > COUNT :LST [STOP] [(PRINT ITEM :N
:LST GPROP ITEM :N :LST :PROP) MAKE "N :N + 1]]
END

LIST.PROPS [JOHN ANN] "TELEPHONE
JOHN Ø734 55555
ANN 91 44444

LIST.PROPS [JOHN ANN] "HOBBY
JOHN FISHING
ANN READING
```

If you want to keep a copy of your property lists on disc or tape you can do so using SAVE and LOAD.

14.1 Summary of primitives

Primitive	Effect
ERPLIST	Erases a property name or list of property names, together with their properties
ERPLISTS	Erases all property names and their properties
GPROP	Returns the value of a property for a given name
PLIST	Returns property list associated with a given name
PPALL	Prints all properties of every name
PPROP	Associates a property and its value with a given name
PPS	Prints the properties associated with a name, together with their values
REMPROP	Removes a property and its value from a given name

14.2 Primitives

ERPLIST

```
ERPLIST <object>
```

Erases the property names specified by `<object>`, together with their properties. `<object>` can be either a word or a list.

Examples

```
ERPLIST "JOHN
```

```
ERPLIST [JOHN ANN]
```

ERPLISTS

Erases all property names and their properties.

GPROP

```
GPROP <name> <property name>
```

Stands for Get PROPerty. Returns the value of a property associated with `<name>`. If there is no such property, it returns the empty list.

Example

```
PRINT GPROP "JOHN "TELEPHONE
0734 55555
```

PLIST

```
PLIST <name>
```

Returns the property list associated with `<name>`.

Example

```
PRINT PLIST "ANN
TELEPHONE [91 44444] AGE 13 HOBBY READING
```

PPALL

Prints the property list of every name. Contrast the output with that of `PLIST`.

Example

```
PPALL
ANN's TELEPHONE is [91 44444]
ANN's AGE is 13
ANN's HOBBY is READING
```

```
ANN's HAIR is ASH BLONDE
JOHN's TELEPHONE is [0734 55555]
JOHN's AGE is 12
JOHN's HOBBY is FISHING
JOHN's HAIR is BROWN
```

PPROP

```
PPROP <name> <property name> <object>
```

Stands for Put PROPerty. Gives `<name>` the property `<property name>` with value `<object>`. `<object>` can be a word or a list. `PPROP` can be used to change an existing property, as well as create a new one.

Example

```
PPS "ANN
ANN's HOBBY is READING
PPROP "ANN "HAIR [ASH BLONDE]
PPS "ANN
ANN's HOBBY is READING
ANN's HAIR is [ASH BLONDE]
PPROP "ANN "HOBBY (LIST GPROP "ANN "HOBBY "CYCLING)
PRINT GPROP "ANN "HOBBY
READING CYCLING
```

PPS

```
PPS <object>
```

Stands for Print PropertieS. Prints the property list(s) of everything associated with `<object>`, which can be a word or a list.

Example

```
PPS [ANN JOHN]
ANN's TELEPHONE is [91 44444]
ANN's AGE is 13
ANN's HOBBY is READING
ANN's HAIR is ASH BLONDE
JOHN's TELEPHONE is [0734 55555]
JOHN's AGE is 12
JOHN's HOBBY is FISHING
JOHN's HAIR is BROWN
```

REMPROP

`REMPROP <name> <property name>`

Removes the property `<property name>` and its value from the property list of `<name>`.

Example

```
PRINT PLIST "JOHN
TELEPHONE [0734 55555] AGE 12 HOBBY FISHING
REMPROP "JOHN "HOBBY
PRINT PLIST "JOHN
TELEPHONE [0734 55555] AGE 12
```

15 Screen modes and the use of colour

The BBC Microcomputer and Acorn Electron both have an excellent range of colour graphics and this chapter tells you how to make use of them. Even if you do not have a colour monitor you may still find some parts of it useful: text and graphics will be displayed in different levels of brightness on your screen and you will still be able to use some of the special effects.

The first section in this chapter describes the screen modes you can use; the second gives you the information you need to handle colour on your computer. The last two sections describe the primitives that influence or are influenced by colour.

15.1 Screen modes

The range of colour and the graphics resolution you can use depend upon the graphics mode. This depends, in its turn, upon the equipment you have and how big your program is.

The graphics mode is set using the `SETMODE` primitive and the modes you can use are shown in table 15.1.

Table 15.1 Graphics modes

n	Description
0	This uses two colours with very high resolution graphics and needs 20K of memory to map the screen (16K on US machines).
1	This uses four colours with high resolution graphics and needs 20K of memory to map the screen (16K on US machines).
2	This uses 16 colours with medium resolution graphics and needs 20K of memory to map the screen (16K on US machines).
3	This uses two colours and is a text-only mode. It needs 16K of memory to map the screen.

n	Description
4	This uses two colours with high resolution graphics and needs 10K of memory.
5	This uses four colours with medium resolution graphics and needs 10K of memory.
6	This uses two colours and is a text-only mode. It needs 8K of memory to map the screen (Note: Logo reserves 10K for screen memory, so 2K of this is unavailable).
7	This displays Teletext characters and needs 1K of memory to map the screen (Note: Logo reserves 10K for screen memory and the editor's buffer, so using mode 7 does not free any more space for your programs than mode 6).

Logo is always in one of two 'states':

1. *Graphics Mode:* In this state the screen is used partly for graphics and partly for text.

2. *Text Mode:* In this mode the screen is used entirely for text. This is also the mode used by the editor.

Logo remembers two screen modes:

1. *Default Text Mode:* This is initially the screen mode which was used when Logo was entered. However, if the mode used when Logo was entered was mode 7, mode 6 will be used instead.

2. *Default Graphics Mode:* This is initially the screen mode which was used when Logo was entered. If this was one of the text modes 3, 6 or 7, mode 4 will be selected instead.

The following primitives affect screen modes and the 'states' of Logo:

1. `TS`: This selects the default text mode and the text state. If there is insufficient memory to select this mode, the default text mode will be changed to the current screen mode.

2. `DRAW` and *graphics primitives:* These select the default graphics mode and the graphics state. If there is insufficient memory to select this mode, the default graphics mode will be changed to the current screen mode. If this is a text-only mode (modes 3, 6 or 7), mode 4 will be selected instead.

3. EDIT: Unless a 6502 Second Processor is in use, this primitive changes the default text mode to mode 6 and uses this mode subsequently.

4. SETMODE: This selects the screen mode. If there is insufficient memory to select this mode, nothing happens (note that there is always sufficient memory to select modes 4 to 7).

If Logo is in the graphics state and the new mode is not a text-only mode (3, 6 or 7), the default graphics mode is changed to the new mode and the graphics state remains selected. In all other cases, the default text mode is changed to the new mode and the text state is selected.

15.2 Using colour

In each of the modes mentioned in table 15.2 there are a fixed number of 'logical colours' that you can use. For example, in mode 5 you can have only four logical colours, 0 to 3.

Table 15.2 shows the 'physical colours' which the logical colours are preset to for each mode.

Table 15.2 Preset physical colours

Logical colour numbers					Physical colour	
Mode 0	Mode 1	Mode 2	Mode 4	Mode 5		
0	0	0	0	0	0	black
	1	1		1	1	red
		2			2	green
	2	3		2	3	yellow
		4			4	blue
		5			5	magenta (blue-red)
		6			6	cyan (blue-green)
1	3	7	1	3	7	white
		8			8	flashing black-white
		9			9	flashing red-cyan
		10			10	flashing green-magenta
		11			11	flashing yellow-blue
		12			12	flashing blue-yellow
		13			13	flashing magenta-green
		14			14	flashing cyan-red
		15			15	flashing white-black

The full range of logical colours is available only in mode 2. However, you can redefine a logical colour number in each mode to map onto any physical colour using the PAL (for PALette) primitive. For example, when you are in mode 5, logical colour 3 is preset to white but you can set it to cyan by typing:

```
PAL 3 6
```

The physical colour numbers are shown just to the left of the actual colours in table 15.2.

Logo remembers the physical to logical colour relationships for the default graphics mode and the default text mode. It restores these relationships when entering the graphics state or the text state. When the screen mode corresponding to one of these default modes is changed, the colour relationships for that state are reset to the BBC Microcomputer or Electron defaults.

15.3 Summary of primitives

Primitive	Effect
BG	Returns background colour
MODE	Returns the display mode
PAL	Sets logical colour to a specific physical colour
PC	Returns pen colour
PM	Protects mode when you are switching between different screen modes
SETBG	Changes background colour
SETMODE	Changes the display mode
SETPC	Changes pen colour
SETPT	Defines use of colour on screen

15.4 Primitives

BG

Returns an integer that represents the logical background colour. Logical colours are shown in table 15.2

MODE

Returns the current screen mode.

PAL

```
PAL <logical colour> <physical colour>
```

This primitive sets the logical colour of the BBC Microcomputer and Electron to a specific physical colour. It is equivalent to the BBC BASIC `VDU` code `VDU19`.

Example

The following command line changes logical colour 0 to flashing white-black:

```
PAL 0 15
```

PC

Returns the current pen colour.

PM

```
PB <n>
```

If you need to switch between screen modes, `PM` will ensure that you do not use more space than would be allowed in mode `<n>`. You can thus return to mode `<n>` without any space problems.

SETBG

```
SETBG <n>
```

Changes the logical background colour (initially black, or 0) to the value represented by `<n>`. The graphics area is cleared. `<n>` can be one of the values shown in table 15.2.

SETMODE

```
SETMODE <n>
```

Selects the screen mode which the computer is about to use. Eight modes are available with the BBC Microcomputer and seven with the Electron, and they are shown in table 15.1, together with the appropriate value of `<n>`.

A full description of the effect of `SETMODE` is given in section 15.1.

Example

```
TO SQUARE
REPEAT 4 [FORWARD 200 LEFT 90]
END
```

```
DRAW
SETMODE 4
SQUARE
SETMODE 5
SQUARE
```

SETPC

```
SETPC <n>
```

This primitive changes the pen colour to the value represented by <n>. The values which you can use are shown in table 15.2.

If you have a monochrome monitor the colours will be represented by different intensities.

Example

The following procedures draw a black spinning square shape on a yellow background.

```
TO SPIN
REPEAT 24 [LEFT 15 SQUARE]
END

TO SQUARE
REPEAT 4 [FORWARD 200 LEFT 90]
END

SETMODE 5
SETPC 0
SETBG 2
SPIN
```

SETPT

`SETPT <n>`

Defines the way that colour is to be used on the screen. The input `<n>` has the following effect:

n	Effect
0	Use the colours given by the `SETPC` and `SETBG` primitives
1	OR the pen colour and background colour
2	AND the pen colour and the background colour
3	Exclusive-OR the pen colour and the background colour
4	Invert the colour of the point passed

Examples

The following procedure draws a circle and then erases it:

```
TO CIRCLE.WIPE
SETPT 4
REPEAT 720 [FORWARD 2 LEFT 1]
END
```

16 Creating a Logo environment

In some circumstances you might want to restrict the facilities that Logo offers or extend them in some way. For example, you might want to:

1. Restrict the precision of numbers.

2. Redefine primitives such that `FORWARD 10` moves the turtle by 100 steps instead of 10.

3. Change the initial screen mode and start up colours.

4. Have certain of your procedures treated as primitives in that they cannot be edited by users.

5. Rename primitives for use with other languages.

You can do any of these things by creating a 'Logo environment'.

As an example, suppose you want to set up a turtle graphics environment for young children such that the following applies:

`F` has the same effect as `FORWARD 100`

`B` has the same effect as `BACK 100`

`L` has the same effect as `LEFT 30`

`R` has the same effect as `RIGHT 30`

First of all, you would create the procedures `F`, `B`, `L` and `R` as follows, using the editor:

```
TO F
FORWARD 100
END

TO B
BACK 100
END

TO R
RIGHT 30
END
```

```
TO L
LEFT 30
END
```

Try them out now. When you type F and press RETURN, this will have the same effect as:

```
FORWARD 100
```

The only problem now is this: someone could tamper with these new procedures by using the editor. You can prevent this happening by typing:

```
BURYALL
```

This 'buries' all the procedures in your workspace such that they now look like primitives and cannot be edited. You can 'unbury' them at any time by typing:

```
UNBURYALL
```

If you wish, you can bury only the procedures F, B and R by typing:

```
BURY [F B R]
```

If you want to set up this environment so that it can be easily created when you start up Logo, you can do so by first ensuring that F, B, R and L are unburied, then typing the following:

```
TO LOADINIT
BURYALL
END
```

and then saving your workspace into a file (MYFILE, say):

```
SAVE "MYFILE
```

Note that if procedures are to be saved, they must not be buried when you use the SAVE primitive.

Now, after you have started up Logo, if you type:

```
LOAD "MYFILE
```

the procedures F, B, L and R will be loaded into your workspace and buried automatically. Logo always looks for a procedure LOADINIT after you have used LOAD; if it finds one in the file just loaded, it runs the procedure automatically.

You can put other things into LOADINIT, if you want to. For example, the following will change the screen mode and background colour, as well as bury your procedures:

```
TO LOADINIT
SETMODE 5
SETBG 2
BURYALL
END
```

Suppose you now want to introduce the children to the 'normal' primitives FORWARD and BACK, together with the concept of inputs, but you want to redefine their scope such that FORWARD 10 and BACK 10 both give a movement of 100 steps. First of all, type the following:

```
COPYDEF "F "FORWARD
COPYDEF "B "BACK
REDEF
```

The first two lines create 'copies' of FORWARD and BACK, and REDEF allows you to redefine primitives. You can now change FORWARD by typing:

```
EDIT "FORWARD
```

and modifying FORWARD to look like the following:

```
TO FORWARD :STEPS
F :STEPS * 10
END
```

Alternatively, it would be possible to define FORWARD using the primitive DEFINE, which is described in chapter 7, 'Procedures and variables'.

After modifying BACK in a similar way, you can prevent anyone redefining primitives by typing:

```
NOREDEF
```

You can also prevent them tampering with FORWARD and BACK by typing:

```
BURYALL
```

Note that redefining FORWARD does not affect FD, so you might want to redefine this also. One way would be to use COPYDEF (described in chapter 7, 'Procedures and variables'):

```
REDEF
COPYDEF "FD "FORWARD
NOREDEF
```

There is one other thing which you can do to set up a special Logo environment: you can program the user function keys on your computer using the operating system command *KEY. Type in the following, for example:

```
RUN [*KEY0 "CS:M"]
RUN [*KEY1 "FORWARD "]
RUN [*KEY2 "RIGHT "]
RUN [*KEY3 "LEFT "]
```

If you now press the function key f1, then type `100` and press RETURN, this will have the same effect as:

```
FORWARD 100
```

The other keys are set up to clear the screen and turn the turtle right and left respectively. Note the use of the `:M` on the first line. This forces a RETURN after the `CS` command to save you having to press RETURN yourself.

16.1 Summary of primitives

Primitive	Effect
`BURIEDQ`	Tests if a procedure is buried
`BURY`	Buries a named procedure or list of procedures
`BURYALL`	Buries all the procedures in your workspace
`NOREDEF`	Prevents the redefinition of primitives
`REDEF`	Allows the redefinition of primitives
`REDEFQ`	Tests if primitives can be redefined
`UNBURY`	Unburies a named procedure or list of procedures
`UNBURYALL`	Unburies all the procedures in your workspace

16.2 Primitives

BURIEDQ

```
BURIEDQ <word>
```

Tests if the procedure named by `<word>` is buried. If it is, `BURIEDQ` returns `TRUE`, otherwise it returns `FALSE`.

BURY

```
BURY <object>
```

Buries the named procedures in your workspace such that you cannot save, edit, list or redefine them. This effectively makes the named procedures look like primitives.

Examples

```
POTS
TO R
TO L
TO F
TO B
BURY "F
POTS
TO R
TO L
TO B
```

The following buries the procedures F, B, L and R:

```
BURY [F B L R]
```

BURYALL

Buries all of the procedures in the workspace. Note that buried procedures will not be saved.

NOREDEF

Prevents the redefinition of primitives.

REDEF

Allows the redefinition of primitives.

REDEFQ

Tests if primitives can be redefined. If they can, it returns TRUE, if not it returns FALSE.

UNBURY

```
UNBURY <object>
```

Unburies the named procedures in your workspace such that you can edit, list or redefine them.

Examples

The following command line unburies the procedure L:

```
UNBURY "L
```

The following unburies the procedures F, B, L and R:

```
UNBURY [F B L R]
```

UNBURYALL

Unburies all of the procedures in your workspace.

Appendix A

Logo primitives

A hash symbol (#) indicates that a procedure can take any number of inputs (ie, it is greedy). If you give it more than the number indicated, you must enclose the entire expression in brackets, for example:

```
PRINT (PRODUCT 50 10 15)
```

+ indicates that the function applies to the current turtle, unless a turtle number is enclosed in brackets, for example:

```
(HEADING 3)
```

Primitive	Effect
`ADDITEM <n> <object> <newitem>`	Returns an object made up of the old `<object>` with `<newitem>` added at position `<n>`.
`ADVAL <n>`	If `1<=<n><=4`, returns an integer (0 to 4095) representing the ADC voltage, otherwise returns integer representing value (see the User Guide for your computer).
`ALIVEQ <n>`	Returns `TRUE` if turtle `<n>` is alive.
`#ALLOF <a> <b>`	Returns `TRUE` if both `<a>` and `<b>` are `TRUE`.
`#ANYOF <a> <b>`	Returns `TRUE` if at least one of `<a>`, `<b>` is `TRUE`.
`ASCII <character>`	Returns ASCII code for `<character>`.
`ASN <number>`	Returns the arcsine of `<number>` (in degrees).
`ATN <number>`	Returns the arctangent of `<number>` (in degrees).
`BACK(BK) <distance>`	Moves turtle `<distance>` steps back.
`BEEP`	Generates a brief sound from loudspeaker.
`BG`	Returns number representing background colour.

Primitive	Effect
BREAK	Breaks out of REPEAT or DOFOREVER loop.
BURIEDQ <object>	Returns TRUE if the procedure <object> is buried.
BURY <object>	Buries named procedures in workspace.
BURYALL	Buries all procedures in workspace.
BUTFIRST(BF) <object>	Returns all but first element of <object>.
BUTLAST(BL) <object>	Returns all but last element of <object>.
BUTTONQ <n>	Returns TRUE if button on joystick <n> (1 or 2) is down.
CALL <n>	Passes control to machine code routine at the byte address <n>.
CAPS <object>	Changes any lower case letters in <object> to capitals.
CAT <object>	Catalogues drive specified by <object>.
CATCH <name> <list>	Runs <list>; returns when THROW <name> is encountered or end of <list> is reached.
CHAR <n>	Returns a word containing a single character whose ASCII value is <n>.
CI	Clears the keyboard input buffer.
CLEAN	Clears graphics screen without moving turtle. Can be used to select the graphics mode from text.
CONTINUE(CO)	Resumes a procedure after a PAUSE or interruption by ESCAPE.
COPYDEF <newname> <name>	Copies definition of <name> to <newname>.
COS <degrees>	Returns the cosine of <degrees>.
COUNT <object>	Returns the number of elements in <object>.
CS	Clears screen and homes cursor.

Primitive	Effect
CT	Clears text area of screen.
CURSOR	Returns position of text cursor as list of x,y coordinates.
DASIZE	Returns the size (in bytes) of the data area as an integer.
DATAAREA	Returns byte address of data area used to pass data to operating system and machine code routines. If <n> is specified an area of <n> bytes is reserved.
DECS	Indicates number of decimal places used in calculations.
DEFINE <name> <list>	Makes <list> the definition of procedure <name>.
DEFINEDQ <word>	Returns TRUE if <word> is the name of a procedure or primitive.
DEPOSIT <byteaddress> <byte>	Puts the value <byte> into the address <byteaddress>. Both are integers.
DISTANCE <list>	Returns distance from current turtle position to screen position <list>.
DOFOREVER <list>	Repeats <list> forever, or until a command such as BREAK or STOP is encountered.
DOT <list>	Returns logical colour of dot at screen position given by <list>.
#DRAW <n>	Clears screen, kills all but one turtle, sets WRAP, moves turtle to [0 0], sets heading to 0, resets pen state. If <n> is used, reserves <n> lines for text.
EDALL	Edits all procedures and names in workspace.
#EDIT (ED) <object>	Starts the Logo editor and loads named procedure(s) into the edit buffer. If <object> is not present, displays edit buffer or nothing (if contents have been erased).

Primitive	Effect
#EDN <object>	Puts variable name(s) given by <object> into the edit buffer. If <object> is not present, displays edit buffer or nothing (if contents have been erased).
EDNS	Edits all names in workspace.
EDPS	Edits all procedures in workspace.
EMPTYQ <object>	Returns TRUE if <object> is the empty list or the empty word.
END	Completes definition of a procedure.
ENVELOPE (14 inputs)	Used with SOUND to control volume and pitch of a sound.
ERALL	Erases all procedures and variables from workspace.
ERASE (ER) <object>	Erases procedure(s) named by <object> from workspace.
ERFILE <filename>	Erases the file <filename>.
ERITEM <n> <oldobject>	Returns an object with the same contents as <oldobject> but with element <n> erased.
ERN <object>	Erases most recent version of variable(s) named by <object> from workspace.
ERNS	Erases all variables from workspace.
ERPLIST <object>	Erases property name(s) given by <object>, together with their properties.
ERPLISTS	Erases all property names, together with their properties.
ERPS	Erases all procedures from workspace.
ERRMSG <list>	Given a list returned by ERROR, it prints the associated error message.
ERROR	Returns details of last error or the empty list.
EXAMINE <n>	Returns the value held in the byte address given by <n>.

Primitive	Effect
EXP <number>	Returns the exponential function of <number>.
EXPLORE <number>	Moves floor turtle forward by <number> steps. If an obstacle is encountered, returns the distance travelled.
FENCE	Fences the turtle within outline of screen.
FIRST <object>	Returns first element of <object>.
FLOOR	Applies subsequent commands to floor turtle.
FORGET <object>	Forgets turtle(s) given by <object>.
FORWARD(FD) <distance>	Moves turtle forward by a given <distance>.
FPUT <newobject> <oldobject>	Returns the object formed by putting <newobject> at the front of <oldobject>.
GO <name>	Transfers control to command following the label given by <name>.
GPROP <name> <pr>	Returns the property <pr> of <name>.
#HATCH <object> <shape>	Creates (hatches) turtle(s) specified by <object> at current turtle position and with a given shape. <object> is an integer or a list of integers.
+HEADING	Returns heading of turtle
HEX <word>	Returns decimal value of <word>.
HIBYTE <n>	Returns the most significant byte of the two byte value given by <n>.
HIDETURTLE(HT)	Makes turtle invisible.
HOME	Moves turtle to [0 0] and sets heading to 0.
HOOT	Sounds hooter on floor turtle.
IF <expression> <list>	If <expression> is TRUE, runs <list>. Returns a value, if <list> does.

Primitive	Effect
IF <expression> <list1> <list2>	If <expression> is TRUE, runs <list1>, otherwise, runs <list2>. Returns a value, if either list does.
IFFALSE <list>	Runs <list> if most recent TEST was FALSE.
IFTRUE <list>	Runs <list> if most recent TEST was TRUE.
INKEY <n>	If 0 <= <n> <= 3276, waits for <n> tenths of a second or until a key is pressed. Result is a null word if no key was pressed or a 1 character word if a key was pressed. If <n> > 3276, generates error. If <n> < 0, tests if specific key was pressed and returns TRUE or FALSE.
INT <number>	Returns integer part of number.
ITEM <n> <object>	Returns element <n> of <object>.
KEYQ	Returns TRUE if a key has been pressed but not used by RC, RL or RW, otherwise returns FALSE.
LAST <object>	Returns last element of <object>.
LEFT(LT) <degrees>	Turns turtle to left (anticlockwise) by the angle specified.
#LIST <object1> <object2>	Returns a list whose elements are <object1>, <object2>.
LISTQ <object>	Returns TRUE if <object> is a list.
LN <number>	Returns natural log of <number>.
LOAD <filename>	Loads contents of file into workspace.
LOBYTE <n>	Returns the least significant byte of the two byte value given by <n>.
LOCAL <name> <object>	Makes <name> local and makes its contents equal to <object>.
LOOP	Returns to the beginning of the REPEAT/ DOFOREVER list and increments the repeat count, if REPEAT.

Primitive	Effect
LPUT <newobject> <oldobject>	Returns object produced by putting <newobject> at end of <oldobject>.
MAKE <name> <object>	Makes <name> refer to <object>.
MEMBER <object1> <object2>	If <object1> is an element of <object2>, returns the element number, otherwise returns 0.
MEMBERQ <object1> <object2>	Returns TRUE if <object1> is a member of <object2>.
MODE	Returns the current display mode.
NOREDEF	Prevents anyone redefining primitives.
NOT <a>	Returns TRUE if <a> is FALSE or FALSE if <a> is TRUE.
NUMBERQ <object>	Returns TRUE if <object> is a number.
OSBYTE <A> <X> <Y>	Calls the operating system OSBYTE routine with register contents <A>, <X> and <Y>. Returns integer value formed by the contents of the <X> and <Y> registers. <X> and <Y> can be omitted.
OUTPUT(OP) <object>	Returns control to caller and returns <object> as result of a procedure.
PAL <col1> <col2>	Sets logical colour <col1> to physical colour <col2>.
PAUSE	Makes procedure pause.
+PC	Returns the current pen colour.
PE	Puts turtle's eraser down.
+PEN	Returns current pen parameters in the form of a list: pen state, visibility, colour, nib, pen type.
PENDOWN(PD)	Puts turtle's pen down.
PENRESET	Resets pen colour to 7, nib to 8 and pen type to 0. Puts pen down and shows turtle.
PENUP(PU)	Lifts turtle's pen.

Primitive	Effect
PENUPQ	Returns TRUE if pen is up.
PI	Returns the value pi.
PLIST <name>	Returns property list of <name>.
PM <n>	Logo will reserve space such that SETMODE will work with mode <n>, even if you are not using that mode at present.
PO <object>	Prints out definition of procedure(s) given by <object>.
POALL	Prints definition of every procedure and contents of every variable in workspace.
PONS	Prints name and value of every variable in workspace.
POPS	Prints definition of every procedure in workspace.
+POS	Returns turtle's position as list [x y].
POTS	Prints title line of every procedure in workspace.
PPALL	Prints properties of all words that have them.
PPROP <name> <pr> <object>	Gives the word <name> a specific property and associates the value <object> with it.
PPS <object>	Prints properties of the name(s) given by <object>.
PRIMITIVEQ <object>	Returns TRUE if <object> is a primitive.
#PRINT(PR) <object>	Prints <object> in text area and ends text with a RETURN. Successive objects are separated by spaces.
PRSCREEN	Copies contents of screen to printer.
#PRODUCT <number1> <number2>	Returns the product of <number1> and <number2>.
PX	Makes turtle pen perform an exclusive-or operation on the colour passed over.

Primitive	Effect
QUOTIENT <number1> <number2>	Returns integer part of <number1>/ <number2>.
RANDOM <n>	Returns a random, non-negative integer less than <n>.
RC	Returns character typed at keyboard, waiting if necessary. Character is not displayed.
READLIST(RL)	Returns a line from keyboard in the form of a list. Waits, if necessary.
READPICT <filename>	Reads the picture from <filename> on to the screen.
READWORD(RW)	Returns first word of a line from the keyboard.
REDEF	Permits the redefinition of primitives.
REDEFQ	Returns TRUE if primitives can be redefined.
REMAINDER <number1> <number2>	Returns remainder of <number1>/ <number2>.
REMPROP <name> <pr>	Removes property <pr> from <name>.
REPEAT <n> <list>	Runs <list> <n> times.
#RERANDOM <n>	Makes RANDOM behave in a repeatable way if <n> is specified. If <n> is omitted, then RANDOM becomes random again.
RIGHT(RT) <degrees>	Turns turtle to the right (clockwise) by a given angle.
ROUND <number>	Returns <number> rounded to the nearest integer.
RUN <list>	Executes <list> and returns whatever <list> does.
SAVE <filename> <object>	Writes all names and some or all procedures in workspace to <filename>. <object> can be omitted.

Primitive	Effect
SAVEPICT <filename>	Saves the current screen picture into <filename>.
SCR	Returns the aspect ratio of the screen.
SCREEN	Changes from floor turtle to screen turtle.
SCREENQ	Returns TRUE if screen turtle is in use.
SECT <radius> <angle> <width>	Draws a sector through a given <angle> with <radius> and <width>.
SENSE <n>	Returns a value if turtle sensor <n> is touching anything. Value depends on floor turtle used.
#SENTENCE <object1> <object2>	Returns a list formed by <object1> and <object2>. If either object is a list, SENTENCE takes the elements of that list, but not the list itself.
SETBG <n>	Changes logical background colour to <n>.
SETCURSOR <list>	Puts text cursor at the position given by <list>, which is in the form [<column> <line>].
SETDECS <n>	If <n> is in range 0 to 7, numbers will be rounded to <n> places. If it is 8, normal output will be restored.
SETDOT <list>	Puts a dot at the position given by <list> in current pen colour. <list> has the form [x y].
SETERR <list>	When called with the <list> returned by ERROR, it regenerates the corresponding error.
SETHEADING(SETH) <degrees>	Sets turtle heading to <degrees>.
SETITEM <n> <object1> <object2>	Returns object derived from <object1> with element <n> changed to <object2>.
SETMODE <n>	Selects display mode of computer.
SETNIB <n>	Selects graphics options of the BBC BASIC PLOT statement.

Primitive	Effect
SETPC <n>	Changes turtle's pen colour to <n>.
SETPEN <list>	Sets current pen parameters to <list> (in form returned by <PEN>).
SETPOS <list>	Moves screen turtle to the position given by <list>. <list> has the form [x y].
SETPT <n>	Defines use of colour on screen as in the BBC BASIC GCOL statement.
SETSCR <n>	Sets screen aspect ratio to <n>.
#SETSH <object>	Changes current turtle's shape to the value given by <object>.
SETX <number>	Moves turtle horizontally on the screen to the x-coordinate <number>.
SETY <number>	Moves turtle vertically on the screen to the y-coordinate <number>.
+SH	Returns current turtle's shape as a list.
SHOW <object>	Prints <object> followed by carriage return, with brackets for list.
SHOWTURTLE(ST)	Makes turtle visible.
SIN <degrees>	Returns sine of <degrees>.
SOUND <chan> <loud> <pitch> <dur>	Generates a sound of loudness <loud>, pitch <pitch> and duration <dur> on sound channel <chan>.
SQRT <number>	Returns square root of <number>.
STAMP	Leaves imprint of turtle's shape on screen.
STOP	Stops procedure and returns control to calling environment.
#SUM <number1> <number2>	Returns the sum of <number1> and <number2>.
TAN <degrees>	Returns the tangent of <degrees>.
TC	Prints details of current procedure calls.

Primitive	Effect
TELL <object>	Selects turtle(s) given by <object> and applies subsequent commands to it/them. <object> can be an integer or a list of integers.
TEST <a>	Notes if <a> is TRUE or FALSE.
TEXT <name>	Returns definition of procedure <name> as a list of lists.
THING <object>	Returns the contents of <object>.
THINGQ <object>	Returns the value TRUE if <object> has some value, otherwise returns FALSE.
THROW <name>	Transfers control to the corresponding CATCH.
TIDY	Performs garbage collection.
TIME	Returns time in tenths of a second since computer was switched on, CTRL BREAK was pressed or last TIMERESET was used.
TIMERESET	Resets time counter to zero.
#TITLE <object>	Prints <object> at turtle position in current pen colour. Does not output a RETURN. No spaces are left between successive objects.
#TO <name1> <name2> ... <namen>	Starts definition of procedure <name1>.
TOWARDS <list>	Returns heading turtle would have if it faced the position given by <list>. <list> has the form [x y].
TRACE <n>	Introduces or removes tracing and sets trace characteristics.
TS	Allots entire screen for text and may change mode.
TURTLES	Returns list of living turtles.
#TYPE <object>	Prints <object> in text area. Does not output a RETURN. No spaces are left between successive objects.

Primitive	Effect
UNBURY <object>	Unburies procedure(s) named by <object> from workspace.
UNBURYALL	Unburies all procedures from workspace.
#VDU <object>	Sends control codes to VDU driver. (VDU <n> ";) sends <n> as a two-byte value.
WAIT <n>	Waits for <n> tenths of a second.
WHO	Returns list of current turtles as selected by TELL.
WINDOW	Removes bounds from turtle field.
#WORD <word1> <word2>	Returns word made up of <word1> and <word2>.
WORDQ <object>	Returns TRUE if <object> is a word.
WRAP	Alters turtle field so turtle reappears at the opposite side when it reaches the edge of the screen.
WS	Returns list of total number of bytes available and maximum workspace for any individual item.
+XPOS	Returns x-coordinate of turtle's position.
+YPOS	Returns y-coordinate of turtle's position.

Appendix B

Logo error messages

PAR1 and PAR2 are the elements of the syntax (procedures, numbers, words, etc) which give rise to the error.

Error number	Error message
300	Unknown error
301	Logo has run out of space
302	Word is too long
303	Not enough inputs to PAR1
304	Too many local variables
305	Logo doesn't know how to PAR1
306	PAR1 has no value
307	Logo doesn't know what to do with PAR1
308	PAR1 is a primitive
309	PAR1 doesn't like PAR2 as input
310	PAR1 didn't output
311	Logo wants another ')'
312	Too much inside '()'s
313	Logo has found an extra ')'
314	Nothing inside '()'s
315	Too many inputs to PAR1
316	PAR1 is already defined
317	File PAR1 already exists
318	Logo needs to TEST before PAR1
319	PAR1 must be in a procedure
320	PAR1 is only allowed as a direct command
321	PAR1 is not allowed from within the editor
322	Logo can't load file PAR1
323	Logo can't find file PAR1
324	PAR1 can't find PAR2
325	Result of PAR1 is too big
326	Turtle is outside fence
327	Turtle hit fence

Error number	Error message
328	PAR1 is not at start of line
329	Too few items in PAR1
330	PAR1 is not within REPEAT or DOFOREVER
331	Turtle PAR1 is not alive
332	Turtle PAR1 is already alive
333	PAR1 is buried
334	PAR1 can't be used with screen turtle
335	PAR1 can't be used with floor turtle
336	Unknown error
337	Unknown error
338	Unknown error
339	Unknown error

Appendix C

ASCII code table

Action	Code (Decimal)		
Nothing	0	!	33
Next to printer	1	"	34
Start printer	2	#	35
Stop printer	3	$	36
Separate cursors	4	%	37
Join cursors	5	&	38
Enable VDU	6	'	39
Beep	7	(	40
Back	8	)	41
Forward	9	*	42
Down	10	+	43
Up	11	,	44
Clear text area	12	-	45
Carriage return	13	.	46
Paged mode on	14	/	47
Paged mode off	15	0	48
Clear graphics area	16	1	49
Define text colour	17	2	50
Define graphic colour	18	3	51
Define logical colour	19	4	52
Default logical colours	20	5	53
Erase line or Disable VDU	21	6	54
Select Mode	22	7	55
Reprogram characters	23	8	56
Define graphics area	24	9	57
Plot	25	:	58
Default screen areas	26	;	59
Nothing	27	<	60
Define text area	28	=	61
Define graphic origin	29	>	62
Move text cursor to 0,0	30	?	63
Move text cursor to X,Y	31	@	64
Space	32	A	65

B	66	a	97
C	67	b	98
D	68	c	99
E	69	d	100
F	70	e	101
G	71	f	102
H	72	g	103
I	73	h	104
J	74	i	105
K	75	j	106
L	76	k	107
M	77	l	108
N	78	m	109
O	79	n	110
P	80	o	111
Q	81	p	112
R	82	q	113
S	83	r	114
T	84	s	115
U	85	t	116
V	86	u	117
W	87	v	118
X	88	w	119
Y	89	x	120
Z	90	y	121
[	91	z	122
\	92	{	123
]	93	:	124
^	94	}	125
_	95	~	126
£	96	Backspace and delete	127

Index

Write cursor 101
WS 54,60

XPOS 35

YPOS 35

" 11
: 11
[] 12
\ 12
^ 12
() 12
+ 85
- 85
* 12,85
/ 86
> 86
< 86
= 87