

Introduction to Logo

**on the BBC Microcomputer
and Acorn Electron**

BARRY MORRELL

ACORNSOFT

Acknowledgements

The *Logo* software was designed by Richards Computer Products and was implemented by Chris Jobson and John Richards.

Many people helped in the preparation of this book. In particular, I should like to thank Richard Noss of AUCBE and Julian Pixton and Linda Spear of Walsall Logo project for their teaching advice. I also owe a lot to Steve and Linda for their help with the illustrations.

Thanks are also due to John Richards, Chris Jobson, Mike Fellows, John Laski and Jenny Raggett. Finally, I should like to thank Paul Fellows and Paul Christensen of Acornsoft, and Elaine Morrell.

Barry Morrell

ISBN 0 907876 95 1

Copyright © Acornsoft Limited 1984

All rights reserved

First published in 1984 by Acornsoft Limited

No part of this book may be reproduced by any means without the prior consent of the copyright holder. The only exceptions are as provided for by the Copyright (photocopying) Act or for the purposes of review or in order for the software herein to be entered into a computer for the sole use of the owner of the book.

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	1
1 Meet the turtle	3
What if I make a mistake?	5
Summary of chapter	7
2 Saving yourself some work	8
The REPEAT command	9
Spinning squares	10
Summary of chapter	11
3 The artistic turtle	12
The turtle's pen and eraser	12
Using different nibs	13
Fields and fences	14
Putting colour on the map	16
Summary of chapter	19
4 Teaching the turtle	20
Summary of chapter	23
5 Saving your procedures and pictures	24
Looking at procedures in your workspace	24
Saving your procedures	25
Saving your pictures	27
Summary of chapter	28
6 Learning more about procedures	30
Using one procedure within another	30
Using several procedures within another	32
Summary of chapter	33

7 What to do when your procedures don't work	34
Keeping your procedures tidy	34
Getting rid of the bugs	36
Summary of chapter	39
8 Using inputs with your procedures	40
Summary of chapter	42
9 Using numbers with Logo	43
Changing the order of calculation	44
Using decimals	44
Using negative numbers	45
Summary of chapter	45
10 Using boxes with Logo	47
Boxes with the same name	48
Summary of chapter	49
11 Recursion	50
The third wish	50
Stopping your spirals	51
Recursion without turtles	51
Summary of chapter	52
12 Multiple turtles	54
Random movements	56
Drawing multiple circles	57
Moving letters	58
Summary of chapter	59
13 Turtle navigation	60
Getting to the point	60
Find the turtle	61
Turtle Island	62
Summary of chapter	62

14 Playing with words	65
How does Logo handle words?	65
Getting words out of lists	67
Getting little lists out of big ones	67
Joining lists together	69
Words can be lists too!	69
Writing back-to-front	70
Summary of chapter	72
15 Writing interactive procedures	74
Reading characters from the keyboard	74
Improving your procedures	75
Summary of chapter	76
Appendix A	77
Editing your command lines	77
Appendix B	78
Editing your procedures	78
Appendix C	80
Use of colour	80
Appendix D	82
Logo primitives	82
Glossary	85
Further reading	87
Index	88

How to use this book



If you are not familiar with the computer language Logo, this book is for you. It teaches you the main ideas of Logo using examples that you can, and should, try out for yourself. If you are experienced in Logo, you might want to go straight to *Logo on the BBC Microcomputer and Acorn Electron*, the reference manual which is part of the Acornsoft Logo package. If you start by reading this introductory book and find that it is going too slowly for you, you will find the main points summarised at the end of each chapter.

Whichever type of reader you are, you will find a glossary at the back of this book to help you with unfamiliar terms.

Don't worry about damaging your computer. You can type anything you like without doing it any harm. If there is anything you don't understand in this book, try it out on your computer and talk it over with your friends.

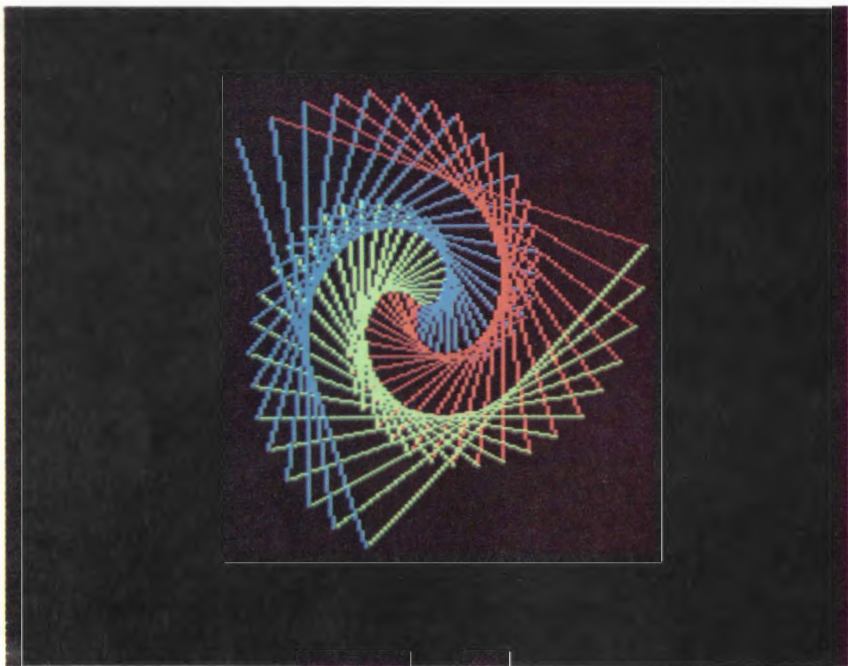


Figure 1

1 Meet the turtle



The Logo turtle is a small creature which lives in a 'field' on your screen and you can 'teach' it to do things without being bitten or trampled underfoot. Your turtle already understands a few things; for example, you can get it to move forward or back, or you can get it to turn.

Attached to the turtle is a 'pen' which can draw a 'trail' in different colours as the turtle moves. You can use the words which the turtle already understands, together with this pen, to explore the world of Logo and draw complicated and attractive patterns like those shown in figure 1. You can also teach it new words which extend its 'vocabulary' and help you make it draw even more interesting patterns in brilliant colours.

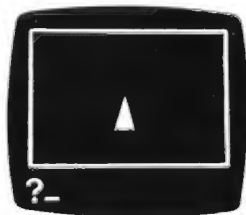
There are two kinds of turtle. The first kind are called 'floor turtles' because they crawl across the floor. The others are called 'screen turtles', because they appear on your television screen. Each kind can be controlled from the keyboard of your computer. You can use floor turtles with Acornsoft Logo, if you wish, but for the present we will stick to screen turtles. If you want to use a floor turtle, the Logo reference manual tells you how to do so.

Now, let's take a look at your turtle. To do this:

- Make sure that your computer is connected up and switched on, as described in the user guide for your computer.
- The message `Welcome to Logo` should have appeared on your screen. If it hasn't, type `*LOGO` then press the RETURN key.

Remember, whenever you want the computer to do what you have typed, you tell it to do so by pressing the RETURN key. Until you do this, it will do nothing except show what you have typed on the screen.

The screen should now look like this:



Your turtle is the triangle in the middle of the screen and its 'field' is the area surrounded by a line.

The ? in the bottom part of your screen is called a 'prompt'. It appears whenever the computer is waiting for you to type something on your keyboard. To the right of the prompt is a flashing underline symbol. This is called the 'cursor' and it shows where the next letter or number you type will appear on the screen.

When you have finished reading this paragraph, try typing the two lines printed below it and watch what happens. Check each line after you have typed it and, if you make a mistake, don't worry. Just use the DELETE key to remove the text before the cursor, then retype the correct version.

```
FORWARD 100  
LEFT 60
```



FORWARD 100



LEFT 60

FORWARD, LEFT and other words which the turtle understands are called 'commands'. The numbers after them are called 'inputs'. You can think of commands in the same way as you would the pieces of a Lego or Meccano set: they are 'bricks' with which you can build more complicated structures.

RIGHT and BACK are two other commands related to the two you have typed in; they also need inputs. Try finding out for yourself how RIGHT and BACK work and use them to put the turtle back where it started from (this is called its 'home position'). If you lose track of what you have done, you can get back to the starting point of this paragraph by typing:

```
DRAW  
FORWARD 100  
LEFT 60
```

You can use different numbers as inputs after FORWARD and LEFT if you like. Try using them with all kinds of numbers and see what happens.

Instead of typing the full names of commands you can use their short forms. FORWARD has the short form FD and LEFT has the short form LT. When we introduce a new command in the rest of this book we will put its short form (if it has one) after it, in brackets, for example: FORWARD (FD). Try to find out for

yourself what the short forms of **BACK** and **RIGHT** are. If you can't find them, look in the summary at the end of this chapter.

There are three other commands which will help you with your pictures: **DRAW**, **HOME** and **CLEAN**. **DRAW** and **HOME** both return the turtle to its home position, but they each have a different effect on the turtle's trail. Try typing them in and work out what the difference is (but first make sure that the turtle is away from its home position). Then try using **CLEAN** and find out what that does.



What if I make a mistake?

Everyone makes mistakes when they learn something new. In Logo, the most common mistake is missing out the space between a command and its input. For example:

```
FORWARD100
```

The next most common mistakes are typing errors (pressing the wrong key or pressing two keys at the same time). For example, you might type **FORWRAD 100** instead of **FORWARD 100**. If you notice the mistake before you press the **RETURN** key, you can correct it using the **DELETE** key as described earlier. If you do not, Logo will notice the mistake and reply with the message:

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

You can then retype the entire line.

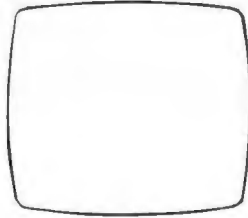
Whenever you type something wrong, or ask Logo to do something it does not understand, it will reply with a message which tells you clearly what it thinks is wrong. Try it.



Turtle fun

Here are a few ideas that you might like to try out.

1. Find out the height and width of the screen in steps. You could write your results on this diagram, if you like.



2. Try drawing the following pictures. If you are not sure how to do them, try 'playing turtle' and pace them out on the floor.



3. Try drawing the following picture using only the `RIGHT` and `BACK` commands.



Summary of chapter

1. To display the turtle, switch on your computer (if necessary, type *LOGO to get into Logo). The turtle is the triangle at the centre of the screen (its 'home position') and the ? symbol is a 'prompt'.

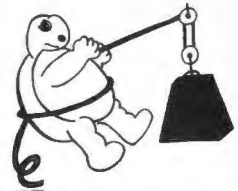
2. You can move the turtle using the commands FORWARD (FD), BACK (BK), LEFT (LT) and RIGHT (RT). For example:

```
FORWARD 100  
LT 60
```

The number on each line is called an 'input'.

3. HOME and DRAW return the turtle to its home position. DRAW clears the screen as well, but HOME does not. CLEAN clears the screen but leaves the turtle where it was.

2 Saving yourself some work



If you tried some of the activities in the last chapter you might have drawn a square using commands like these:

```
FORWARD 100  
LEFT 90  
FORWARD 100  
LEFT 90  
FORWARD 100  
LEFT 90  
FORWARD 100  
LEFT 90
```



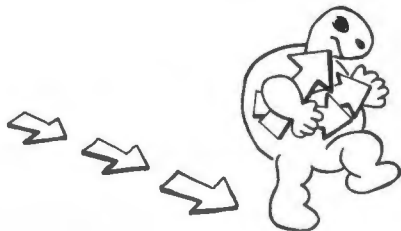
If you didn't try these activities, clear the screen using `DRAW` and try them out now.

So far, you have typed a single command on each line but you don't have to do this. If you want, you can put a number of commands on each line to make them easier to read:

```
FORWARD 100 LEFT 90  
FORWARD 100 LEFT 90  
FORWARD 100 LEFT 90  
FORWARD 100 LEFT 90
```

Clear your screen, then try this out and see for yourself. Try using other commands in this way as well. In particular, try typing them all on the same line and watch what happens when you reach the end of the line. See how many you can get on the same line.

You might have noticed that the last few lines you typed are displayed at the bottom of the screen to remind you of what you have done. If you type two or three commands on each line, these lines will show you much more than if you had only typed one command on each line.



The REPEAT command

Suppose someone said to you: 'I want you to lay a treasure trail for someone to follow. Walk forward ten paces then put down a marker'. You would probably not mind doing this. But if they had to say it to you ten times, one after the other, you would both be a little irritated. They are telling you to do the same things over and over again and you are not stupid. Why couldn't they just say: 'I would like you to repeat a number of actions ten times. The actions are as follows:

- Walk forward ten paces.
- Put down a marker.

Now go ahead and do them, please'.

In Logo you can be in the same situation. A lot of the things you type in are groups of the same commands repeated over and over again. When you drew the square opposite, for example, you typed the following commands four times:

```
FORWARD 100 LEFT 90
```

Instead, you could have typed:

```
REPEAT 4 [FORWARD 100 LEFT 90]
```

This just means: 'Take the list of commands inside the square brackets and repeat it four times'. Clear your screen, then try it out and see.

If you mistype the number of repeats and end up with something like:

```
REPEAT 4000 [FORWARD 100 LEFT 90]
```

Logo will keep drawing squares for a long time when you press RETURN. If you want to stop Logo drawing, you can always do this by holding down the CTRL key, then pressing ESCAPE. Try the above example and see for yourself, then remember what happened! You will find this combination of keys a useful tool whenever you need to stop Logo doing something.

You could use REPEAT to draw a triangle. Try working out for yourself how to do this. You might like to see what happens if you put other commands in the square brackets.

Here are two other examples to try. Type them in and watch what happens.

```
REPEAT 6 [FD 200 BK 200 LT 60]
```

```
REPEAT 12 [FD 200 BK 200 LT 30]
```

Can you make a picture with 24 rays?

Next, try the following:

```
REPEAT 24 [FD 200 LT 60 FD 100 BK 100 RT 60 BK 200 LT 15]
```

```
REPEAT 6 [FD 100 LT 30 FD 50 RT 90]
```

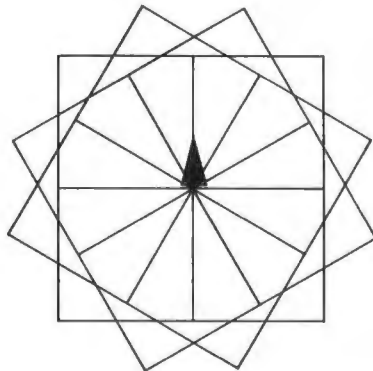
Can you draw some different pictures using REPEAT?

Spinning squares

You can make some very attractive pictures by drawing squares and turning each square relative to the one drawn before it. Try typing this:

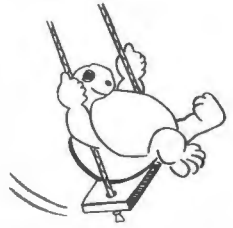
```
REPEAT 12 [LEFT 30 REPEAT 4 [FORWARD 200 LEFT 90]]
```

It draws a number of squares like the following:



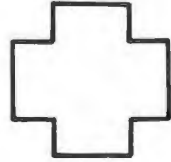
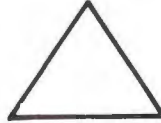
It might surprise you that REPEAT can be used inside the square brackets of another REPEAT command like this. You can, in fact, use most commands in this way.

Actually, there is a much neater way of drawing this picture, but you'll have to wait a little while before you find out how!



Turtle fun

1. Try using REPEAT to draw the following pictures. (Hint: work out first which part is being repeated.)



2. The following command draws six 'spinning squares':

```
REPEAT 6 [LEFT 60 REPEAT 4 [FORWARD 200 LEFT 90]]
```

Try drawing some more using a different number of squares. (Hint: look at the relationship between the number of repeats and the angle of spin.)

3. Try some different commands inside the brackets (for example RT) and see what sort of pictures you can make.

Summary of chapter

1. You can put a number of commands on one line. For example, the following draws a square:

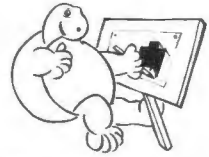
```
FORWARD 100 LEFT 90  
FORWARD 100 LEFT 90  
FORWARD 100 LEFT 90  
FORWARD 100 LEFT 90
```

2. You can repeat a number of commands several times using the REPEAT command. For example, the following will draw the same square:

```
REPEAT 4 [FORWARD 100 LEFT 90]
```

Here the commands inside the square brackets are repeated four times.

3 The artistic turtle



The turtle has a 'pen' underneath it, as you have already discovered. It also has an 'eraser'. In addition, the pen can have a number of 'nibs', allowing it, for example, to draw normal lines or dotted lines.

If you have a colour screen you can make the turtle leave trails in different colours and change the 'background' colour of your screen (this is set to black when you start). If you have a black and white screen you cannot get the full benefit of these facilities, but try them anyway. You will get pen and background colours in different shades of grey, depending upon which colour you selected.

Now let's see if you can turn yourself into an artist.

The turtle's pen and eraser

You control the turtle's pen with the `PENUP(PU)` and `PENDOWN(PD)` commands. These do exactly what they say. Try them out for yourself and see.

When you have finished exploring, try typing in the following commands and watch what happens:

```
REPEAT 9 [FD 50 LT 20 PENUP FD 50 LT 20 PENDOWN]
```

This draws an 18-sided figure, but the pen is lifted for every other side. If you look again at the instructions to the turtle you can work out what is happening at each stage.

Now, before you do anything else, type the following and watch what happens:

```
PE REPEAT 9 [FD 50 LT 20 FD 50 LT 20]
```

The first command on the line, `PE`, is short for Pen Erase and puts the turtle's eraser down. The rest of the commands retrace the turtle's previous path and erase the lines drawn. If you try to move the turtle now it will leave no trail as the eraser is still down. To lift it you must replace it with the pen by typing:

```
PENDOWN
```

Try using these three commands yourself. See if you can make the turtle's trail look like a dotted line and draw some figures with it. Try drawing a square and a triangle, for example.

Using different nibs

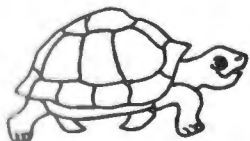
With some pens you can get nibs which produce lines of different thicknesses. Logo's SETNIB command can draw dotted lines and other nice effects. Try the following:

```
DRAW  
SETNIB 80  
REPEAT 4 [FD 200 LT 90]
```



```
SETNIB 80 REPEAT 4 [FD 200 LT 90]
```

The number after the SETNIB command tells Logo the type of nib you want. This is the 'value' of the nib, each value giving another type of nib.



You can hide the turtle shape by typing HIDE TURTLE(HT) and bring it back by typing SHOW TURTLE(ST). This can improve your picture – try it out and see!

Now try this:

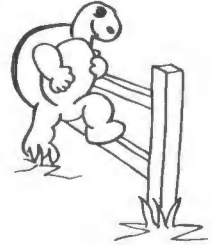
```
DRAW  
SETNIB 16  
HIDE TURTLE  
REPEAT 4 [FD 200 LT 90]
```



```
SETNIB 16 REPEAT 4 [FD 200 LT 90]
```

This is much easier than using `PENUP` and `PENDOWN`, which you may have tried earlier, but at least you learned how to use them!

The turtle's nib normally has the value 8 and is reset to this value (called its 'default') whenever you type `DRAW`. If you want to explore the other nib types available, you can find descriptions of them under the command `SETNIB` in the Logo reference manual. Or you could just try typing in some `SETNIB` commands and see what happens. Some numbers will give the same effect. Can you work out which numbers change the effect? Try numbers up to 80.



Fields and fences

Type the following and watch what happens:

```
DRAW  
RIGHT 35 [REPEAT 10 [FORWARD 190]]
```

Your screen display should now look like this:



Whenever the turtle goes off the screen it will reappear at the opposite side. This type of screen display is called 'wrap mode' to distinguish it from other modes which you will come across later. See if you can get the lines going in a different direction to make a 'check' pattern.

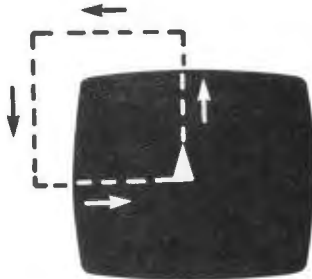
Now try typing in the following commands and see what happens. Don't worry if you lose sight of the turtle after the second command. Just keep on typing.

```
DRAW
WINDOW
FORWARD 1000
RIGHT 180
FORWARD 1000
RIGHT 180
```

Your turtle should have moved off the screen, turned round and come back to the home position, then turned round again. Just because it is off the screen doesn't mean it can't move! What you can see is a small part of the turtle's field, just as though you are looking through a small window. There is a lot of space beyond this. Clear the screen and try it again, if you like.

The type of display you have now is called 'window mode'. You get into it by typing `WINDOW` and you get back to wrap mode by typing `WRAP`.

When you feel that you understand what is happening, try this: see if you can get the turtle to move forward off the screen then go round the outside of the screen, coming back to the home position from the left-hand side. The idea is shown in the diagram below:



If you want to stop the turtle moving off the screen, you can do so using the `FENCE` command. This draws a thick line as a 'fence' around the edge of the screen. If the turtle hits this fence, the following message appears:

```
Turtle hit fence
```



Try it out by typing:

```
DRAW
FENCE
FORWARD 600
BACK 400
```

You can remove the fence and restore the wrap on your territory again by using the **WRAP** command. Try it now by typing:

```
WRAP
CLEAN
FORWARD 1000
RIGHT 180
FORWARD 1000
RIGHT 180
```



Putting colour on the map

The background colour of your screen at the moment is black, and the pen colour white, but you can change these if you want to. The colours you can use depend upon the 'screen mode' and this, in turn, depends upon the equipment you have. Screen modes are described in detail in Appendix C, 'Use of colour'.

When Logo is first loaded into your computer it always uses screen mode 4. You can change this by typing **SETMODE** followed by a number corresponding to the mode you want. Try typing the following, for example, and watch what happens:

```
SETMODE 5
SETBG 2
SETPC 1
REPEAT 24 [LT 15 REPEAT 4 [FD 200 LT 90]]
```

You should get a picture like the one in figure 2. The **SETBG** (**SET BackGround** colour) and **SETPC** (**SET Pen Colour**) commands define the colour of the screen and pen trace, respectively. The inputs you should give them are defined in Appendix C for each screen mode. Try a few different ones and see what happens. After you have done this, try combining them with the **SETNIB** command to see what other effects you can come up with. For example, try this:

```
SETMODE 5
SETBG 2
SETPC 1
SETNIB 80
REPEAT 24 [LT 15 REPEAT 4 [FD 200 LT 90]]
```

Then try this:

```
SETMODE 5
SETBG 2
```

```
SETPC 1
SETNIB 80
PENUP FD 200 PENDOWN
REPEAT 24 [LT 15 REPEAT 4 [FD 200 LT 90]]
```

You could also use the SETPT (SET Pen Type) command, described in the Logo reference manual. This defines how colour is to be used. Don't worry if you don't understand it at first. Have a go with a few different numbers and see what happens. This example will get you started:

```
HOME
CLEAN
SETNIB 80
SETPT 3
REPEAT 4 [FD 200 LT 90]
```

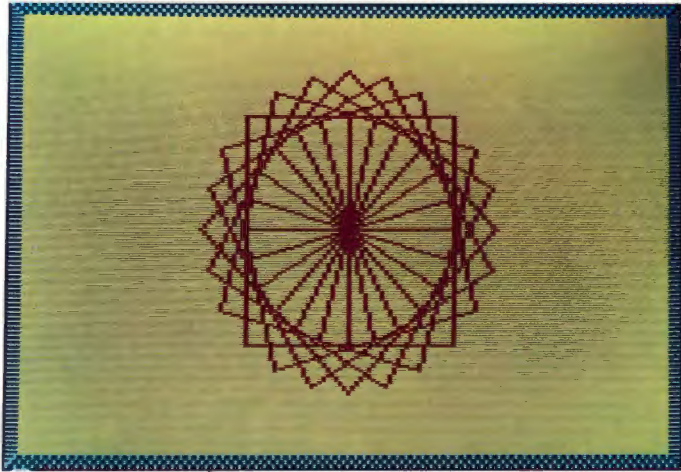


Figure 2

4. Now try to get the turtle to go up off the screen, then right around the outside and back from the top.

Summary of chapter

1. You can stop the turtle leaving a trail by lifting its 'pen'. You do this using `PENUP (PU)` and put it back down using `PENDOWN (PD)`.

2. You can remove lines using the turtle's 'eraser'. This is brought into action using `PE (Pen Erase)` and the pen is reset using `PENDOWN`.

3. The turtle's 'pen' can have different 'nibs' which allow you to draw, for example, normal lines or dotted lines. The nib is changed using the `SETNIB` command and reset to its normal state or value (8) by `DRAW`.

4. When you first use Logo and start using the turtle your screen is in 'wrap mode'. If the turtle disappears off one side of the screen it will reappear on the other side. To get back into wrap mode, type `WRAP`.

5. If you type `WINDOW` the area you see on your screen is just part of the turtle's 'field'. If you move the turtle off the screen you can still give it commands in this mode (called 'window mode', because you are effectively looking through a 'window' at it).

6. You can put a 'fence' around the screen to stop the turtle wandering off it by using the `FENCE` command. If the turtle attempts to cross this fence it will stop at the fence and display the message:

```
Turtle hit fence
```

7. You can set the pen and background colours using the `SETPC (SET Pen Colour)` and `SETBG (SET BackGround colour)` commands respectively. The colours depend upon the screen mode, and this is set using the `SETMODE` command. For example, the following commands set the pen colour to red and the background colour to yellow in screen mode 5:

```
SETMODE 5  
SETPC 1  
SETBG 2
```

The pen and background colours, together with the screen modes, are described in detail in Appendix C, 'Use of colour'.

4 Teaching the turtle



Imagine that you are stranded on your own in the north of Canada. You have a lot of food but you cannot cook, so you start by heating a few cans of baked beans.

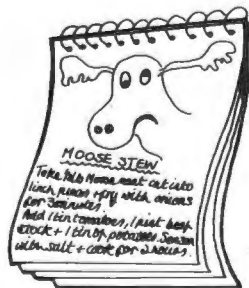
You can live off baked beans for a long time, but after a few days of having the same meals every day you would probably get bored. However, you have a wide variety of food and, on looking further, you find a recipe book. Your first attempt at making an omelette works and soon you are building an oven to help you cook more exciting meals.

As there are plenty of animals in this part of Canada you start hunting to get some fresh meat for your dishes. You shoot a moose and then run up against a problem: your recipe book doesn't tell you how to cook moose meat.

You now have to improvise and try cooking it a number of ways, based upon methods you have already used for tinned beef or lamb. Eventually you start to write your own recipes based upon your successful efforts. And soon after, you aren't on your own any more. Everyone from miles around has noticed the cooking smells and you make a fortune selling your moose stews to them!

The recipe for moose stew is a set of instructions, or a 'procedure', that tells you, or someone else, how to do a certain task. You write it once and it can be used over and over again by anyone who can read it.

You can write procedures in Logo to cook up the effects you want on the screen. They may not be as tasty as moose stew, but they can still make life easier for you.



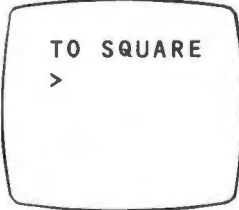
As a simple example, we can turn the commands you used to draw a square into a procedure by adding the two commands underlined below (but don't type them yet):

```
TO SQUARE  
FORWARD 100  
LEFT 90  
FORWARD 100  
LEFT 90
```

```
FORWARD 100
LEFT 90
FORWARD 100
LEFT 90
END
```

The first of these lines, `TO SQUARE`, is the 'title line'. `TO` tells Logo that you want to *define* a procedure and you will call it by the name that follows (in this case `SQUARE`). The last line of the procedure, `END`, tells Logo that this is the end of the procedure. In between these two lines is the 'body' of the procedure.

When you type in the commands they are not executed immediately, as they were before. Instead, they are held in the computer's memory for later use. When you type the first line, the `?` prompt disappears and you now have one that looks like `>`.



```
TO SQUARE
>
```

The `?` means 'Tell me to do something'; the `>` means 'Teach me how to do something.'

When you type the `END` line, the screen display returns to the way it was before and the prompt changes back to `?`

Now type the procedure in, as shown above. It is a good idea to check each line for mistakes before you press `RETURN`; if you make any mistakes use the `DELETE` key to correct them. If you want to erase the entire procedure and start again, you can do so by typing the following (including the `"`):

```
ERASE "SQUARE
```

The `ERASE` command merely tells Logo to forget about a procedure you have typed in.

When you have typed in your procedure you will get the message `SQUARE defined` printed out. You can now run it by typing:

```
SQUARE
```

Try doing this and see what happens. The turtle should have drawn a square, as it did before. Next, type:

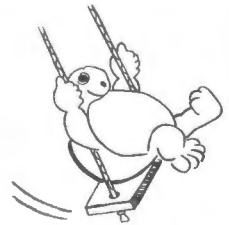
**DRAW
SQUARE**

This should clear the screen and draw the square again.

Now try writing a procedure similar to **SQUARE**, but with a few differences. What would happen if you used a different input from **FORWARD**? Think about it, try it out and watch what happens. Don't forget to give your new procedure a different name; if you try to use a name you have already used, Logo will complain.

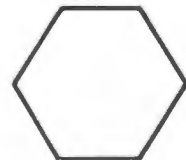
When you have done this, try putting your new procedure within the **REPEAT** brackets and see what happens. Then try 'doodling' a little by putting a number of different commands inside a procedure.

Procedures are just another type of command with one difference: they are commands which you define yourself. The other group of commands, things like **FORWARD**, **LEFT** and **DRAW**, are called 'primitives' because they are the basic things you can use to build more complex commands. From now on, we will use the word 'procedure' to refer to a command you have defined yourself, and the word 'primitive' to refer to Logo's built-in commands.



1. Think of the things you have done up to now which have produced attractive pictures. Now turn them into procedures so that you can run them at any time. You could start with the examples in chapter 1, 'Meet the turtle', and then try the ideas you have had yourself.

2. Try to define procedures to draw the following figures:



Summary of chapter

You can define 'procedures' which save you typing in the same group of commands at different times. This is done by adding two lines to the group of commands, as shown below:

```
TO SQUARE  
FORWARD 100  
LEFT 90  
FORWARD 100  
LEFT 90  
FORWARD 100  
LEFT 90  
FORWARD 100  
LEFT 90  
END
```

The first line is the 'title line'; it tells Logo that you are defining a procedure and gives the name by which it will be known (in this case, **SQUARE**). The last line tells Logo that it has reached the end of the procedure.

When you type the title line, Logo allows you to type in each line of the procedure without it being executed. When you type the **END** line, Logo returns to its normal state. You call the procedure by typing:

```
SQUARE
```

Procedures are commands which you define for yourself. 'Primitives' are the built-in commands which Logo gives you to start with, so that you can build up your own procedures.

ERASE tells Logo to forget about a procedure you have typed in, for example:

```
ERASE "SQUARE
```

5 Saving your procedures and pictures



When you define a procedure it is stored in your computer's immediate memory; this memory is called its 'workspace'. If you want to look at the procedures in your workspace, you can do so using a number of primitives described in this chapter.

Unfortunately, the workspace cannot hold your procedures indefinitely: when you switch the computer off, the contents of your workspace are lost. If you want to keep a set of procedures after this point you must save them on disc or cassette tape. Then, when you use Logo again, you can load them back into your workspace.

You can also save the pictures you have drawn on to disc or tape. So if you have built up an interesting screen, you don't have to remember how you did it.

Looking at procedures in your workspace

First of all, you need to be able to display more than six lines of text on your screen, otherwise you may not be able to see a complete procedure. You can ask Logo to use the entire screen for text using the **TS** (Text Screen) primitive. All you need do is type:

```
TS
```

and you should do this now. **DRAW** takes you back to the turtle graphics screen.

You can print the title lines of all your procedures using the primitive **POTS** (Print Out Title lineS). Try it now, by typing:

```
POTS
```

To look at the procedures themselves, you use the **POPS** (Print Out ProcedureS) primitive. Try this one, as well, by typing:

```
POPS
```



If you just want to look at one procedure, the **PO** (PrintOut) primitive will do this for you. To look at the procedure **SQUARE**, you should type:

```
PO "SQUARE
```

Type this in and see what happens. Then try looking at some of your other procedures.

Don't forget, when you want to display the turtle graphics screen again you can do so by typing `DRAW`.

Saving your procedures

You can save the procedures in your workspace with the `SAVE` primitive and read them back with the `LOAD` primitive. The instructions for using these are given below. Where you see the symbol , the instructions that follow refer to cassette tape. Where you see the symbol , the instructions refer to floppy discs.

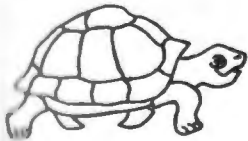


First, wind the tape you are going to use to a free space.

Next, you should type:

```
SAVE "MYWORK
```

and press the `RECORD` button. All of the procedures you have created in your workspace will now be stored on the tape in a 'file' called `MYWORK`. You could, of course, have used any name instead of `MYWORK`, as long as it consisted of no more than ten characters.



Notice that the filename must have quotes before it. This is because filenames are a special type of Logo object called 'words'. These are described in chapter 14, 'Playing with words'.

You will, of course, want to load the procedures you have saved back into workspace at some time. To do this, you use the `LOAD` primitive.

First of all we'll get rid of all the procedures in your workspace. You can do this using the `ERPS` (`ERase all ProceduresS`) primitive:

```
ERPS
```

Rewind your tape to the beginning of the file you want to load. Then type:

```
LOAD "MYWORK
```

You should now start your tape recorder running. Your procedures will then be loaded back into your workspace, replacing its current contents.

You will now be able to save any procedures you have written and retrieve them again at any time. It is a good idea to save your workspace periodically, when working on long jobs. This will prevent you losing everything if, for example, you have a power failure.



Ensure that the disc doesn't have a write protect label on it.

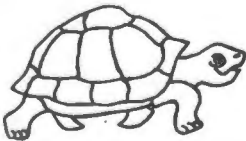
Next, you should type:

```
SAVE "MYWORK
```

All of the procedures you have created in your workspace will now be stored on the disc in a 'file' called **MYWORK**. You could, of course, have used another name instead of **MYWORK**, and you need to think of a new name each time you want to save new work:

Logo will allow you to have filenames of any length. However, the filing system you have may itself impose a limit on the filenames used. For example, the BBC Microcomputer Disc Filing System allows filenames of up to seven characters only.

Notice that the filename must have quotes before it. This is because filenames are a special type of Logo object called 'words'. These are described in chapter 14, 'Playing with words'.



You will, of course, want to load the procedures you have saved back into workspace at some time. To do this, you use the **LOAD** primitive.

First of all we'll get rid of all the procedures in your workspace. You can do this using the **ERPS** (**ER**ase all **P**rocedure**S**) primitive:

```
ERPS
```

Now all you need to do is type:

```
LOAD "MYWORK
```

Your procedures will then be read back into your workspace, replacing its current contents.

You should now be able to save any procedures you have written and retrieve them again at any time. It is a good idea to save your workspace periodically, when working on long jobs. This will prevent you losing everything if, for example, you have a power failure.

Checking your files

You can check that you've saved a file by using the `CAT` primitive. If you are using discs, all you need do is type:

```
CAT
```

If you are using cassette tape, rewind the tape, type `CAT` then press `RETURN`. If you then press `PLAY`, this displays the names of all your files and `MYWORK` should be included. At the end of the tape you should press `ESCAPE` to regain control.

If you can't remember the name of a file that you want to load, remember that you can check it by typing:

```
CAT
```

Erasing your files

If you want to erase a file on disc you can do so by using the `ERFILE` primitive. For example, to erase the file `MYWORK` you should type the following:

```
ERFILE "MYWORK
```

Don't forget the quotes!

Saving your pictures

As well as saving and reading procedures, Logo allows you to save and read pictures. You do this using the `SAVEPICT` and `READPICT` primitives.

Note that you must have a picture to save. If you try to use `SAVEPICT` when `TS` has been used, nothing will be saved.



First wind the tape you are going to use to a free space.

Next, type:

```
SAVEPICT "SPINFIL
```

This saves your picture into a file named `SPINFIL`. You could, of course, have used a different filename.

When you have done this, clear your screen and rewind the tape back to the beginning of the file. Now try to read the picture back on to the screen again by typing:

```
READPICT "SPINFIL
```

and then starting your tape recorder.



Make sure that your disc doesn't have a write-protect label upon it. Next, type:

```
SAVEPICT "SPINFIL
```

This saves your picture into a file named `SPINFIL`. You could, of course, have used a different filename.

When you have done this, clear your screen and try to read the picture back on to the screen again by typing:

```
READPICT "SPINFIL
```

Summary of chapter

The computer's immediate memory is called its 'workspace'. When you switch the computer off, everything in the workspace is lost. If you want to save procedures, variables and pictures in your workspace you can do so as follows:

1. The `SAVE` primitive saves the contents of your workspace on to disc or cassette tape. The following example saves it into a file called `MYWORK`:

```
SAVE "MYWORK
```

2. The `LOAD` primitive loads the contents of a file into your workspace. The following example loads the contents of the file `MYWORK`:

```
LOAD "MYWORK
```

3. The `SAVEPICT` primitive saves the picture on your screen on to disc or cassette tape. The following example saves the screen into the file `SPINFIL`:

```
SAVEPICT "SPINFIL
```

4. The `READPICT` primitive displays the picture in a file. The following example loads the picture contained in the file `SPINFIL`:

```
READPICT "SPINFIL
```

You can, of course, change the filename in each of the command lines shown above.

If you want to examine the files on your disc or tape you can do so using the **CAT** primitive. If you are using disc, you just need to type:

CAT

If you are using tape, you should first rewind it, then type **CAT** and finally press **PLAY**. To regain control at the end of the tape you must press **ESCAPE**.

If you want to erase a file you can do so using the **ERFILE** primitive. For example, the following erases the file **MYFILE**:

ERFILE "MYFILE

You can examine the procedures in your workspace using a number of Logo primitives:

1. You can print the title lines of your procedures with the primitive **POTS** (Print Out Title LineS).
2. You can examine your procedures with the primitive **POPS** (Print Out ProcedureS).
3. A procedure itself can be listed using the primitive **PO** (Print Out). The following example lists the procedure **SQUARE**:

PO "SQUARE

You can use the entire screen to display text by using the **TS** (Text Screen) primitive.

6 Learning more about procedures



Using one procedure within another

If you have saved your procedure `SQUARE`, load it now. If you haven't, redefine it.

Next, start off by typing the following commands and watch what happens:

```
TO SPIN
REPEAT 6 [SQUARE LEFT 60]
END

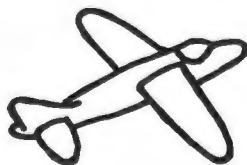
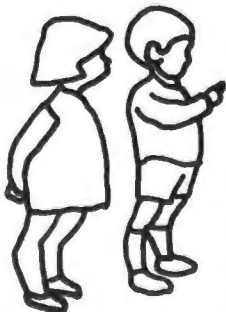
SPIN
```

You should recognise the picture from chapter 2, 'Saving yourself some work'. The only difference is that here the squares are drawn by the procedure `SQUARE`, and this is called by `SPIN`. We have a situation where one procedure is calling another one.

In case this puzzles you, let's explore it a bit more. There are two things to think about:

- Why should you want to have one procedure calling another one?
- How does it work?

The first problem can be illustrated by an example from outside the world of computers.



If you were making a model aeroplane from a kit you would probably use a plan. This tells you the types and quantities of materials you need and how to break the job up into a number of smaller jobs, or 'modules'. You would not start at the nose then work round one wing to the tail, and finally come back round the other wing. If you built it this way it would probably never fly! Instead, you would probably break the job up into:

1. The fuselage, or body.



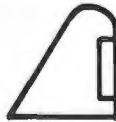
2. The wings.



3. The tailplane.



4. The tail fin.



There are so many problems involved in doing each part separately that by doing it this way you will prevent a problem that occurs, for example, in the fuselage, from affecting the wings or other parts.

You can use the same method in many other areas of life, but especially with computers. In the SPIN example above, you have a lot of similar squares 'spinning' around one point. If you were designing the picture from the beginning you might notice this and think: 'That square is a separate job in itself. It could be put into a separate procedure and made to work. When I've done this I'll be well on the way to solving the larger job, which itself is a procedure.'

Now we will look at the other problem: how can you use one procedure inside another?

This is probably the easier one to solve: in chapter 4, 'Teaching the turtle', we saw how the Logo commands you had been using (for example, FORWARD, LEFT and DRAW) could be used within a procedure. But procedures themselves are another type of command, so they, also, can be used within a procedure.

Using several procedures within another

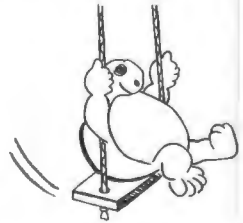
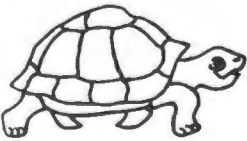
In the *SPIN* example we only have one procedure within another, but there is no reason why you cannot have one procedure calling several others. Look at the following picture, for example:



This has two shapes you have already drawn, at various times. Try drawing the picture using a procedure `HOUSE` which calls two other procedures, `SQUARE` and `TRIANGLE`.

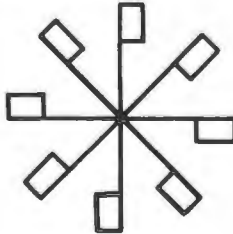
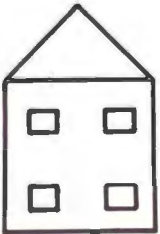
If you run into problems and you need to change a procedure, you can do this in two ways.

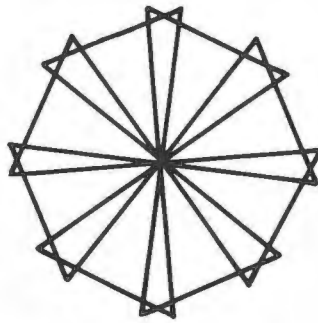
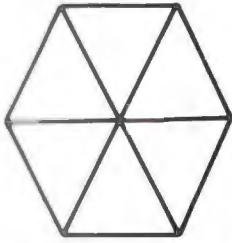
One way is to erase the procedure using the `ERASE` primitive and then type it in again. The other way is to use the Logo editor to change the procedure. This is described in Appendix B, 'Editing your procedures', and its use is covered in the next chapter.



Turtle fun

Try drawing the following pictures by breaking them up into smaller problems and solving these separately:





Summary of chapter

You can use procedures within other procedures. For example:

```
TO SPIN  
REPEAT 6 [SQUARE LEFT 60]  
END
```

If you want to change a procedure you can do this in two ways:

1. You can erase the procedure using the ERASE primitive, then type in the correct version.
2. You can use the Logo editor to change the procedure. This is described in Appendix B, 'Editing your procedures', and its use is covered in the next chapter.

7 What to do when your procedures don't work



We are now going to introduce two new words: 'bugs' and 'debugging'. Bugs are faults in procedures. The process of removing them is called debugging.

The most common bugs are caused by mistyping. Fortunately, a lot of these can be weeded out. For example, if you type in the primitive:

```
FORWARD 100@
```

Logo will warn you with the message:

```
FORWARD doesn't like 100@ as input
```

Messages like this which Logo uses to tell you what is wrong are called 'error messages'. Everyone gets error messages like this at some time!

If you type this mistake within a procedure, the error will not be spotted until you run the procedure. To see this type the following, then run the procedure and watch what happens:

```
TO BADSQUARE  
REPEAT 4 [FORWARD 100@ LEFT 90]  
END
```

However, if you type `FORWARD 1000` when you really mean `FORWARD 100`, you will get no warning until your procedure starts doing unexpected things. This might not happen until much further into the procedure than the faulty line. If you have two or more bugs like this in the same procedure you can end up with very strange results.

Keeping your procedures tidy

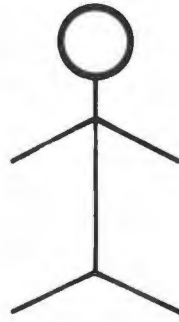
The most effective way of removing bugs is to trap them before you run the procedure. For example, look at the following primitives; they produce the picture shown on their right.

```
RIGHT 60  
BACK 100  
FORWARD 100  
LEFT 120  
BACK 100
```

```

FORWARD 100
RIGHT 60
FORWARD 200
RIGHT 60
BACK 100
FORWARD 100
LEFT 120
BACK 100
FORWARD 100
RIGHT 60
FORWARD 50
RIGHT 90
REPEAT 36 [FORWARD 10 LEFT 10]
LEFT 90

```



This sequence of primitives works, but you can't see what it is doing without spending a lot of time thinking about it. The primitives may do something particularly clever, but this won't be obvious to anyone trying to follow them. As well as this, if they took the form of a long procedure, this could well have a lot of bugs hiding in it.

How could you improve it? Well, you could split the problem into a number of smaller procedures. Each of these is unlikely to contain more than one bug and can be tested on its own. So, for the above example, you could make your procedures look like this:

```

TO MAN
VEE
FORWARD 200
VEE
FORWARD 50
HEAD
END

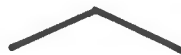
```

It is much clearer what this is meant to do. The procedure will also be easy to debug when you have written it since you only need to debug VEE, HEAD and the main procedure, MAN. The first two might look like this:

```

TO VEE
RIGHT 60
LINE
LEFT 120
LINE
RIGHT 60
END

```



where **LINE** is defined by:

```
TO LINE  
FORWARD 100  
BACK 100  
END
```

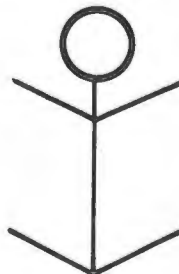
and

```
TO HEAD  
RIGHT 90  
REPEAT 36 [FORWARD 10 LEFT 10]  
LEFT 90  
END
```



Getting rid of the bugs

If you type in the above procedures and run them (by typing **MAN**) you will get the picture shown below. This is because there are bugs somewhere in the procedures.



You have two primitives to help you with debugging: **PAUSE** and **CONTINUE** (**CO**). If you put a **PAUSE** primitive in your procedure, the procedure will stop when it reaches that point. You can then look at what has happened so far and, if you're happy with it, make it run on again from the same point by typing **CONTINUE**.

Our **MAN** seems to be going wrong during the calls to **VEE**: both the legs and arms are being drawn upside down. To find out what is going wrong, we need to put two **PAUSE** primitives into **VEE** as shown below:

```
TO VEE  
RIGHT 60  
LINE
```

```
PAUSE
LEFT 120
LINE
PAUSE
RIGHT 60
END
```

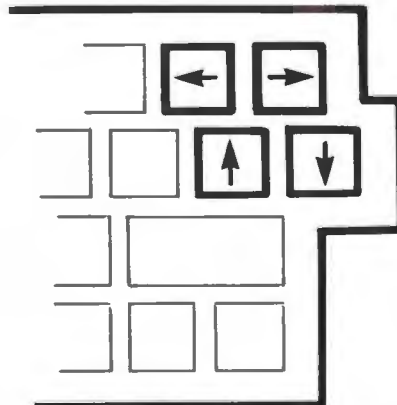
To change VEE like this we need to use the Logo editor, for which the primitive is EDIT (ED). First of all type:

```
EDIT "VEE
```

VEE will be shown like this:



You can move the cursor about the procedure using the cursor keys:



Try doing this, without typing anything else. Then, when you are ready, move the cursor to the end of the first line containing `LINE` and press `RETURN`. The text will move down, leaving a gap where you can type the new primitive, `PAUSE`. Now move the cursor down to the end of the second line containing `LINE` and do the same again. Finally, to leave the editor, press the `COPY` key.

Now try running `VEE` on its own and watch what happens:

```
VEE
Pausing, in VEE
PAUSE
VEE?
```



```
CONTINUE
Pausing, in VEE
PAUSE
VEE? CONTINUE
```



The picture went wrong right from the start and there are only two primitives before the `PAUSE` that could have caused this: either `RIGHT 60` or `LINE`. We didn't try to debug `LINE` first, because it contains only two simple primitives. So it is likely that the error is in `LINE`. In fact, if you look closer at `LINE`, the order of the primitives is reversed. `LINE` should look like this:

```
TO LINE
BACK 100
FORWARD 100
END
```

Try changing them using the editor. You can delete a line by moving the cursor to the end of the line then pressing the `DELETE` key until the text disappears and the lines close up. As well as this, remove the `PAUSE` primitives from `VEE`. If you try running `VEE` again you should get the right picture, with the vee pointing upwards.

Now try running **HEAD** and then **MAN** itself. You should get the correct result this time too:



If you would like to know more about the editor, turn to Appendix B, ‘Editing your procedures’. Up to now we have used only a few of the things you can do with it.

Summary of chapter

The most effective ways of keeping bugs out of your procedures are as follows:

1. Keep your procedures short and simple.
2. Keep detailed design notes, so that you know what your procedure is doing at each stage of its development.
3. Check your typing before you press the **RETURN** key.

If, despite these measures, you still get bugs in your procedures, look at the problem carefully and, if you still cannot see what is going wrong, use the **PAUSE** and **CONTINUE (CO)** primitives to isolate the bugs. If you put a **PAUSE** in your procedure it will stop when it gets to the **PAUSE** primitive. When you are ready, you can run it from that point by typing **CONTINUE**.

When you have found the bugs, you can correct your procedures using the **EDIT (ED)** primitive. This is described in detail in Appendix B, ‘Editing your procedures’.

8 Using inputs with your procedures



Up to now, whenever you have wanted to draw similar pictures of different sizes you have had to write a separate procedure for each size of figure. There's nothing wrong with this, but you can make life a lot easier for yourself by using only one procedure and giving it a special 'input' to pass different sizes into the procedure.

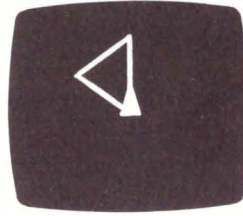
For example, type in the following procedure:

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

Now try the following:



TRIANGLES 100



TRIANGLES 300

What is actually happening is this: the first line of the procedure definition (the title line) names a 'variable' called `SIDE`. Think of a variable as a 'box'. Whenever you call the procedure, by typing something like:

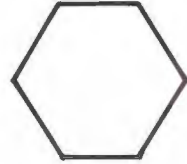
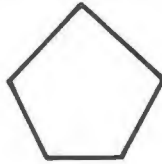
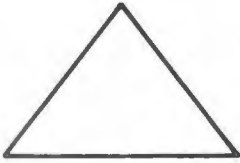
```
TRIANGLES 200
```

you put the input value (in this case 200) into the box called `SIDE`. Logo takes this value from the box and puts it wherever `:SIDE` occurs inside the procedure (`:` is called 'dots' in Logo). Notice that there is no space between the dots and the variable name (`SIDE`).

In the above example, the value 200 will be put into the box called `SIDE` and used inside the procedure. This makes the second line of the procedure look like the following line:

```
REPEAT 3 [FORWARD 200 LEFT 120]
```

Try writing procedures with inputs for some of the other figures you have drawn. You could try these to start with:



When you have finished experimenting, think about this new idea: if a procedure can have one input, why can't it have more? Below is the title line for a procedure with two inputs:

TO RECTANGLE :SIDE1 :SIDE2

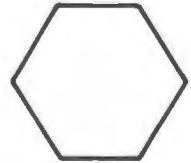
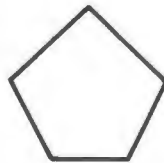
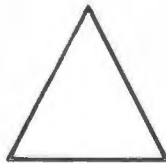
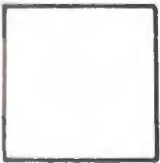
This is intended to draw a rectangle. See if you can write the rest of the procedure yourself.

Variables have many other uses and can have other things besides numbers as values. You will learn more about this when you read chapter 14, 'Playing with words'.

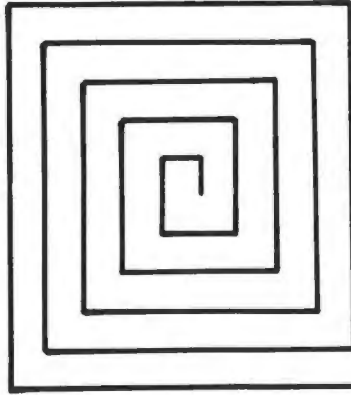


Turtle fun

1. Try to write some procedures with the input **:SIDE** which will draw the following figures:



2. Try to write a procedure SPIRALSQUARE which draws a picture like this:



3. Try writing other procedures to produce, for example, spiral triangles and six-sided figures.

Summary of chapter

Procedures which you define yourself can have inputs, just like primitives. These allow you to vary the size of items within the procedure. For example:

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

`SIDE` is a 'variable' and variables are preceded by dots (`:`) when referring to their value.

To use the procedure, you call it as follows:

```
TRIANGLES 100  
TRIANGLES 300
```

and whatever you typed as the input is used in place of `:SIDE` within the procedure.

You can use more than one input to a procedure.

9 Using numbers with Logo



You can use numbers with Logo and you have two ways of showing the result of any calculations.

First of all, you can use the turtle. Type the following in and watch what happens:

```
BACK 200  
FORWARD 50 + 200 - 50
```

Now think of a few calculations, work out what you think the results are, then check them using the turtle.

The symbol for multiplication is * and / stands for division. To see them in action, try the following:

```
FORWARD 10 * 20  
RIGHT 180 / 2
```

Now try a few other calculations using * and /.

The other way you can look at the result of a calculation is by using Logo's built-in calculator. Work out the following, then type it in just as it is, and watch what happens:

```
1 + 3 - 2
```

Logo will have worked out the result, but you have not told it what you want it to do next. If all you want to do is show the number on the screen, you can use the PRINT (PR) primitive:

```
PRINT "HELLO  
HELLO
```

```
PRINT 1 + 3 - 2  
2
```

Now repeat some of your earlier examples, but try using the calculator this time.

When you have finished experimenting, think about this calculation:

$$2 - 1 * 5 + 5$$

Work out what you think the answer should be, then try it out. Did the answer surprise you?

What actually happens is this: Logo tackles calculations in a particular order. Multiplication and division are done first, then addition and subtraction.

Now run through it again, using this rule to work out what is happening.

Changing the order of calculation

You can get Logo to work out one part of a calculation first by putting it in brackets. For example, look at this:

$$3 + 2 * 2$$

The calculation $2 * 2$ will be worked out first, because it is a multiplication. However, if you want to add the 2 to the 3 first you can do this by using brackets:

$$(3 + 2) * 2$$

Try out both calculations yourself and see what happens. Then try working out the following and check your answers:

$$20 * (10 - 5)$$

$$20 * (10 - 5 * 2) + 100$$

$$10 - (200 - 200 / 2)$$

Using decimals

You can use decimals with Logo, either on their own or mixed with whole numbers. Try the following and see what happens (the full stop key near the bottom of your keyboard is used to print the decimal point). Think about each line before you type it in.

```
FORWARD 100
```

```
LEFT 20
```

```
FORWARD 200 * 0.5
```

```
LEFT 60 / 3
```

```
FORWARD 400 * 0.25
```

```
LEFT 50 / 2.5
```

Make up a few calculations yourself for further practice. Work out beforehand what you think will happen, then check out your results with the turtle or the calculator.

Using negative numbers

You can also get negative numbers as a result of your calculations. For example, try the following:

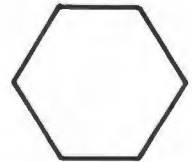
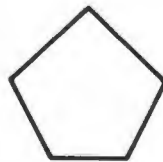
```
FORWARD - 100 - 100  
FORWARD 200
```

If you are not used to dealing with negative numbers, try a few more calculations using both the turtle and the calculator to check the results. Try using other symbols such as * with negative numbers and find out what happens with them.



Turtle fun

Try to write a procedure POLY with the inputs :SIDE and :ANGLE which will draw the following figures:



See what other figures you can get by changing :ANGLE.

Summary of chapter

Calculations can be used with primitives. For example:

```
FORWARD 100 + 200  
LEFT 30 + 60
```

They can also be used within procedures and by themselves.

The symbol for multiplication is * and / stands for division.

In calculations, Logo carries out multiplication and division first, then addition and subtraction. You can specify operations you want worked out first by enclosing them in brackets. In the following example:

```
20 * (10 - 5)
```

the subtraction will be worked out first.

You can use decimals and negative numbers in calculations.

You can display the results of a calculation with the PRINT (PR) primitive. For example:

```
PRINT 100 + 150
```

```
250
```

10 Using boxes with Logo



Imagine a cardboard box. You could give this a name based upon what you are going to use it for. If you want to put rubbish into it you could call it a rubbish bin, or you could put toys into it and call it a toy box. Whatever you call it, the contents of the box will probably vary from time to time: the rubbish you put into it today probably won't be the same as the rubbish you put into it tomorrow.

Logo has boxes as well, and these are called 'variables'. You have already come across one type of variable: the inputs you used with your procedures.

Logo variables have names, just like ordinary boxes. They also have contents that can vary from time to time; these are called Logo 'objects' and they include such things as numbers.

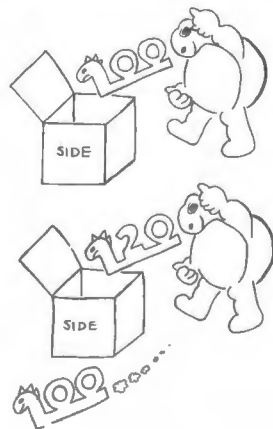
When you specify an input to a procedure, the value of the variable is passed to the procedure from the outside. In the procedure below, it is set up within the procedure itself by the **MAKE** primitive.

```
TO SPIRAL
MAKE "SIDE 100
REPEAT 32 [FD :SIDE LT 90 MAKE "SIDE (:SIDE + 20)]
END
```

SPIRAL is basically very similar to a square-drawing procedure, but after each side is drawn the variable **SIDE** has 20 added to it. So a 'spiral square' is drawn. Try it out and see for yourself.

What happens is this. The first **MAKE** primitive names the box **SIDE** and gives it the value 100 (the quotes (") tell Logo that you are referring to the name of the box).

Next, the **REPEAT** primitive uses the contents of the box to draw a line, turns 90 degrees, then puts another value, 120, into the box. The second **MAKE** primitive does this by adding 20 to the present value, 100.



When it next goes down this path, **MAKE** uses the new contents of the box and adds 20 again to give it a new value, 140. And so on . . .

If you are not sure about what is happening, read through it again. Variables are extremely useful in Logo, and you need to understand them properly.

When you feel confident, try modifying **SPIRAL** to draw a 'spiral triangle'. There are two ways you can do this: by varying either the side or the angle. Have a go at one of them first, then the other, then both.

You should remember one thing when defining variables: you can give them any name you want, but it is better if names are meaningful so that your procedures will be easy to understand.

Boxes with the same name

The names you use as inputs to your procedures are 'private' to the procedures themselves. As a result, you don't need to worry about Logo getting mixed up between inputs to different procedures. Look at the following procedures for example (and try them out for yourself):

```
TO PRIVATE :NUMBER
PRINT :NUMBER
DOPRINT :NUMBER +1
PRINT :NUMBER
END
```

```
TO DOPRINT :NUMBER
PRINT :NUMBER
END
```

Now look at the results:

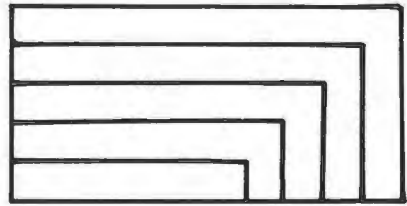
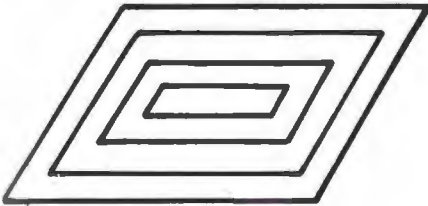
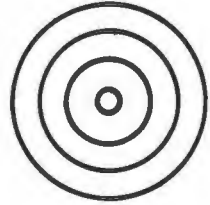
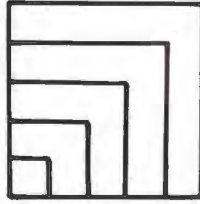
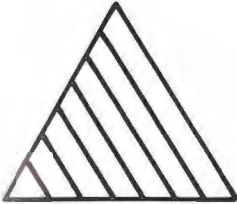
```
PRIVATE 1
1           (this is printed in PRIVATE)
2           (this is printed in DOPRINT)
1           (this is printed in PRIVATE)
```

NUMBER has the value 1 in **PRIVATE** and 2 in **DOPRINT**. This is because **PRIVATE** and **DOPRINT** each have their own, private, version of **NUMBER**.



Turtle fun

Try writing procedures to draw these shapes:



Summary of chapter

Inputs are just one example of a general Logo class called 'variables'.

Variables are like 'boxes'; they can have names and different contents. Their contents are called Logo 'objects'; numbers are one type of Logo object.

You can define a variable within the body of a procedure using the **MAKE** primitive. For example:

```
MAKE "NUMBER 10
```

gives the variable **NUMBER** the value 10. The quotes (") tell Logo that you are referring to the name of the variable.

You can give a variable any name you want, but it is better to stick to short, meaningful names so that your procedures are easy to follow.

What actually happens in `NEW.SPIRAL` is this: when the call to `NEW.SPIRAL` is made from within itself, another version of `NEW.SPIRAL` is created, and so on, until the whole thing is stopped using the `ESCAPE` key.

Try writing another version of `NEW.SPIRAL` that spirals inwards upon itself.

Stopping your spirals

Up to now, your recursive procedures will only stop if you press `CTRL` and `ESCAPE` together, or, in some cases, when the computer runs out of memory. There is a much neater way of stopping them. The following modified version of `NEW.SPIRAL` shows you how:

```
TO NEW.SPIRAL :SIDE
IF :SIDE > 500 [STOP]
FORWARD :SIDE
LEFT 90
NEW.SPIRAL (:SIDE + 20)
END
```



In this version, the `IF` primitive in the second line helps the turtle decide what to do by testing if a condition (`SIDE > 500`) is true or false. If it is true, the procedure stops; in other words, the `IF` primitive lets the procedure continue from where it left off if there is more still to do, or stops if not. Try changing your version of `NEW.SPIRAL`, using the editor, to include the new line.

Recursion without turtles

Here is an example of recursion which does not use turtles:

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

Try it out with a few examples like this:

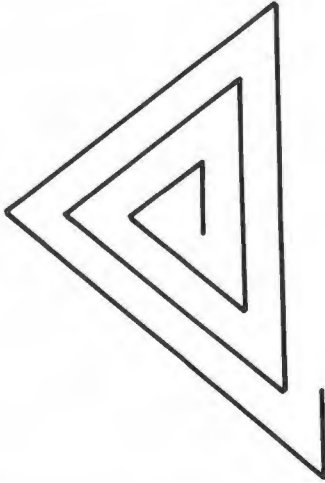
```
COUNTDOWN 10
```

Work out for yourself what is happening.



Turtle fun

1. Write a recursive procedure that draws spiral triangles like the following:



2. Write a similar procedure that varies the angle of the triangle.

Summary of chapter

A procedure can call itself, and this is known as 'recursion'. For example:

```
TO NEW.SPIRAL :SIDE
FORWARD :SIDE
LEFT 90
NEW.SPIRAL (:SIDE + 20)
END
```

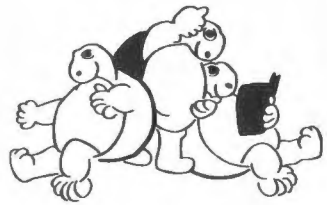
You can use a full stop as part of a procedure name (as above) to give it legibility.

You can stop a procedure by holding down the CTRL key and pressing the ESCAPE key.

You can also stop a procedure using the IF primitive in conjunction with STOP. For example:

```
TO NEW.SPIRAL :SIDE
IF :SIDE > 500 [STOP]
FORWARD :SIDE
LEFT 90
NEWSPIRAL (:SIDE + 20)
END
```

12 Multiple turtles



Until now you have had only one turtle on the screen, but Acornsoft Logo allows you to control more than one turtle at the same time.

The part of Logo which controls such 'multiple turtles' is held on disc or tape and is an 'extension' to the normal Logo system. To load it you must use the **LOAD** primitive described in chapter 5, 'Saving your procedures and pictures'. First of all load the cassette or disc which is included in your Logo package and then type:

```
LOAD "MULT
```

The extension will then be loaded.

The turtle on your screen is called 'turtle 0'. You can create other turtles using the **HATCH** primitive, and these are 'hatched' at the position turtle 0 occupies at the time. For example, type the following:

```
DRAW  
HATCH 1
```

You will not see the new turtle (turtle 1) yet, as it is hidden by turtle 0. In fact, if you turn turtle 0 by typing the following:

```
RIGHT 45
```

you still won't see it. This is because turtles are invisible when they are first hatched. To make one visible you must first tell Logo to talk to it using the **TELL** primitive:

```
TELL 1
```

You can then make turtle 1 visible and turn it by typing:

```
SHOWTURTLE  
LEFT 45
```

Type these lines in and watch what happens. You should have two turtles on your screen in the form of a 'V':



Primitives such as `LEFT` and `FORWARD` which are used after `TELL` has been called are applied *only* to the turtle selected. That is why turtle 0 didn't move.

If you want, you can select a number of turtles at the same time by putting their numbers in the `TELL` command in the following way:

```
TELL [0 1]
```

This selects turtles 0 and 1, and subsequent primitives are applied to *both* at the same time. To see the effect, type it in then type the following as well:

```
REPEAT 100 [FORWARD 100]
```

You should end up with a check pattern on the screen.

If you look at the last `TELL` command you will see that the turtle numbers are enclosed by square brackets. Anything that appears in this way is called a 'list', and you have come across these in two other places. First of all, you have used them with `REPEAT`. Then you used them in chapter 11, 'Recursion' (the `IF` statement contained the list `[STOP]`).

Lists have many uses and more of these are described in chapter 14, 'Playing with words'. For the present, think of them as a way of telling Logo that you want to do something to a number of turtles at the same time.

Next, have a look at the following procedures:

```
TO CROSS  
DRAW  
HATCH [1 2 3]  
START 0 START 1 START 2 START 3  
TELL [0 1 2 3]  
END
```

```
TO START :NUMBER  
TELL :NUMBER  
RIGHT :NUMBER * 90  
SHOWTURTLE  
END
```

Together, they hatch turtles one to three and put them into the positions shown in the diagram below. All you need do is type **CROSS**.



Notice the **DRAW** primitive at the very start of **CROSS**. As well as clearing the screen, it tells Logo that if any turtles have been hatched, it should forget about them and start from scratch again.

You can use these procedures to start off some interesting patterns. For example, try typing the following (if you still have the procedure called **SQUARE**, you don't need to type it in again):

```
TO SQUARE
REPEAT 4 [FD 100 LT 90]
END

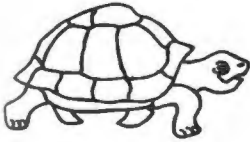
CROSS
FORWARD 200 SQUARE
```

Your turtles should draw four squares all at the same time. Try using procedures that draw other shapes. Then modify **CROSS** and **START** to handle more than four turtles.

Random movements

When you have finished exploring, ensure that **CROSS** and **START** are in their original form, then type the following and watch what happens (you can stop the movement at any time and get back to Logo by pressing **CTRL** and **ESCAPE** together):

```
CROSS
REPEAT 500 [LEFT (RANDOM 360) FORWARD (RANDOM 100)]
```



The brackets around `RANDOM 360` and `RANDOM 100` are not necessary, but are used to show that these things are worked out before `LEFT` and `RIGHT` are acted upon.

`RANDOM` is a new primitive. It takes the number used as its input and produces a random number less than this number. In the example used above, `RANDOM 360` tells Logo to choose any number between 0 and 359. This number is then used by `LEFT` to make the turtle turn left by this random amount. Each time `RANDOM` is called, it will pick a different number.

Work out for yourself what the other example of `RANDOM` does. Then try changing `CROSS` to give the turtles different colours.

Drawing multiple circles

Again, ensure that `CROSS` and `START` are in their original form, then type in the following procedure:

```
TO MULTI.SPIRAL  
CROSS  
DOFOREVER [FD 100 LT 10]  
END
```

Now try running it. You can stop it using the `CTRL` and `ESCAPE` keys!

`DOFOREVER` is a new primitive that is similar to `REPEAT`. It runs the list of commands inside the square brackets, then does the same thing again, . . . and again, and goes on forever! To stop it, hold down the `CTRL` key then press `ESCAPE` (there are other ways of stopping it, using `IF` but we don't need them here).

Try changing the inputs to `FD` and `LT` and run the procedure with different values. Then try defining the angle of turn outside the `DOFOREVER` list, using `MAKE`, and changing its value within the list.

Moving letters

You can redefine the shape of your turtles to produce space ships, animals and other things. There are a number of ways of doing this and they are all described in the Logo reference manual. For the present, we will look at just one.

First of all, use the Logo editor to change your procedure `START` to the following (the new lines are underlined):

```
TO START :NUMBER  
TELL :NUMBER  
SETSH 65 + :NUMBER  
PENUP  
RIGHT :NUMBER * 90  
SHOWTURTLE  
END
```

Now type the following and watch what happens! You should get part of the alphabet flying around your screen!

```
CROSS  
REPEAT 500 [LEFT (RANDOM 360) FORWARD (RANDOM 100)]
```

The important line is the one containing the new primitive, `SETSH` (`SET SHape`). `SETSH` takes a number or a list as its input and uses this to redefine the current turtle's shape. In our case it redefines turtles 0 to 3 to have the values 65 to 68, and these correspond to the shapes A to D.

Try putting some other numbers after `SETSH` and see what happens. If you want to know more about this subject, look at the Logo reference manual.



Turtle fun

1. Try to get the whole alphabet flying around the screen.
2. See if you can get two turtles starting at opposite sides of a circle and moving together. Hint: the following procedure draws a circle:

```
TO CIRCLE  
REPEAT 180 [FD 10 LT 2]  
END
```

Summary of chapter

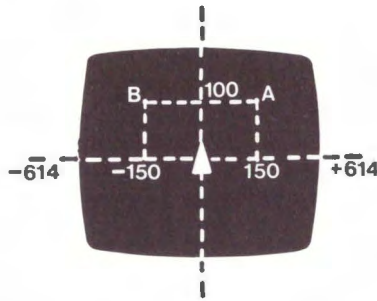
1. New turtles can be 'hatched' at the current turtle position using the **HATCH** primitive. This can apply to one turtle or a list of turtles.
2. When turtles are hatched they have the same position as the current turtle but they are invisible.
3. You tell Logo to 'talk' to a specific turtle or list of turtles using the **TELL** primitive. Subsequent commands then apply to this (those) turtle(s).
4. To make a turtle visible you must first use **TELL** and then **SHOWTURTLE**.
5. You can redefine a turtle's shape using the primitive **SETSH**. This takes a number or list of numbers as input and these are used to define the shape.
6. The primitive **RANDOM** can be used to generate random numbers. It takes one input and the number returned is between zero (inclusive) and the value of this input (exclusive). Each time **RANDOM** is used it produces a different number:
7. The primitive **DOFOREVER** is used to repeat a list of commands forever. You can stop it using **CTRL** and **ESCAPE**.

13 Turtle navigation



If you want to get to a particular place in the world, or tell someone else how to, it helps if you have a map. Have a look at a map of the area where you live. You will probably find that it has a 'grid' upon it which consists of lines drawn from north to south and others running east to west. Using these lines you can pinpoint any location and say, for example, 'London Street is in box E4,' or 'Bristol has the position 110637.'

The turtle's world consists of the screen and you can pinpoint any position on it in a similar way. If you draw a horizontal line and a vertical line through the home position, as shown below, the turtle's position can be shown in terms of these lines.



The horizontal line is called the 'x-axis' and the vertical line the 'y-axis'. We show the position of a point on the screen, relative to these axes, in the form $[x\ y]$. The home position is represented by $[0\ 0]$ and point A in the diagram has the position $[150\ 100]$. In mathematics, numbers like this which represent a position are called 'coordinates', and we will use this word from now on.

Getting to the point

You can move the turtle to an exact position on the screen using the SETPOS primitive, which has the general form:

```
SETPOS [x y]
```

You could move it to the position A, $[150\ 100]$, by typing this:

```
SETPOS [150 100]
```

Try it and see. Then clear the screen again using `DRAW`.

There is another way in which you can move to the point A (apart from using `RIGHT` and `FORWARD`, of course). Type the following primitives and watch what each one does:

```
SETX 150  
SETY 100
```

Now try to move the turtle to the point B. Remember, to get a number less than 0, you must put a minus sign (-) before it.

If you want to turn the turtle to point to a given 'heading' as well, you can do so with the `SETH` (`SET Heading`) primitive. For example:

```
SETH 180
```

makes the turtle point 'south'. Try it and see.

Find the turtle

As well as getting to a particular point yourself, you might have to describe to someone else where it is so that they can join you. You can get the turtle to tell you exactly where it is (even if it is off the screen) using the primitives `POS`, `XPOS` and `YPOS`.

Move the turtle to the point shown as A on the last diagram. Then type the following and watch what happens each time:

```
PRINT POS  
PRINT XPOS  
PRINT YPOS
```

The first of these prints the position of the turtle, the second its x-coordinate and the third its y-coordinate. Notice that the square brackets are not printed. Try moving the turtle around a bit and use these primitives a few times.

When you have finished experimenting, look at this procedure then try it out. All it does is draw a circle:

```
TO CIRCLE  
REPEAT 360 [FD 5 LT 1]  
END
```

Now add another primitive which prints the turtle's position after each move:

```
TO CIRCLE  
REPEAT 360 [FD 5 LT 1 PRINT POS]  
END
```

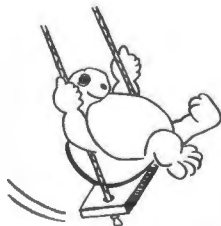
When you have typed it in, try the following and watch how the printed coordinates change. If you want to stop the turtle and examine them, you can do so by pressing CTRL and SHIFT together; if you release these keys, the turtle will continue its movement.

```
SETX 300  
CIRCLE
```

Turtle Island

Real turtles have a marvellous navigation system which allows them to travel across thousands of miles of sea and still return to the same place to lay their eggs. Figure 3 shows 'Turtle Island' and the spot (●) where they lay their eggs. Try to draw the island and the features shown by using the primitives you have found in this, and previous chapters.

When you have drawn the island, start the turtle off from its home position and take it to where it lays its eggs without crossing the land. (Hint: use a procedure to draw the island.)



Turtle fun

Hatch turtle 1 somewhere on your screen using SETPOS and put turtle 0 back in the home position. Now, 'home' turtle 0 on to turtle 1. To help you a little, the following primitives set the heading of turtle 0 to point towards turtle 1:

```
SETH TOWARDS (POS 1)
```

Next, hatch four turtles at the corners of the screen and get each one to chase its left hand neighbour.

Summary of chapter

The turtle's position can be given in the form of coordinates [x y], where [0 0] is the home position. You can move the turtle to a specific point and change its heading using the following primitives:

1. SETX moves the turtle in the x direction, for example:
SETX 200

2. SETY moves the turtle in the y direction, for example:
SETY 150

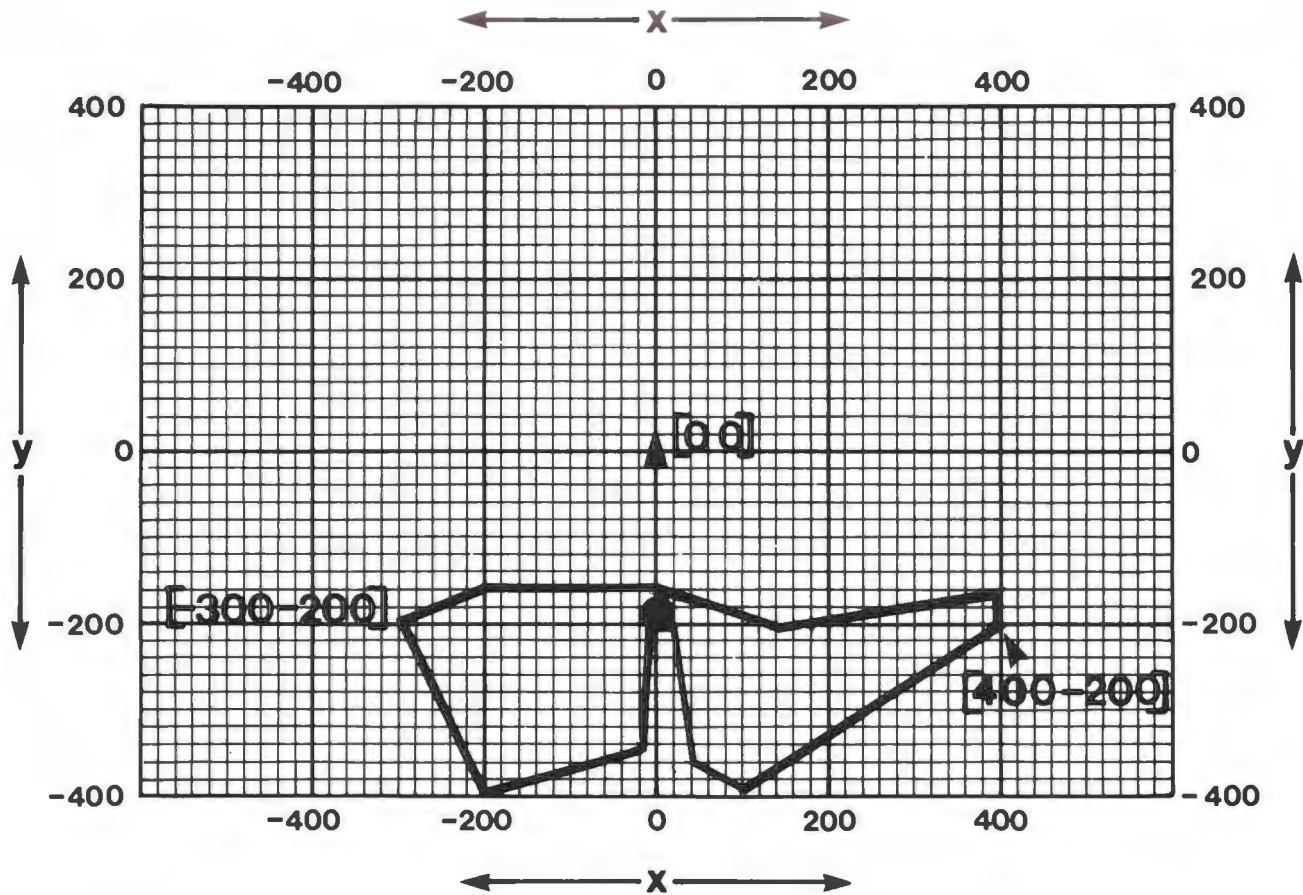


Figure 3 Turtle Island

3. **SETPOS** moves the turtle in both the x and y directions, for example:

SETPOS [200 -150]

4. **SETH** alters the turtle's heading, for example:

SETH 90

You can find out the turtle's position using the following primitives:

1. **XPOS** returns the x-coordinate of the turtle's position.

2. **YPOS** returns the y-coordinate of the turtle's position.

3. **POS** returns both the x- and y-coordinates of the turtle's position in the form [x y].

14 Playing with words



Logo is one of several computer languages designed to help you talk to a computer. Most computer languages (for example, BASIC and FORTRAN) have been designed to manipulate numbers. In computer jargon, they are called 'number crunchers' because they devour numbers with great relish!

You have probably heard of computers which perform mathematical calculations of amazing complexity in only a fraction of a second. These will have been programmed with a language of this type.

The problem with number crunchers however, is this: they don't like words. If you try and instruct a computer to deal with ordinary English, using a language devised originally to deal with numbers, it is difficult, if not impossible.

Logo is a member of a group of programming languages which can, however, cope with words. This doesn't mean that it can't do arithmetic: you have already seen that it can!

In this chapter, you will find out a little of how Logo handles words and sentences. Once you have learned this you will be encouraged to enter a world of exploration.

How does Logo handle words?

A computer cannot attach meaning to a word in the same way that we do. For instance, if you type **MOUSE** and **FURRY** at the keyboard and then press **RETURN**, the computer will remain totally unimpressed. It will give you a message equivalent to 'so what?'

This happens because the computer has been given absolutely no clues at all about how **MOUSE** and **FURRY** relate to one another, or indeed what sort of words they are.

When we use words to speak to one another, we group them together in sentences. We also use special words like 'the', 'an', 'a', and special symbols like commas and full stops as clues to show what part each word plays in the sentence. In order to get a computer to use words, much the same ideas apply. Two things are needed: a way of grouping words together and rules which clearly indicate where one word has a particular role within the group.

In Logo, the 'word cruncher' lists are the central and most important tool. Here is a Logo list:

[NOSE PRETTY PINK]

This list resembles a sentence in that it is a group of words. Notice that the beginning and end of the list are marked by square brackets and that the words are separated by spaces. The computer, of course, hasn't the slightest idea what the individual words mean. All it 'sees' is groups of characters. For this reason, total nonsense words and even numbers are quite acceptable to it: Ax99Z, BULLABR, POORQ, GU and G are happily dealt with as Logo objects in a list. In this case, the Logo objects are also Logo words.

Sometimes the order of words in a list looks strangely like a sentence! For instance:

[THIS NOSE IS PRETTY AND PINK]

To Logo, this is just another list. So long as each Logo word is separated by a space and is within square brackets, Logo will be able to use it.

We said earlier that symbols indicating particular roles and relationships of words were useful in helping computers to 'word crunch'. Let's now look more carefully at how square brackets make lists in Logo.

Look at the following example. It's a description of a house:

MAKE "HOUSE1 [LOUNGE KITCHEN STAIRS BATHROOM BEDROOM]

Look at this example, too:

MAKE "HOUSE2 [[LOUNGE KITCHEN] STAIRS [BATHROOM BEDROOM]]

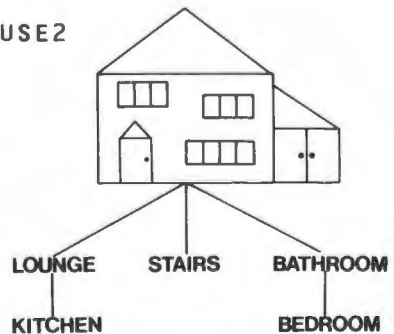
The first is relatively simple to understand. The word HOUSE has been made the name of a long list of rooms. In the second description, some extra brackets have been added. Look at the effect that this has on our model:

HOUSE1



LOUNGE
KITCHEN
STAIRS
BATHROOM
BEDROOM

HOUSE2



LOUNGE STAIRS BATHROOM
KITCHEN BEDROOM

In the second model, two smaller lists have been made on the basis of downstairs and upstairs rooms. These are 'lists within a list'. The computer knows about their existence on account of the new sets of square brackets placed within the original pair.

Getting words out of lists

You must by now be wondering how, once you have made a list like this, you get the words 'out' again. This would be impossible were it not for special Logo tools provided for that very purpose. These are the primitives needed:

FIRST	Returns the first element in a list
BUTFIRST (BF)	Returns all but the first element in a list
LAST	Returns the last element in a list
BUTLAST (BL)	Returns all but the last element in a list

To see how they work, type in the first description of the house:

```
MAKE "HOUSE1 [LOUNGE KITCHEN STAIRS BATHROOM BEDROOM]
```

Now type:

```
PRINT FIRST :HOUSE1
```

You should get the answer:

```
LOUNGE
```

Try also:

```
PRINT LAST :HOUSE1
```

```
PRINT BUTFIRST :HOUSE1
```

```
PRINT BUTLAST :HOUSE1
```

and lastly:

```
PRINT :HOUSE1
```

This should give all the rooms in the house.

Getting little lists out of big ones

By now, you ought to be familiar with **FIRST**, **BUTFIRST**, **LAST** and **BUTLAST** as tools for extracting words from lists.

Now try out the same four operations on the second model of the house:

```
MAKE "HOUSE2 [[LOUNGE KITCHEN] STAIRS [BATHROOM BEDROOM]]
```

This time you should find that the 'lists within a list' are treated in exactly the

same way as were the individual words in the first example. This means that, for instance:

```
PRINT FIRST :HOUSE2
```

gives

```
LOUNGE KITCHEN
```

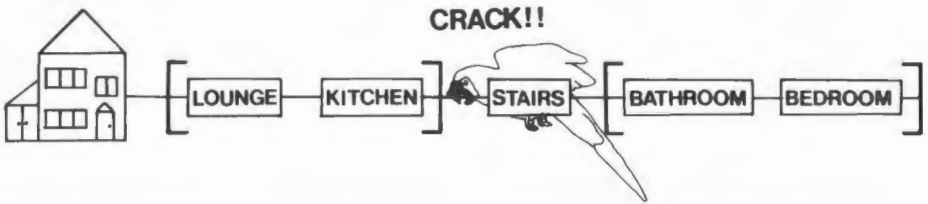
– the downstairs!

How can you print a list of upstairs rooms? See if you can work it out for yourself!

To print the stairs alone, we do this:

```
PRINT FIRST BUTFIRST :HOUSE2
```

Logo works this out from the inside outwards, so that `BUTFIRST` is applied initially to the list. This causes the stairs and upstairs to be nibbled off together and the downstairs ‘thrown away’:

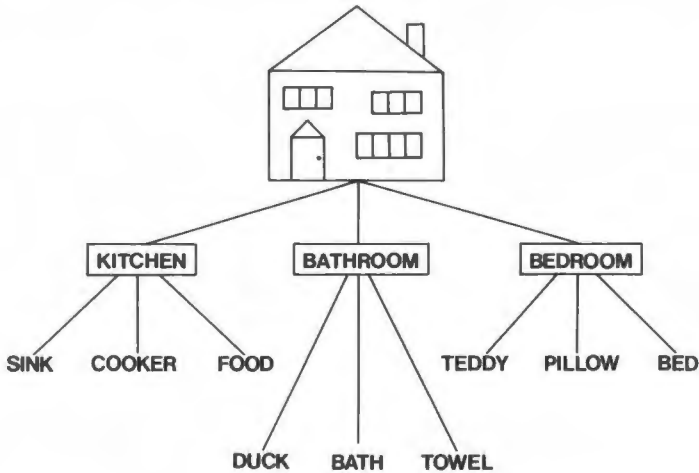


Then, `FIRST` takes the first item of the list left over and removes and prints it:



Joining lists together

In order to give the computer a more detailed model of an object such as a house you might like to take several descriptive lists and join them together to make a 'tree' like this:



You can do this by joining up the lists for each room. First, make up the lists for each room:

```
MAKE "KI [KITCHEN [SINK COOKER FOOD]]
MAKE "BA [BATHROOM [BATH TOWEL DUCK]]
MAKE "BE [BEDROOM [TEDDY PILLOW BED]]
```

Now join them up under one roof! This is done using the LIST primitive:

```
MAKE "HOUSE (LIST :KI :BA :BE)
```

The brackets on this line are needed to tell Logo that LIST has more than two inputs.

How does HOUSE now behave towards FIRST, BUTFIRST, LAST and BUTLAST? How many 'levels' does the computer model have? Experiment with these Logo tools at your disposal until you understand the model clearly. Then build a more complicated one!

Words can be lists too!

So far you will have gathered that Logo can cope with lists of words and even lists-within-lists of words. What you may not realise is that words themselves can be seen as lists. This becomes apparent when you apply the Logo primitives for breaking down lists to words themselves.

For example, try these:

```
PRINT "INFATUATE
PRINT FIRST "INFATUATE
PRINT BUTLAST BUTLAST BUTLAST BUTLAST "INFATUATE
PRINT BUTFIRST "INFATUATE
```

Don't forget to put quotes (") before INFATUATE. This tells Logo that INFATUATE is a word and it is always needed when the word is not part of a list.

You can combine a number of words to form a longer one using the WORD primitive. For example, type the following and watch the result:

```
PRINT WORD "CAT "FISH
```

See if you really understand how to break up a list by printing the words ATE, U, IN and FAT from INFATUATE. Then combine them again using WORD (if you try to use more than two inputs to WORD you must surround it with brackets, just as we did in the LIST example above).

Writing back-to-front

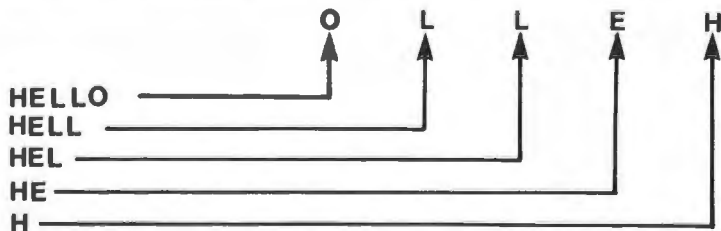
You already have a number of useful list processing commands in FIRST, BUTFIRST, LAST, BUTLAST and PRINT. In this section you are going to find out now how to build another list-processing command from existing Logo commands, or primitives.

Let's try writing a procedure called, say, REVERSE that will allow you to do the following:

```
PRINT REVERSE "HELLO
OLLEH
```

```
PRINT REVERSE "THERE
EREHT
```

What we are doing here is stripping off the last letter (O), then taking the remaining word, (HELL) and stripping off its last letter (L), then taking the remaining word (HEL) and stripping off its last letter . . . and so on.



Sounds just like recursion again, doesn't it? In fact, you would use a recursive procedure to do it:

```
TO REVERSE :TEXT
OUTPUT WORD (LAST :TEXT) (REVERSE BUTLAST :TEXT)
END
```

The brackets on the second line are used to show you the inputs to **WORD**.

The **OUTPUT (OP)** primitive in the second line of **REVERSE** passes back the value of its input to the calling procedure. In this case, the value passed back is the result of the **WORD** primitive. **OUTPUT**, in this case, effectively joins the last letter of the word to all the earlier letters in reverse order.

If you now try out this procedure, it won't work! There is nothing inside **REVERSE** that tells it how to stop. It keeps on going until it tries to apply **LAST** to a word with no elements, an 'empty word'. Try it and see.

What you need to do is stop it when it gets to the empty word. You can do this by inserting the following line as the first line of the procedure:

```
IF :TEXT = " [OUTPUT "]
```

Include it in your procedure, now, and try it out.

" on its own is used to refer to an empty word, and it is called 'the empty word'. **[]** on its own is used to refer to an empty list and it is called 'the empty list'.

You could use **REVERSE**, now, to test if a word reads the same in reverse as it does forwards (this type of word is called a 'palindrome'):

```
TO PALINDROME :TEXT
IF :TEXT = REVERSE :TEXT [OUTPUT "TRUE]
OUTPUT "FALSE
END
```

```
PRINT PALINDROME "KAYAK
TRUE
```

```
PRINT PALINDROME "KAY
FALSE
```

Try it and see. Then try to understand how the procedure works.



Turtle fun

If sentences can be seen as lists of words, it should be possible to first break them up and then join them together to form new and different sentences.

Try to use Logo to make some new sentences from the following:

```
[Pigs are [rather fat [with hairy ears]]]
[You are [extremely beautiful [and very clever]]]
```

Summary of chapter

1. Groups of characters such as "FRED and "ABC123 are called 'words'. A word is always preceded by quotes (").
2. Words separated by spaces and enclosed in square brackets are called 'lists', for example:

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

3. Words can be joined using the WORD primitive, or broken into shorter strings using the FIRST, LAST, BUTFIRST (BF) and BUTLAST (BL) primitives. For example:

```
PRINT WORD "CAT "FISH
CATFISH
```

```
PRINT FIRST "CAT
C
```

4. Lists can be combined to form longer strings using the LIST primitive, or can be broken into shorter ones using the FIRST, LAST, BUTFIRST and BUTLAST primitives. For example:

```
LIST [OTTAWA LONDON] [OSLO HARARE WASHINGTON]
returns [[OTTAWA LONDON] [OSLO HARARE WASHINGTON]]
PRINT FIRST [CATS ARE FURRY]
CATS
```

5. A word containing no characters is called 'the empty word'. It is represented by "" on its own.
6. A list containing no elements is called 'the empty list'. It is represented by [].
7. The OUTPUT (OP) primitive passes back to the calling procedure the value of its input.

15 Writing interactive procedures



'Interactive' procedures are ones which let you change what is happening in the computer as it is happening. Arcade games are good examples.

This chapter shows you how to write interactive procedures. It uses, as an example, a procedure which allows you to control the movement of the turtle with only five keys; such a procedure could be used to show small children the turtle at work.

Reading characters from the keyboard

You can read characters such as the letters A to Z from the keyboard using the primitive RC (Read Character). Look at the following, for example:

```
TO INPUT
MAKE "LETTER RC
IF :LETTER = "@" [STOP]
PRINT :LETTER
INPUT
END
```

```
INPUT
ASDFQWER
```

This procedure puts the value of the key you press into LETTER and will print any character you type other than @. When it reads this character it will stop. Type it in and try it for yourself. Then see if you can modify INPUT so that it stops when you press the 1 key.

Next, look at the following procedures. They allow you to control the turtle using only five keys:

```
TO TURTLEMOVE
GETKEY
TURTLEMOVE
END
```

```
TO GETKEY
MAKE "KEY RC
IF :KEY = "F [FORWARD 10]
```

```
IF :KEY = "L [LEFT 15]
IF :KEY = "R [RIGHT 15]
IF :KEY = "B [BACK 20]
IF :KEY = "D [DRAW]
CI
END
```

CI (Clear Input) is another new primitive. It merely tells Logo to forget any keys that were pressed before it was called.

Work out what you think is happening. Then type the procedure in and try it out.

You could also, if you wish, redefine other keys to produce particular shapes. For instance, you could use the S key to draw a square and the T key to draw a triangle. You will need to alter GETKEY.

You could also try thinking of some new ideas, such as getting the turtle to draw other figures or draw lines in different colours.

Improving your procedures

RC makes your procedure stop and wait until a key is pressed before it does anything. This would not be much use in a fast-moving arcade game.

Logo provides another primitive which allows your procedure to run continuously and be stopped only when a key has been pressed: the KEYQ primitive. This outputs the value "TRUE if a key has been pressed and "FALSE otherwise. If a key has been pressed, you can then get the character using RC.

Your procedures might now look like the ones shown below. The underlined parts are the ones that have been changed. Also, the line in GETKEY that moved the turtle forward has been deleted so that the turtle always moves forward unless one of your special keys is pressed.

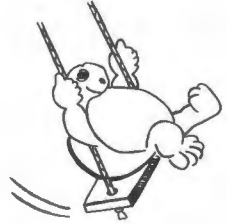
```
TO TURTLEMOVE
FORWARD 10
IF KEYQ [GETKEY]
TURTLEMOVE
END

TO GETKEY
MAKE "KEY RC
IF :KEY = "L [LEFT 15]
IF :KEY = "R [RIGHT 15]
```



```
IF :KEY = "B [BACK 20]  
IF :KEY = "D [DRAW]  
CI  
END
```

Put these new lines in, using the editor, then satisfy yourself that they work.



Turtle fun

1. Set up a 'target' on the screen and use the procedures you have defined to 'home' the turtle on to it.
2. In chapter 12, 'Multiple turtles', you may have written some procedures to home one turtle on to another. Try to modify these so that you can control the 'homing' turtle from your keyboard.
3. Change the same set of procedures so that this time you are controlling the 'target', instead of the 'homing' turtle, from your keyboard.

Summary of chapter

1. Interactive procedures let you change what is happening in the computer as it is happening.
2. Your procedure can read characters from the keyboard using RC (Read Character). This waits until a key is pressed then outputs its value.
3. CI (Clear Input) tells Logo to ignore any text input at the keyboard.
4. KEYQ can be used to make your procedures faster. This outputs the value TRUE if a key has been pressed and FALSE otherwise. When the value is TRUE, you can read the character using RC.

This introduction to Logo is now complete and we hope you have enjoyed exploring its uses. But don't stop here. The things we have covered are just a small part of Logo. To find out more, look at *Logo on the BBC Microcomputer and Acorn Electron* and read some of the books given under 'Further reading' at the back of this book.

Appendix A

Editing your command lines

As well as the DELETE key, which allows you to correct keyboard mistakes, there is a group of five keys on the right hand side of the keyboard which can be used to edit or alter program lines displayed on your screen. The arrow keys enable you to move a flashing cursor around the screen to a line that you want to edit. As soon as you press one of these keys, the computer enters a special 'editing mode' where it displays two cursors. The large white block is called the 'write cursor' and it shows you where anything you enter will appear. The other small, flashing cursor (the 'read cursor') is the one that can be moved around the screen by the arrow keys.

If you move the read cursor (using the arrow keys) until it is under a letter and then press the COPY key, everything that the read cursor passes under will be copied into the new input line. Halfway through copying a line you can always use the arrow keys to move the read cursor to some new place on the screen before using COPY again to duplicate some other text on your new input line. You can also use the DELETE key to delete characters from the input line, or you can type in new characters at any time. When you have completed your new input line you just press RETURN in the normal way.

Appendix B

Editing your procedures

You get into the Logo editor by typing `EDIT`, followed by quotes then your procedure name. For example:

```
EDIT "VEE
```

You can get out of it at any time, leaving your original procedure unchanged, by pressing `ESCAPE`. To carry out the changes you have made, you must leave the editor by pressing `COPY`.

When you get into the editor, the screen might look like this:



If any text is too long to fit on to one line, the part that overflows onto the next line is displayed in what is called 'inverse video': you get dark letters printed on a light background.

You can move the cursor around your procedure using the cursor control keys at the right of your keyboard (the arrow keys). The following list shows what you can do to your procedures and the actions and keys involved:

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

Function	Actions necessary
Move cursor down one row	Press the ↓ key
Move cursor to start of Logo line	Hold down the CTRL (BBC) or FUNC (Electron) key then press the ← key
Move cursor to end of Logo line	Hold down the CTRL (BBC) or FUNC (Electron) key then press the → key
Move cursor to top of text	Hold down the CTRL (BBC) or FUNC (Electron) key then press the ↑ key
Move cursor to bottom of text	Hold down the CTRL (BBC) or FUNC (Electron) key 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 down the CTRL (BBC) or FUNC (Electron) key and press N simultaneously, or move the cursor to the end of the previous line and press RETURN
Delete character at cursor position	Hold down the CTRL (BBC) or FUNC (Electron) key then press the D key
Delete character before cursor	Press the DELETE key
Delete from cursor to end of line	Hold down the CTRL (BBC) or FUNC (Electron) key and press the L key
Delete line	Move the cursor to any point on the line, then hold down the CTRL (BBC) or FUNC (Electron) key and press the U key
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	Press the ESCAPE key
Exit from the editor and preserve the edited procedure	Press the COPY key

Appendix C

Use of colour

You can change the pen and background colours, if you want, using the `SETPC` and `SETBG` primitives. The colours you can use depend upon the 'screen mode'.

Eight screen modes are available with the BBC Microcomputer, and seven with the Electron, but only five are relevant to Turtle Graphics. They are summarised below:

Mode	Description
0	This uses two colours with very high resolution graphics and needs 20K of memory (16K on US machines) to map the screen.
1	This uses four colours with high resolution graphics and needs 20K of memory (16K on US machines).
2	This uses 16 colours with medium resolution graphics and needs 20K of memory (16K on US machines).
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.

The mode in use when Logo is loaded is Mode 4. You can change the screen mode by typing `SETMODE` followed by the number corresponding to the mode you want. For example:

```
SETMODE 5
SETBG 1
SETPC 2
SETNIB 80
REPEAT 4 [FORWARD 100 LEFT 90]
```

You should end up with a yellow square shape on a red background. The `SETBG` command defines the screen colour and the `SETPC` command defines the pen colour. The numbers you can put after them are given below, for each valid screen mode.

Colour numbers					Screen/pen colour
Mode 0	Mode 1	Mode 2	Mode 4	Mode 5	
0	0	0	0	0	black
	1	1		1	red
		2			green
	2	3		2	yellow
		4			blue
		5			magenta (blue-red)
		6			cyan (blue-green)
1	3	7	1	3	white
		8			flashing black-white
		9			flashing red-cyan
		10			flashing green-magenta
		11			flashing yellow-blue
		12			flashing blue-yellow
		13			flashing magenta-green
		14			flashing cyan-red
		15			flashing white-black

Appendix D

Logo primitives

Below is a list of the primitives you have used in this book. There are many more. For a full list, see the Logo reference manual.

Primitive	Effect
BACK(BK) <n>	Moves turtle <n> steps back.
BUTFIRST(BF) <object>	Returns all but first element of <object>.
BUTLAST(BL) <object>	Returns all but last element of <object>.
CI	Clears the keyboard input buffer.
CLEAN	Clears graphics screen without moving turtle.
CONTINUE(CO)	Resumes a procedure after a PAUSE.
DOFOREVER <list>	Repeats list forever, or until a STOP is encountered.
DRAW	Clears screen, kills all but one turtle, sets WRAP, moves turtle to [0 0], sets heading to 0, resets pen state.
EDIT(ED) <name>	Starts the Logo editor and loads the procedure <name> into the edit buffer. If <name> is not present, displays edit buffer or nothing (if contents have been erased).
ERASE <procedure>	Erases the named procedure.
ERPS	Erases all procedures.
FENCE	Fences the turtle within outline of screen.
FIRST <object>	Returns first element of <object>.
FORWARD(FD) <n>	Moves turtle <n> steps forward.
HATCH <turtles>	Creates (hatches) turtle or turtles at current turtle position.
HIDETURTLE(HT)	Makes turtle invisible.

HOME	Moves turtle to [0 0] and sets heading to 0.
IF <expression> <list>	If <expression> is TRUE, runs <list>. Returns a value, if <list> does.
KEYQ	Returns TRUE if a key has been pressed, but not used by RC, otherwise returns FALSE.
LAST <object>	Returns last element of <object>.
LEFT(LT) <degrees>	Turns turtle <degrees> to left (anticlockwise).
LIST <object1> <object2>	Returns a list whose elements are <object1>, <object2>.
LOAD <name>	Loads file <name> into workspace.
MAKE "<name> <object>	Makes <name> refer to <object>.
OUTPUT(OP) <object>	Returns control to caller and returns <object> as result of a procedure.
PAUSE	Makes procedure pause.
PE	Puts turtle's eraser down.
PENDOWN(PD)	Puts turtle's pen down.
PENUP(PU)	Lifts turtle's pen.
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.
PRINT(PR) <object>	Prints <object> in text area and ends text with a RETURN.
PO <procedure>	Prints out definition of <procedure>.
READPICT <name>	Reads the picture from the file <name>.
RANDOM <number>	Returns a random, non-negative integer less than <number>.
RC	Returns character typed at keyboard and waits if necessary.
REPEAT <n> <list>	Executes <list> <n> times.

RIGHT(RT) <degrees>	Turns turtle <degrees> to the right (clockwise).
SAVE <name>	Writes the entire workspace to the file <name>.
SAVEPICT <name>	Saves the current screen picture into the file <name>.
SETBG <n>	Changes background colour to <n>.
SETHEADING(SETH) <degrees>	Sets turtle heading to <degrees>.
SETMODE <n>	Selects display mode of computer (see Appendix C).
SETNIB <n>	Selects graphic option of turtle.
SETPC <n>	Changes turtle's pen colour to <n>.
SETPOS <pos>	Moves turtle to <pos>, where <pos> is a list [x y].
SETSH <object>	Redefines turtle shape.
SETX <xpos>	Moves turtle horizontally to x-coordinate <xpos>.
SETY <ypos>	Moves turtle vertically to y-coordinate <ypos>.
SHOWTURTLE(ST)	Makes turtle visible.
STOP	Stops procedure and returns control to caller.
TELL <turtles>	Selects turtle or turtles and applies subsequent commands to it/them.
TO <name> <inputs>	Starts definition of procedure <name>.
TOWARDS <pos>	Returns heading turtle would have if it faced <pos>. <pos> is a list [x y].
TS	Allows the entire screen to be used for text.
WINDOW	Removes bounds from turtle field.
WORD <word1> <word2>	Returns word made up of <word1> and <word2>.
WRAP	Wraps turtle field round edges of screen.
XPOS	Returns x-coordinate of turtle's position.
YPOS	Returns y-coordinate of turtle's position.

Glossary

Commands	These are the means by which you can control the turtle or tell Logo to perform some other action. They include primitives and procedures.
Computer language	The medium used when you 'talk' to a computer. There are several languages for different purposes. Logo is designed for ease of use and to support drawing and work on words.
Expression	A collection of one or more words that you can use to make something happen to the turtle.
Input	A number, word or list which makes a command give different effects. For example: <code>FORWARD 100</code> <code>FORWARD 200</code>
List	A number of elements which can be words, other lists, or a combination of both. They are contained within square brackets, for example: <code>[LONDON [TORONTO WASHINGTON]]</code>
Memory	The computer has two types of memory: its immediate memory, or workspace, and its filing system memory (discs or cassette tape). When the computer is switched off, the contents of its workspace are lost. To keep them safe, you must store them on disc or cassette tape.
Name	A word used to identify a variable or file.
Object	A word or list.
Primitive	These are commands which are built into Logo, for example <code>FORWARD</code> and <code>RIGHT</code> .
Procedure	You can make up new commands of your own using the primitive <code>TO</code> . These are called procedures.

Variable	A named object whose contents can vary. You can think of a variable as being a 'box' whose contents you can change using either the input of a procedure or MAKE.
Word	A string of letters and/or numbers preceded by quotes ("). For example: "R2D2 "FRED
Workspace	(see Memory.)

Further reading

ABELSON, Harold
diSESSA, Andrea

Turtle Geometry: The Computer as a Medium for
Exploring Mathematics (MIT Press)
(For the more advanced reader)

ABELSON, Harold

LOGO for the Apple II (Byte/McGraw-Hill)

PAPERT, Seymour

Mindstorms: Children, Computers and Powerful
Ideas (The Harvester Press)

Index

- BACK (BK) 4
- Background colour 12,16
- Bugs 34
- BUTFIRST (BF) 67
- BUTLAST (BL) 67

- Calculation – order of 44
- CAT 27
- CI 75
- CLEAN 5
- Colour 16,80
- Commands 4,23
- Computer language 65
- CONTINUE (CO) 36
- Coordinates 60
- Cursor 4
 - read 77
 - write 77

- Debugging 34
- Decimals 44
- Default 14
- Defining a procedure 21
- DOFOREVER 57
- Dots 40
- DRAW 5,24

- EDIT (ED) 37
- Editing command lines 77
- Editing procedures 78
- Editor 32,37
- Empty list 71
- Empty word 71
- ERASE 21
- Eraser 12
- Erasing files 27
- ERFILE 27
- ERPS 25,26
- Extensions 54

- Fence 14
- FENCE 15
- Field 4,14
- Files – erasing 27
- FIRST 67
- Floor turtle 3
- FORWARD (FD) 4

- HATCH 54
- HOME 5
- Home position 4
- HIDETURTLE (HT) 13

- IF 51
- Input 4,40
- Interactive procedures 74
- Inverse video 78

- KEYQ 75

- LAST 67
- LEFT (LT) 4
- Lists 66,67
 - joining 69
- LOAD 25,26
- MAKE 47
- Memory 85
- Mistakes 5
- Mode 16
- Modules 31
- MULT 54
- Multiple turtles 54

- Name 47
- Negative numbers 45
- Nibs 13
- Numbers 43

Objects 47,60
Order of calculation 44
OUTPUT(OP) 71

PAUSE 36
PE 12
Pen 3,12
Pen colour 16
PENDOWN(PD) 12
Pen type 17
PENUP(PU) 12
PO 24
POPS 24
POS 61
POTS 24
Primitives 22
PRINT(PR) 43
Procedure 20,22
– defining 21
– editing 78
– saving 25
Prompt 4,21

Quotes 47

RANDOM 56
RC 74
Read Cursor 77
READPICT 28
Recursion 50
REPEAT 9
RIGHT(RT) 4

SAVE 25,26
Saving – pictures 27
– procedures 24,25
SAVEPICT 27,28
Screen mode 16
Screen turtle 3
SETBG 16
SETH 61
SETMODE 16,80
SETNIB 13

SETPC 16
SETPOS 60
SETPT 17
SETSH 58
SETX 61
SETY 61
Short forms 4
SHOWTURTLE(ST) 13
Spinning squares 10
STOP 51

TELL 54
Title line 21
Trail 3
TS 24
Turtles 3
– floor 3
– multiple 54
– screen 3

Variable 40,47

Window mode 15
WINDOW 15
Word 25,26,66,67,69
WORD 70
Workspace 24
WRAP 15
Wrapmode 15
Write cursor 77

XPOS 61

YPOS 61