building with logo

on the commodore 64

creative use of the logo language

boris allan

BORIS ALLAN BUILDING WITH LOGO — C64 SUNSHINE

building with logo

on the commodore 64

creative use of the logo language

boris allan

First published 1984 by:
Sunshine Books (an imprint of Scot Press Ltd.)
12–13 Little Newport Street
London WC2H 7PP

Copyright © Boris Allan, 1984

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording and/or otherwise, without the prior written permission of the Publishers.

British Library Cataloguing in Publication Data
Allan, Boris
Building with Logo on the Commodore 64.
1. Commodore 64 (Computer) — Programming
2. LOGO (Computer program language)
I. Title
001.64'24 QA76.8.C64

ISBN 0–946408–48–3

Cover design by Grad Graphic Design Ltd.
Cover illustration by Richard Dunn.
Typeset and printed in England by Commercial Colour Press, London E7.

CONTENTS

CONTENTS

Contents in detail

Program Notes

Although the presentation of program listings is explained in the text, here are three points of particular importance.

1. The words you read in listings are of two varieties: there are the words you type, and there are the words and symbols which LOGO produces. All words and symbols produced by LOGO are shown in italics, whereas everything you enter is shown in normal type. Thus, in the lines

This
That

you know that LOGO produces 'This' and you have to enter 'That'.

2. Sometimes a line of words and symbols runs off the end of the screen and on to the next line. In LOGO the symbol for running over is an exclamation sign '!': if we have to run on to the next line in this book, we show this by indenting the next line:

This is a very long line which carries on, and extends over to the next line,
    and the next line starts in from the margin by a considerable
    extent.

This indentation does not necessarily match the picture on screen. In the program listings, this indentation will be about six capital letters. Some program lines are indented by about three capital letters.

3. When parameters (that is, variable quantities) are given for a procedure, some parameters are shown in parentheses. Parentheses are used to distinguish distinct parameters. There is a difference between the pair of parameters (5) (−3), and the single parameter 5 −3 (which is equal to the value 2). If you enter the sequence (−3), when you have LOGO print it out (in a listing, for example) you may be surprised that (−3) will appear as (− 3). On some occasions, LOGO may not accept the righthand parenthesis unless it is preceded by a space. It is probably safer always to include spaces with parentheses.

Introduction

This book is in two main parts: in the first part there is an introduction to LOGO, and to LOGO programming in general; and in the second part I examine specific applications from a wide variety of areas.

Learning LOGO

In Part 1, I cover mainly turtle graphics and the use of sprites, but during the process of the explanations I also cover topics such as list processing. I don't spend a great deal of time repeating examples on spirals, or whatever, which are common fodder in most books on LOGO, partly because they are admirably covered in the Commodore 64 LOGO Manual (for example, the appendix on graphics projects in the manual).

For those who may later wish to try spirals, because it seems a standard way to explain about LOGO, here are the two procedures with which to play (note that run-on program lines are indented about six characters):

```
TO INWARD.SPIRAL :ANGLE :SIDE :ANGLE.INC
    FD :SIDE RT :ANGLE
    INWARD.SPIRAL :ANGLE + :ANGLE.INC :SIDE
        :ANGLE.INC
END
TO OUTWARD.SPIRAL :ANGLE :SIDE :SIDE.INC
    FD :SIDE RT :ANGLE
    OUTWARD.SPIRAL :ANGLE :SIDE + :SIDE.INC :SIDE.INC
END
```

I do not make a big thing of list processing as a separate topic because, in fact, list processing is used throughout the book. Here, list processing is used in the way it should be used, as a very powerful tool for doing things — not as some mystical aim in itself. For example, the 'simultaneous' control of sprites requires extensive use of list processing to facilitate that control.

As the computer about which I am writing is the Commodore 64, which has sprites and a high resolution graphics which is difficult to use with any other language, I have accentuated the use of sprites and graphics, because

they are a marvellous medium for learning. But you can adapt what you learn here to your own purposes. Whilst on the subject of sprites, I must thank Marlene Kliman, who was responsible for the production of the sprite procedures provided with the LOGO utility disk. Marlene strongly believes that the use of sprites is the most important part of C64 LOGO (she may be biased), and I am grateful for the chance to talk things over with her.

Doing LOGO

Part 2 contains a wide variety of ideas of things to do with LOGO, some are useful (statistics or as a teaching aid for the handicapped), some are fun (controlling the keyboard or drawing a Greek letter), and one is very difficult (differential geometry). The chapter I tried to write on music convinced me that I was a musical failure, and the small amount I managed was more than adequately covered by the manual. I gave up, not because the task was impossible, but because something did not gel.

Most of these applications have not been examined in popular books before (in fact, I think I am the first with statistics and LOGO). Apart from being (in some cases) substantial applications, these allow an examination of LOGO features under a wide variety of headings. If you consult the index you will see how wide the range covered here is, including a vast number of figures — all drawn using LOGO, and dumped on to a printer. (I would like to thank Micro Simplex Group, Macclesfield for the loan of a printer for the purposes of this book.)

Further LOGO

For those who wish to read more widely in LOGO, the best book is *Mindstorms* by Seymour Papert (Harvester Press). There are two other useful books you should know about, both by me — the *Pocket Guide to LOGO* (Pitman Books) and *Introducing LOGO* (Granada Books).

For those who would like to find out more about their national LOGO group (if one exists — there are groups in a fair number of countries), if they write to me c/o Sunshine Books, I will try to place them in touch with fellow LOGOers.

Turtle well.

This book is undedicated to my VIC 1540 disk drive, may it rot in hell.

PART 1
Introduction to LOGO

CHAPTER 1
Starting Out with LOGO

Are you sitting comfortably? Then I'll begin.
Storyteller, Listen with Mother, BBC Home Service

This introductory chapter is to help you to learn quickly how to use the powerful language LOGO: all you need to follow this chapter is the disk which accompanies C64 LOGO.

If someone else has loaded LOGO into the computer for you, or if you know how to perform this operation, then skip the following section, and move to the section later in this chapter entitled 'Finding the turtle'.

Loading LOGO

Important

Do *not* switch on the C64 before switching on the disk drive.

Start by making sure that the C64 and the disk drive are both switched off. Commodore strongly advise you to switch on the disk drive *before* switching on the C64. On switching on the C64, you should see the standard C64 greeting. If you do not see this message, switch off both devices, and then try again.

When the disk drive is first switched on, the green light is activated, and the red light comes on for a short while. At the same time as the red indicator lights up, there is a whirring noise, so check that the whirring happens for your disk drive. If nothing happens then there is probably an error with the disk drive.

When the C64 is switched on, the whirring occurs again for the disk drive, and its red light comes on for a short while; the screen goes blank, and then the standard greeting message appears.

If the disk drive does not respond to the C64 being switched on, then something is wrong with the connection between the C64 and the disk drive. Either there may be something amiss with the cable connection, or

there may be something wrong with the routines to control disk operation (technically, the disk interface).

If either of these whirrings does not happen, then diagnosis and repair is probably necessary — though check first in the disk user's manual that the drive is set up correctly.

Take the disk labelled 'LOGO', and not the other disk (which is labelled 'LOGO Utility Disk'): insert the LOGO disk into the disk drive. The disk should have the label facing upwards, and towards you (the word 'LOGO' appears upside down).

Push the disk gently into the slot until you feel some resistance — which leaves about a centimetre and a half protruding. Then push the disk in firmly (but not energetically) until the disk catches, and lock the drive door by pressing the catch downwards.

The disk now in the drive contains a copy of the LOGO language system for the C64. The next task is to take the information about LOGO stored on the disk, and to copy that information into the C64's elephant-like memory.

The copying is performed in two stages. The first stage loads a BASIC program into memory: in the second stage, the BASIC program is run, and it is this program which copies the LOGO language system into the C64's memory.

The BASIC program we load into memory is called 'LOGO', and so we copy the program from disk by

LOAD "LOGO",8 [RETURN]

where the ',8' indicates that the program is to be loaded from disk. If the ',8' is omitted then *PRESS PLAY ON TAPE* appears on the screen, and the STOP key has to be pressed. [RETURN] at the end of the sequence is a reminder that the RETURN key has to be pressed.

If the above sequence has been entered correctly, there appears

SEARCHING FOR LOGO
LOADING
READY

and if

LIST [RETURN]

is entered, a short program full of POKEs appears. The final line of the program listing contains

LOAD "LOGO.BIN",8,1

and this is how the LOGO language system is copied from the disk into memory.

If for some reason the above does not happen, check that you are using the correct disk (and that it is not damaged), that the disk door is closed, and that the green light is still on. If the program loading still does not work, try switching off the C64, and starting again. If there is still no result, seek expert advice.

To load the LOGO system, therefore, enter the command

RUN [RETURN]

and then watch the screen change to produce the heading

Loading, please wait...

with much whirring, and blinks of the red light. This display continues for quite a while, until the screen changes colour and

ok

appears. After a much shorter time there is

run

and almost immediately the

COMMODORE 64 LOGO

heading is shown (plus copyright declarations).

If at any time the disk loading dies, the best procedure is to switch off the C64, wait five seconds, and then switch on again. You then have to go through the sequence from 'LOAD "LOGO",8' — if there are problems seek advice.

As soon as the LOGO system is successfully loaded into memory, remove the disk from the drive. To remove the disk, first push in and up on the catch to the drive door. The disk should then pop out.

Finding the turtle

When C64 LOGO is ready for action, we read on the screen

WELCOME TO LOGO!
?

(Note that, throughout this book, what appears on the screen is in italics,

and the information you are to enter is in normal type.) Next to the query (that is '?') there is a flashing cursor. Try the effect of typing in the word SHOWTURTLE, that is,

?SHOWTURTLE [RETURN]

and to this input there are probably two main classes of response by the C64.

The first possible response comes if you make a mistake:

THERE IS NO PROCEDURE NAMED ?SHOWTURTLE
?

which means that you entered ?SHOWTURTLE instead of SHOWTURTLE. If at any time you enter something incorrectly (such as SHOWTURLTE), LOGO will come back to tell you that it does not recognise what it is you have typed. Usually it will tell you that *THERE IS NO PROCEDURE NAMED* ***** where ***** will be replaced by whatever it is you entered incorrectly.

SHOWTURTLE is a LOGO 'command', that is, a built-in LOGO instruction to perform some sequence of operations. Later, you will come across the term 'procedure'. A procedure is either a LOGO command which can be printed out (ie you can list the sequence of events on the screen) or your own sequence of instructions which you can write and then use like commands.

In LOGO, a command or procedure does something, and so the second likely response, if you enter the name correctly, is that LOGO does something. The screen changes to produce a display where the upper three quarters of the screen have changed to a murky grey colour. This is the 'graphics screen', and the query sign is now in the lower segment, together with the cursor. Near the centre of the complete screen is a shape — this shape represents the turtle.

Examine the turtle carefully. It is a triangle, with a thicker line at one end (at present, the side facing downwards). The thick line indicates the back of the turtle, and the opposite point indicates the direction in which the turtle is 'facing'.

Enter

?FULLSCREEN [RETURN]

and the words you have typed in disappear, and the whole screen turns the same colour (murky grey) as the upper segment. You cannot see anything you now type.

How do you get back to the situation where you can see what you type?

You press the f3 (function) key on the righthand side and, lo, the words reappear. Press f5, and you have no words. Press f1 and the turtle disappears, only words remain.

There are three main ways of combining the turtle and text (that is, words). TEXTSCREEN (or key f1) gives the whole display over to text; SPLITSCREEN (or key f3) splits the screen between text and turtle; and FULLSCREEN (or key f5) gives the whole display over to the turtle.

Trying turtle training

So far we have had text alone, turtle and text, and turtle alone: the turtle has done nothing of note, apart from appear and disappear.

If you now enter

```
?RIGHT 30 [RETURN]
```

the turtle tilts slightly, so that the righthand side is vertical. The command RIGHT 30 has obviously made the turtle turn to the right (by 30 units or degrees), and the amount it has turned is given by the number which follows the instruction/command RIGHT. The command RIGHT does something, and it is another example of a built-in LOGO procedure (or command).

After the above command, try these two commands one after the other (note that in SPLITSCREEN the typing appears in the lower text screen)

```
?RIGHT 30 [RETURN]
?RIGHT 30 [RETURN]
```

and, after the first of the two commands, the side which started out on the left is now horizontal. After the second of the two commands, the back of the turtle (the thicker line) is aligned vertically, and the turtle points directly to the right. Now enter

```
?LEFT 90 [RETURN]
```

and the turtle points directly upwards, as at the start. This succession of commands, and the results of these commands on the turtle's orientation, are shown in **Figure 1.1.**

Figure 1.1: The Turtle's Orientation.

The command to turn left, through 90 units of angle, exactly counterbalances three commands to turn right through 30 units, and the commands RIGHT and LEFT are important, as is the command FORWARD, so try:

```
?FORWARD 75
?RIGHT 90
?FORWARD 75
```

(see **Figure 1.2**). (Note that I have stopped the reminder [RETURN] at the end of each line, as it is assumed to be at the end of every line entered.) The turtle has moved forward across the screen (upwards) for 75 units of distance, turned right through 90 degrees (which are the angular units used in LOGO), and then moved forward 75 units to the right, horizontally.

Continue the turtle's travels with

```
?RIGHT 90 FORWARD 75 RIGHT 90 FORWARD 75 RIGHT 90
```

The turtle draws the shape of a square, and finishes by pointing in the same direction (that is, directly upwards) as at the start.

Figure 1.2: The Turtle's Movement.

It would be quicker to draw the square if the commands FORWARD, RIGHT and LEFT were shorter, and if more commands were able to fit on one line, so try

```
?DRAW
```

which clears the square from the screen, and then

```
?FD 75 RT 90 FD 75 RT 90 FD 75 RT 90 FD 75 RT 90
?LT 135
?FD 100 LT 90 FD 100 LT 90 FD 100 LT 90 FD 100 LT 90
```

The first of the three lines contains a series of commands to draw exactly the same square as before (FD is FORWARD and RT is RIGHT); and the second line instructs the turtle to turn to the left through 135 degrees.

The third line has a surprising result. A square is drawn again, but part of the second square disappears into the area where the text is at the moment, and a little corner of a square appears at the top of the graphics screen. Press f5 (or type FULLSCREEN) and it can be seen that the tilted square has a corner which disappears at the bottom **(Figure 1.3)**. It is the same corner which appears at the top of the screen.

Enter

```
?DRAW
?RT 25 FD 999
```

and the screen clears, the turtle turns, and a line is drawn. This effect is shown in **Figure 1.4.**

Only one line is drawn, but the line disappears at the top (point A) to reappear at the bottom (also labelled A), and continues to disappear at the top and to reappear at the bottom. Trying different angles to turn, and different distances forward, can produce interesting effects.

This type of effect is called WRAPround, and can be contrasted with NOWRAPround. To see the difference, enter

```
?DRAW
?NOWRAP
?RT 25 FD 999
```

at which the turtle turns, but does not draw. You read in the text space

```
TURTLE OUT OF BOUNDS
?
```

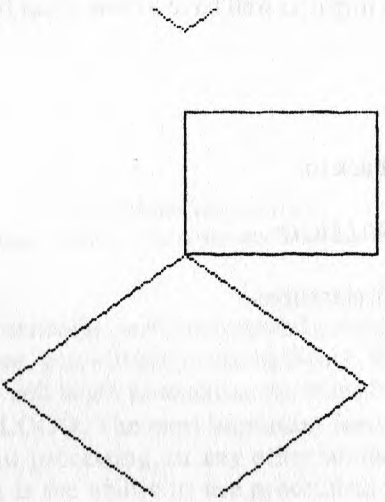


Figure 1.3

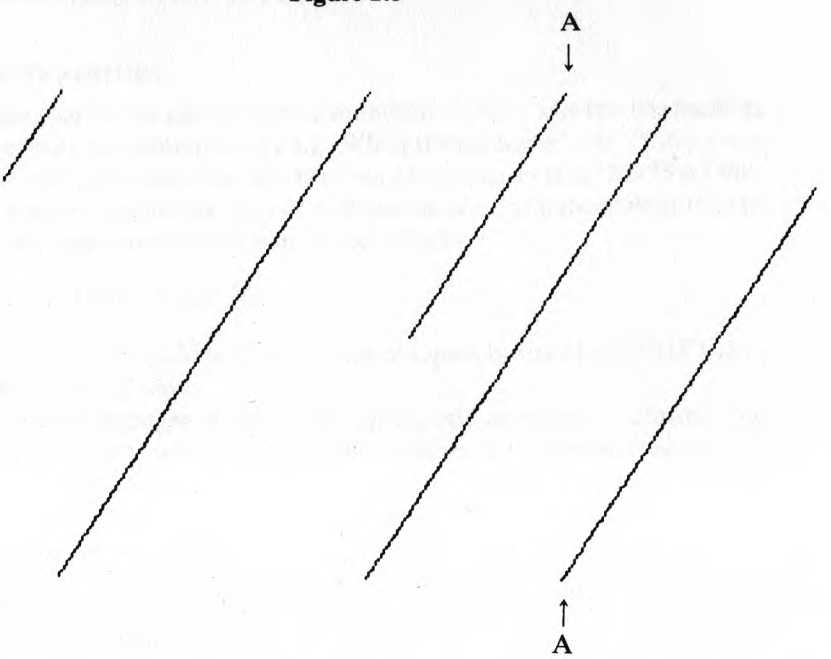


Figure 1.4

which means that, when in NOWRAP mode, the turtle cannot start to draw if the resulting line would end beyond the bounds of the screen.

For the whole of the next chapter we will be in the (normal) WRAP mode. Thus, as we might as well have a clean sheet for our next investigations, we enter

?GOODBYE

and that brings us back to

COMMODORE 64 LOGO

and the copyright declarations.

CHAPTER 2

Proceeding with LOGO

Little by little does the trick.
Aesop, Fables, The Crow and the Pitcher

So far, what we have needed to perform required quite a deal of typing, and if you are as able as me, you will keep making errors. Worry not.

In this chapter we will begin to examine the most important and most powerful feature of LOGO. The most important feature of LOGO is not turtle graphics, or list processing, or any other similar facilities the language provides — it is the ability to use procedures and, by the use of procedures, the ability to create systems of great power and simplicity.

We will start with a somewhat less grandiose aim — to reduce the amount of typing we have to perform.

Tearaway turtles

It is remarkably simple to draw a square in LOGO, but the instructions require quite an amount of typing. When the square of side 75 units was drawn (in the previous chapter) there were four examples of 'FD 75 RT 90'.

We need only write this pair of commands once, and then tell LOGO to repeat the commands four times. Type in the line

```
?REPEAT 4 [FD 75 RT 90]
```

where the square brackets '[' and ']' are obtained by use of the SHIFT key, plus the ':' and ';' keys.

The screen changes to show the turtle, and a square is drawn. The portion between '[' and ']' is repeated four times, and this repeated portion is called a 'list'. A list is shown by the square brackets '[ ]', and in the list '[FD 75 RT 90]' there are four items or elements.

Here is a similar pattern

```
?DRAW
?REPEAT 360 [FD 1 RT 1]
```

which produces an egg shape on the righthand side of the screen (it is

supposed to be a circle). Keeping the egg on the screen, next try entering

```
?REPEAT 360 [FD -1 RT 1]
```

and the turtle goes backwards, to draw another circle — this time on the left of the screen. It takes a long time for the turtle to drag itself round the circle, so we next attempt

```
?DRAW
?HIDETURTLE
?REPEAT 360 [FD 1 RT 1]
```

The turtle disappears, and the circle is drawn more quickly (as the turtle itself does not need to be constantly redrawn). The circle is not really a circle, is it? The circle is really a 360-sided regular shape (a polygon), which ends up looking just like a circle (see **Figure 2.1**).

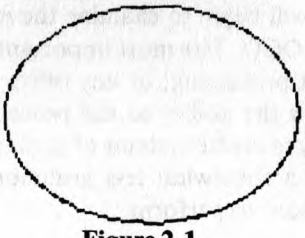


Figure 2.1

Why not try a polygon with twenty sides? If the new polygon has 20 sides, this means that in the 'circular' polygon there are 18 times as many sides as there are in the new one, because 360/20 = 18. Thus, for a polygon with 20 sides which looks almost the same size as the first 'circle', each individual side has to be 18 units long, and not one unit, and each turn has to be through 18 degrees.

So to draw a polygon of 20 sides, we enter:

```
?DRAW
?REPEAT 20 [FD 18 RT 18]
```

to draw what still looks very much like a circle, though this circle has only 20 sides. The time taken to draw the circle is very much faster because there is less calculation involved.

Note how the turtle has reappeared: every time the command DRAW is used, it is as if ST (short for SHOWTURTLE) has been called. The shortened form for HIDETURTLE is HT, so leave the 20-sided polygon on the screen and now enter

```
?HT REPEAT 20 [BACK 18 RT 18]
```

and the (almost) circle is drawn on the left, without the distracting turtle wallowing around (see **Figure 2.2**). The turtle travels backwards to draw the circle, because BACK 18 is the same as FD −18 (and is the same as the shortened form BK 18).

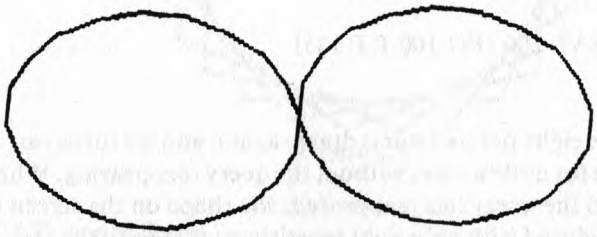


Figure 2.2

A star performer

Here is another shape to try to draw

```
?DRAW REPEAT 5 [FD 100 RT 144]
```

This is a sequence of instructions which produces a star with five points (a 'pentacle', **Figure 2.3**). A star with eight points (**Figure 2.4**) can be produced by entering the following sequence:

```
?DRAW REPEAT 8 [FD 100 RT 135]
```

Now to examine what it is we have performed.

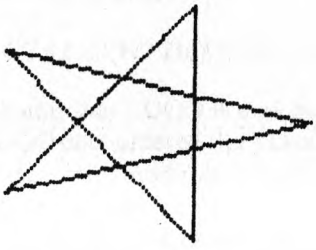


Figure 2.3

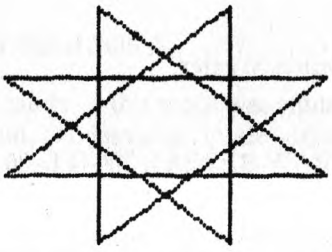


Figure 2.4

What has happened is that, each time, a list of instructions has been repeated: the main difference between the instructions in the two lists is

that the angle turned has changed. It is true that the number of times the list has been repeated has varied, but that is immaterial for the moment. (Note that 5*144 = 720, and 8*135 = 1440, and both 720 and 1440 are multiples of 360. Why is this so, and why are they different multiples?)

To enter now

```
?DRAW REPEAT 200 [FD 100 RT 135]
```

is to find that the eight-pointed star is drawn again, and the turtle carries on round the shape for quite a time, without the query reappearing. When the turtle stops, and the query has reappeared, the shape on the screen is the same as that produced with only eight repetitions: the turtle has carried on around the shape 25 times (because 200/8 = 25). The shape produced by the list has not altered.

If the previous line is altered slightly to

```
?DRAW HT REPEAT 200 [FD 100 RT 135]
```

then, once the star has been drawn, nothing seems to happen until the query appears. If the line is entered yet again, the repetitions can be stopped by holding down the CTRL key at the same time as the G key is depressed (known as CTRL-G). The drawing is stopped, and LOGO actually says

```
STOPPED!
?
```

If you now enter

```
?DRAW REPEAT 200 [FD 80 RT 75]
```

a complex pattern appears (check with **Figure 2.5**), and this indicates that what might appear complex on the surface is in fact very simple. The complex pattern of Figure 2.5 is really a many-pointed star, for all that has happened is that the sequence 'a move forward and a turn right' has been repeated many times.

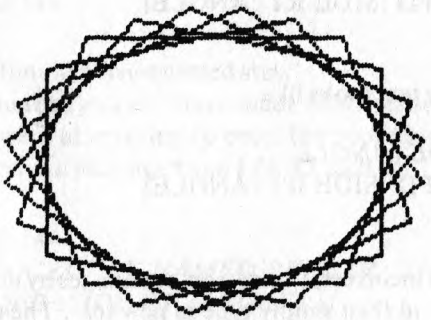


Figure 2.5

A STAR procedure

It is possible to study the effects of using different lengths and different angles by typing in variants of the above instructions, but as LOGO is designed to try to make things as uncomplicated as possible there is a simpler way. First of all, type in the following line, and type it in very carefully. For example, do not separate the ':' from either of the two words, for it should be ':SIDE' and ':ANGLE' and not ': SIDE' ': ANGLE' :

?TO STAR :SIDE :ANGLE

and the world erupts. Well ... at least the screen flashes, and the background to the text becomes murky grey, with

TO STAR :SIDE :ANGLE

appearing at the top left of the screen.

At the bottom of the murky grey screen there is a white strip, on which there are the grey letters

EDIT : CTRL-C TO DEFINE, CTRL-G TO ABORT

This indicates that LOGO is now in 'edit' mode — this mode was entered when LOGO encountered the TO command. (To leave edit mode without taking note of any modifications (that is, to 'abort') we do the same as we did to stop a sequence of LOGO actions, we use CTRL and G (that is, CTRL-G). To leave edit mode, but taking account of any modifications we have made (that is, to 'define'), we use CTRL-C, or the alternative of RUN/STOP).

The flashing cursor is now on the second line, so type in the following (note that there is no query in edit mode):

```
REPEAT 200 [FD :SIDE RT :ANGLE]
END
```

so that the entire text looks like

```
TO STAR :SIDE :ANGLE
REPEAT 200 [FD :SIDE RT :ANGLE]
END
```

and, if the text is incorrect, you can make changes by use of the cursors and the delete key, and then simply type in new text. The editing facilities are rather more powerful, and complex, than this would make it appear: it is surprising, however, just how much you can do with the bare essentials.

Though it is not necessary at the moment, at some time you should read the explanation of more complex editing facilities in the Appendix of the manual *LOGO : A language for learning* (henceforth the manual will be known as *LLL*). The basic details of how to use the editor are given in the Graphics chapter of *LLL*, and complete details in the Appendix.

When the text appears to be correct, enter the definition of the procedure STAR by use of CTRL-C, or RUN/STOP. LOGO's reaction to this action on our part is to respond

```
PLEASE WAIT...
STAR DEFINED
?
```

What we have produced is our own command STAR, a command which will perform what we have directed by means of the definition we have entered. As we saw in the previous chapter, the commands that we produce (that is, the commands we define) are called procedures.

Thus we have defined a procedure called STAR, so when we enter

```
?PO STAR
TO STAR :SIDE :ANGLE
REPEAT 200 [FD :SIDE RT :ANGLE]
END
```

we PrintOut the definition (or content) of the procedure we named STAR.

For STAR, the first value which follows the word STAR gives the distance to be moved, because (in the definition of STAR) the name given to the first value (ie :SIDE) appears in the pair FD :SIDE. The second name (ie :ANGLE) gives the angle to be turned, because :ANGLE appears in the pair RT :ANGLE. To clarify matters slightly, try to use STAR, for example:

```
?DRAW STAR 100 144
```

which results in the familiar five-pointed star.

It is almost certain that you will have made mistakes somewhere along the line whilst you were attempting to enter the procedure STAR. Quite possibly you will receive a message from LOGO such as:

```
?DRAW STAR 100 144
THERE IS NO PROCEDURE NAMED REPEAR, IN LINE
    REPEAR 200 [FD :SIDE RT :ANGLE]
AT LEVEL 1 OF STAR.
?
```

which is simply LOGO trying to be as helpful as possible. What this message means is that LOGO does not understand what the procedure REPEAR is supposed to perform (ignore the question of what LEVEL 1 is). This lack of knowledge on the part of LOGO is not surprising, because REPEAR should be REPEAT.

In the line REPEAR 200 [FD :SIDE RT :ANGLE], there is a mistyped word. To change the definition of STAR to the correct version, there are two possibilities: either enter

```
?TO STAR
```

or

```
?EDIT STAR
```

and in either case the result is that you are placed in edit mode to allow REPEAR to be changed to REPEAT. (It is worth making some deliberate mistakes, just to see what happens.)

Another likely error is one such as

```
?DRAW STAR 100 144
THERE IS NO PROCEDURE NAMED SIDE, IN LINE
    REPEAT 200 [FD:SIDE RT :ANGLE]
AT LEVEL 1 OF STAR.
?
```

and this has occurred because there is no space between the FD and :SIDE. The colon ':' is a special symbol in LOGO: to see what happens when the sequence FD:SIDE is encountered, try:

```
?FD:500
THE : IS OUT OF PLACE AT FD
?
```

and thus it can be seen that, in some way, the colon seems to be out of place at the end of FD. LOGO does not think that the colon starts SIDE, it thinks it ends FD — and it should not end a word.

Of course if you enter, incorrectly

```
?STAR100 144
```

you get

```
THERE IS NO PROCEDURE NAMED STAR100
?
```

which is another reminder of the need for a space between items in LOGO. Many errors in LOGO are little more than the omission of spaces between items.

Inputs and parameters

Another common error takes the form

```
?STAR 100144
STAR NEEDS MORE INPUTS
?
```

which means that LOGO expected two numbers to follow STAR. In fact STAR expected two inputs, and only found one (ie 100144). A more technical term for an input is a 'parameter'. We mentioned parameters above under the description 'names', eg :SIDE.

The procedure named STAR has two parameters which follow the name of the procedure, that is, the objects named :SIDE and :ANGLE. In like manner, the LOGO command FORWARD has one parameter, the number which follows the name of the command, that is, the distance to be moved. For example, in the sequence FORWARD 90, the value of the parameter is 90. The commands RIGHT and LEFT also take one parameter, which is, for both these commands, the value of the angle to be turned.

The command REPEAT, also used in the procedure definition, is more complex than FD or RT, in that REPEAT (like STAR) has two parameters. For example, in the sequence 'REPEAT 360 [FD 1 RT 1]', the first parameter value is 360, so that the first parameter thus indicates the

number of times the following list is to be repeated. The value of the second parameter is [FD 1 RT 1], and so, in this case, the value of the parameter is a list of instructions, and this is known as a complex value (ie it is not a simple value such as a letter or number).

When the procedure

```
?STAR 100 144
```

is used, it works by effectively 'replacing' every occurrence of the parameter :SIDE with the value 100, during the time the procedure is functioning. Thus, where the definition reads FD :SIDE, LOGO will treat this as being the equivalent of FD 100. The parameter :ANGLE has the value 144, and so RT :ANGLE is treated as RT 144.

The value of 100 for :SIDE, used in the running of the procedure on this occasion, is localised to that procedure, and so, if there is another example of :SIDE (not in the procedure), the value of the other example of :SIDE is not affected. The scope of the operation of any parameter is said to be 'local' to the procedure.

With the flexibility given by the use of parameters by the STAR procedure, we can try all different kinds of combinations of numbers, where one very pretty pair of parameters is used in

```
?DRAW STAR 500 175
```

which, because of extensive wraparound, produces a delightful lace pattern. Using Sunshine's telephone number (01 437 4343) produces a result closer to the arrangement of nerve tissues (is this significant? — see **Figure 2.6**). The arrangement is thus

```
?DRAW STAR 437 4343
```

It is worth trying out other phone numbers, to see if the resulting picture reminds you of the relevant person: some people have a right mess of a telephone number.

Finally, given your experience of how easy it is to make mistakes, try to extend the procedure STAR to use three parameters, so that it is possible to choose the number of times for which the instructions in the list are to be repeated. The first line of the definition might be

```
?TO STAR :SIDE :ANGLE :TIMES
```

I am not going to give any more assistance, so, if you want to test yourself and LOGO — try it out.

Do not worry too greatly if you cannot get it right, because this procedure will be discussed in the next chapter, when we have a more detailed

look at what happens when you make errors in LOGO. Before we examine errors, we had better have a short detour, to make a preliminary examination of the ways in which we can save our handiwork on to disk.

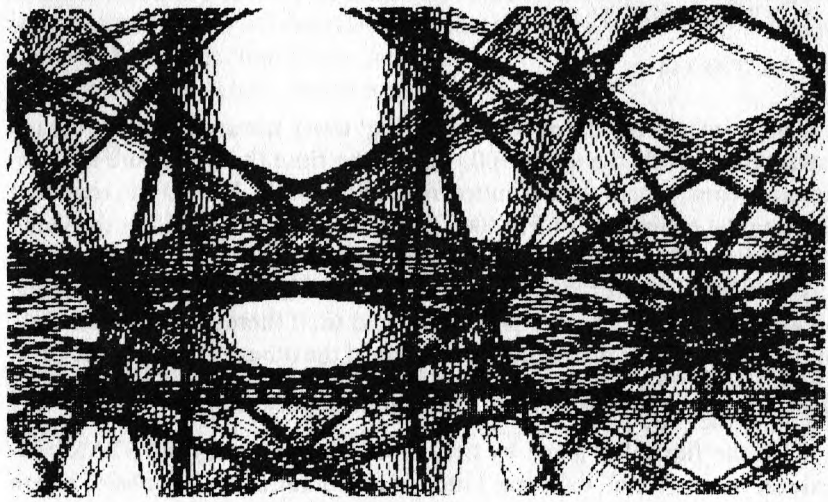


Figure 2.6

Saving information to disk

The appropriate information on elementary aspects of disk use is given in *LLL* (in the Beginning LOGO and Graphics chapters), and the use of disks is far simpler from within LOGO than it is from within BASIC.

Before it is possible to save any information on a disk, the disk has first to be prepared to accept information. Setting up the disk is known as 'formatting'. To format a disk in BASIC the sequence is the rather tedious

```
OPEN 15,8,15
PRINT#15, "N0:DISK NAME,01"
```

which can be explained as opening a channel in the first line, and then formatting a disk in the second line. The formatting uses the name DISK NAME for the disk, where you can choose to use any title you wish. It all seems far too much of a bother, and in any case it is all in BASIC, which seems somewhat pointless.

Friendly LOGO has an easier way: to replicate the BASIC instructions, without having to enter BASIC, this what you do...

1) Place a blank disk in the disk drive.

2) Choose a name to give to the disk, for example C64 LOGO A.
3) Enter

   ?DOS[N0:C64 LOGO A,01]

   and wait until the query returns. This is the means by which you format a disk in LOGO. End of story.
4) Enter

   ?CATALOG
   C64 LOGO A

   664 BLOCKS FREE.
   ?

   and you have formatted a disk, and then CATALOGed it. The CATALOG procedure tells you the name of the items you have stored on the disk, and all you have stored at the moment is the name of the disk, that is, C64 LOGO A.

This is the end of the sequence to format the disk.

Enter the LOGO command POTS to Print Out the TitleS of the procedures you have stored away, and you will probably discover a large number of procedures you did not know existed. These are procedures which have been entered and were incorrect due to some typing error. Now try the fairly explicit command PO NAMES, and you might find some names of objects you have forgotten existed.

The purpose of these enquiries is to discover just what there is in your LOGO workspace, so that when you try to SAVE the contents of your workspace you will have a check on what was there. To copy the contents of the workspace, the process is

?SAVE "DUMMY
?CATALOG
C64 LOGO A

1 DUMMY.LOGO PRG
663 BLOCKS FREE.
?

you know what there was to save (which will be different to what was in my workspace). (Details of the C64 LOGO commands for saving procedures and reading disks are contained in *LLL* in the Graphics chapter).

If you now say GOODBYE to LOGO, and ask

?POTS
?PO NAMES
?

there are no names or titles in your workspace. If you now enter

```
?READ "DUMMY
```

a list is produced of the procedures that were previously in your workspace.

One of the procedures should be given as STAR DEFINED. As the information is READ in from the disk, the procedures have to be defined again, as if you had been typing them in at the keyboard. It is worth trying out POTS and PO NAMES, to see if the objects have returned. Have they? . . .

CHAPTER 3

Getting Used to LOGO

Errors benefit us because
they lead us to study what happened,
to understand what went wrong, and,
through understanding,
to fix it.
Seymour Papert, Mindstorms

There is an old saying which goes: 'To err is human, to forgive, divine'.

Think, however, how often we are taught that mistakes are a bad thing, and that if one makes a mistake it should be obliterated, or ignored, and never studied. 'We learn from our mistakes' is another old saying whose consequences are often forgotten, and Seymour Papert's assertion, given at the beginning of this chapter, reminds us of that proverb.

Seymour Papert is probably the main influence on the development of LOGO, and his influence partly explains why LOGO is a computer language which accepts that errors will be made. What is as important is that LOGO is designed to help the user learn from those errors.

Learning from mistakes

If you are to learn from mistakes, then (so the designers of LOGO realised) the error information with which you are provided by LOGO needs to be as constructively helpful as possible. Sometimes the assistance given by LOGO is not as helpful as perhaps you (the user) would like, but LOGO's intention is always to assist the user as much as possible, given that program designers are only human.

The emphasis placed on giving constructive help is important in LOGO. It is hoped that, in time, the small errors you make will become fewer but, at the same time, it is hoped the topics you will tackle will become more demanding. It does not help in your development as a programmer if, when you make a mistake, there is nothing to point out the nature of the error. So LOGO does not encourage you to make mistakes, rather it tries to help you cope with the errors when they do arise — and they always will arise.

In LOGO the emphasis is on you and your development as a programmer, to do things you want to do: it runs counter to the LOGO ideal for you continually to have to refer to an expert to debug your programs. In LOGO, it is believed that you learn far more through your own efforts. This is why you should not consider either *Building with LOGO* or *LLL* as giving definitive answers because, in fact, both books try to encourage you to ask interesting questions — and then leave you to try to find the answers by your own efforts.

Providing answers and suggestions cannot be ignored completely, however. So, for example, in *LLL* (Beginning LOGO) you will find this important paragraph:

Throughout this tutorial, there are projects. Suggested answers to these projects are in the Appendix. REMEMBER that there are many ways to solve a problem and just because your method is different from ours does not mean it is wrong.

and this shows that the designers of C64 LOGO want you to experiment, probably to make mistakes, to learn from your errors, and to engage in the constructive process of debugging.

As Seymour Papert says (in *Mindstorms*) '. . . *everyone* learns from mistakes.' In LOGO the important question is not 'Is it correct?', but 'How can I fix it?' — though of course getting it right in the end is important.

Try, therefore, to get into a frame of mind where making mistakes is not as important as enjoying correcting the mistakes, and try to see debugging as a learning exercise. Learning to debug is an education in learning to solve problems, and problem-solving skills are highly prized these days: one reason why these skills are so rare is that most people have been taught to fear failure and they feel guilty if they make mistakes. This, taken too far, stifles innovation and creativity.

Exploring bugs

LOGO is a friendly language and, if it cannot understand what you have done, it will tell you so.

If you are using LOGO, then, when you type something into the C64, for all the computer is concerned it could be RUBBISH; so try to enter RUBBISH:

```
?RUBBISH
THERE IS NO PROCEDURE NAMED RUBBISH
?
```

and now you know. What has happened is that the LOGO system (or a part

of the system known as the 'interpreter') knows that you have typed in a sequence of characters, and the characters form the word RUBBISH. The interpreter keeps a list of words that LOGO knows, where a word is a sequence of characters. Against each of the words known by LOGO there is a description of what is supposed to happen when that word is used. RUBBISH is not a word LOGO knows how to use.

Start by entering the following:

```
?GOODBYE
```

at which the screen clears and the copyright declarations appear. Now investigate what happens when you enter the sequence:

```
?.CONTENTS
RESULT: [STARTUP FALSE TRUE]
?RUBBISH
THERE IS NO PROCEDURE NAMED RUBBISH
?.CONTENTS
RESULT: [RUBBISH STARTUP FALSE TRUE]
?
```

The LOGO command .CONTENTS prints the list of all the words known to LOGO, apart from all the 'primitive' commands known by LOGO (see *LLL*, Appendix). A primitive command is one such as FORWARD (or FD) which is available when LOGO is switched on.

When LOGO starts, there are three special LOGO words, apart from the primitives, which the system loads ready for use. These are the three words revealed by the first use of .CONTENTS.

STARTUP: This is a special C64 LOGO word which is used for making 'self-starting' files on disk (see *LLL*, Appendix), where such files are used for automatic execution of procedures. This will be discussed later, when we examine disk use in more detail.

FALSE: This a LOGO word which is what it says. It represents something being false. For example:

```
?2 = 3
RESULT: FALSE
?
```

which is obviously correct, because '2 is equal to 3' is false.

TRUE: Another self-explanatory word:

```
?2 = 2
RESULT: TRUE
?
```

and it is true that '2 is equal to 2'.

It is when we enter some rubbish that LOGO learns some more words.

When RUBBISH is typed in, the LOGO interpreter cannot find the word in its list. Because the word has now been used (even though purely by accident), LOGO still adds RUBBISH to its list of words together with the information 'purpose unknown'. Thus with the second use of .CONTENTS we find that RUBBISH is there, but that LOGO still will not be able to use RUBBISH in any way — for how can LOGO know what to do unless you tell it?

Thus, to enter the word RUBBISH (without any explanation of what LOGO is to do with the word) is a mistake, it is a bug in your programming, yet even so LOGO tries to explain to you why it is refusing to do anything. You are using a very benevolent language.

Type in

```
?FD 50 FD50
```

and a line is drawn on the screen, and then there is the ubiquitous LOGO reminder that there is no procedure with the name FD50. Ask for the list of contents and you will find

```
THERE IS NO PROCEDURE NAMED FD50
?CONTENTS
RESULT: [FD50 RUBBISH STARTUP FALSE TRUE]
?
```

and the word FD50 has joined the list.

After a lengthy LOGO session, it is worth using .CONTENTS just to see how many typing errors you have made. The difference between FD 50 and FD50 is of course the space between FD and 50: the use of spaces to separate items in LOGO is of crucial importance, which is why you cannot have spaces in words.

LOGO recognises the word FD as being in its list of primitives, and finds against the name FD the information 'move turtle forward the number of units given by the following value'. (Exactly the same information appears against the expanded form of FD, that is, FORWARD.) The value following FD is the number 50, and so the turtle moves forward by that amount.

Note that LOGO does not like

```
?FD FIFTY
THERE IS NO PROCEDURE NAMED FIFTY
?FD "FIFTY
FD DOESN'T LIKE FIFTY AS INPUT
?FD :FIFTY
THERE IS NO NAME FIFTY
?
```

and the reasons for the differences in these error messages will become clearer as we progress through this book, and especially in Chapter 8.

What happens then, when no value follows the word FD? Find out:

```
?FD
FD NEEDS MORE INPUTS
?
```

It is clear that LOGO expects FD to have rather more values to follow. LOGO uses the term 'inputs' in place of the rather more unwieldy 'values to follow', thus for FD the LOGO interpreter expected one input, and found none.

If you did not try to write a new procedure

```
?TO STAR :SIDE :ANGLE :TIMES
```

after the end of the last chapter, now is the time to try. Cowardice will not save you — unless you try to do things, you will not learn. Do not forget that you should not worry about making mistakes, for you can and should learn from those mistakes.

If you managed without making mistakes, I am truly surprised, because when I started LOGO I was for ever getting it wrong.

Names, objects and procedures

The task was, you may remember, to extend the procedure STAR to use three parameters, so that it was possible to choose the number of times for which the instructions in the list were repeated. The list of instructions was [FD :SIDE RT :ANGLE], and the list was prefixed by the items REPEAT 200, where 200 was the number of times the list was REPEATed.

All we have to do, therefore, is to modify the definition of STAR, with much changing of screens to murky grey, and so we alter the procedure definition to read

```
TO STAR :SIDE :ANGLE :TIMES
    REPEAT TIMES [FD :SIDE RT :ANGLE]
END
```

and thus, to use this new form of STAR (to draw a square of side 100) we enter the command

```
?STAR 100 90 4
THERE IS NO PROCEDURE NAMED TIMES, IN LINE
    REPEAT TIMES [FD :SIDE RT :ANGLE]
AT LEVEL 1 OF STAR.
?
```

and so there must be something wrong with our definition of STAR.

To see why the new definition of STAR is incorrect, we need to investigate the difference between the names of objects, the values contained by objects, and the names of procedures. To start our investigation, enter the following words, exactly as given below (there is no need to enter the line numbers):

```
?"ME ; QUOTED WORD (Line 1)
RESULT: ME
?ME ; PROCEDURE CALL (Line 2)
THERE IS NO PROCEDURE NAMED ME
?:ME ; VALUE OF "ME (Line 3)
THERE IS NO NAME ME
?MAKE "ME "PROGRAMMER ; THE VALUE OF "ME IS
        "PROGRAMMER (Line 4)
?:ME ; VALUE OF "ME (Line 5)
RESULT: PROGRAMMER
?; THIS IS HOW YOU WRITE A COMMENT (Line 6)
?
```

because there are some very important distinctions which need to be learnt from these six lines.

First, however, use .CONTENTS to disclose what LOGO knows, where the result should be that LOGO knows at least [PROGRAMMER ME TIMES ANGLE SIDE STAR STARTUP FALSE TRUE] plus any other items you have entered by error, or which are left from earlier attempts. Take the above lines one at a time, so that we can discover about these important distinctions.

Line 1: The entry "ME is called a 'quoted' word because the word is preceded by the quote mark '"'. When a sequence of characters (without a

space) has the first character ", the characters which follow the quote mark are treated as a complete word or unit. A word is only a sequence of characters without a space, and thus to prefix with " is sufficient to clarify its status. Therefore to enter "ME means that LOGO treats the sequence as if it was the word ME, and so ME is the result.

Line 2: To enter ME by itself is to cause LOGO to consider the sequence as if it were the name of a procedure. Giving the name of a procedure to LOGO is how you instruct LOGO to perform the procedure, as we did above with STAR or with FORWARD. The process of telling LOGO to activate a procedure is also known as making a procedure 'call'. Thus, in essence, to give a sequence of characters to LOGO which is not preceded by a '"' or a ':' (see line 3) implies that there is a procedure of that name. There is no procedure named ME, which is exactly what LOGO tells us.

Line 3: Entering the sequence :ME is how you inform LOGO that you are talking about a 'value' of an object. The name of the object is ME, and the colon ':' is an instruction to LOGO to produce the value (the thing) contained in that object. At the moment, there is no object with the name ME, and thus ME cannot have a value. ME cannot have a value until it is given one, because no default value is assumed.

Line 4: This is how ME is given a value. The object with the name ME is made to contain the name PROGRAMMER. Both the words ME and PROGRAMMER have to be preceded by quote marks, so that LOGO knows that you are talking about the words and not procedures. The implications of this line will be examined in more detail in a short while, in the discussions on lines 8 – 10.

Line 5: As we have now given a value to ME, we can check to see what that value is. The result of the check is that the value is PROGRAMMER, which is as we would have hoped.

Line 6: This line merely exists to show that anything written after a semicolon is ignored in the execution of LOGO. The use of the semicolon is how comments are written in LOGO — although comments are quite often unnecessary with LOGO because the words you use can be self-explanatory.

	NAME	CONTENT
Procedure	"YY	YY
Object	"XX	:XX

The content of a procedure is an *action*.
The content of an object is a *value*.

Figure 3.1: Objects — their Names and Content.

If you examine **Figure 3.1**, you will see an attempt to illustrate this

phenomenon. First we start with a 'name', and a name can either refer to an 'object' or to a 'procedure'. The content of an object is its 'value', and the content of a procedure is its 'action'.

To refer to simply a name, you prefix the name by '"'. If the name refers to an object, to find the value of that object you prefix the name by ':' — if the name refers to a procedure, then the action of the procedure is found merely by typing in the name, without any prefix. You never refer to the object or the procedure directly, you can only refer to the name or to the content.

The next set of lines begin to examine ME in greater detail:

```
?:ME ; AS LINE 5 (Line 7)
RESULT: PROGRAMMER
?THING "ME ; ANOTHER WAY OF FINDING THE VALUE
        (Line 8)
RESULT: PROGRAMMER
?MAKE ME "PROGRAMMER ; ME IS UNQUOTED (Line 9)
THERE IS NO PROCEDURE NAMED ME
?MAKE "ME PROGRAMMER ; PROGRAMMER IS
        UNQUOTED (Line 10)
THERE IS NO PROCEDURE NAMED PROGRAMMER
?
```

and here is what we have found out about ME and PROGRAMMER.

Line 7: In this line we are merely repeating the action of line 5 (above), to produce the same result.

Line 8: This line explicitly asks for the THING contained in the object whose name is ME (that is, "ME). A line which was THING ME would be invalid, as is shown by the next line.

Line 9: The assignment can be compared with line 4, where the difference arises with the use of ME rather than "ME. LOGO encounters the MAKE command, and thus expects a name to which it can assign a value. After MAKE comes ME, and thus LOGO thinks that ME must be the name of a procedure (and the procedure could possibly provide a name). There is no procedure named ME, as LOGO so rightly informs you. If you enter THING ME you get exactly the same response, that is, there is no procedure with the name ME.

Line 10: For this line, the assignment is to a name (the name given by "ME), but what is assigned is not a value — LOGO assumes that PROGRAMMER is the name of a procedure, which may produce a value when it is executed. However, as we know only too well, there is no procedure with the name PROGRAMMER, and thus there is no value to assign to the object ME.

A new STAR

Right, we can now return to the problem with the STAR procedure, where the definition was

```
TO STAR :SIDE :ANGLE :TIMES
    REPEAT TIMES [FD :SIDE RT :ANGLE]
END
```

so that, when the procedure was called, the outcome of the activation was

```
?STAR 100 90 4
THERE IS NO PROCEDURE NAMED TIMES, IN LINE
    REPEAT TIMES [FD :SIDE RT :ANGLE]
AT LEVEL 1 OF STAR.
?
```

producing a LOGO error message which now becomes rather easier to understand. What has happened is that the list [FD :SIDE RT :ANGLE] has to be repeated an actual number of times, and thus LOGO expects a value to follow REPEAT.

Generally speaking, unless a number is explicitly given, a value is shown by a name preceded by a colon, as with :SIDE and :ANGLE. Though it is always possible for a procedure to produce a value, the procedure has at least to be recognised by LOGO: TIMES is not the name of any procedure known to LOGO, thus the error message.

It only requires a slight modification to produce a correct program

```
TO STAR :SIDE :ANGLE :TIMES
    REPEAT :TIMES [FD :SIDE RT :ANGLE]
END
```

which then operates correctly — for example,

```
?STAR 100 90 4
```

draws a square. The square has sides of 100 units, the angle through which the turn is made at the end of each side is 90 degrees, and there are four sides, so the drawing of a side is repeated four times.

There are many other errors you can make, but most of those are typing errors. For example, why do you think there is an error here?

```
?STAR 100 90 4
THERE IS NO NAME TMES, IN LINE
```

REPEAT :TMES [FD :SIDE RT :ANGLE]
AT LEVEL 1 OF STAR.
?

To help slightly, here is the content of the procedure, obtained by use of the PO command:

```
?PO STAR
TO STAR :SIDE :ANGLE :TIMES
    REPEAT :TMES [FD :SIDE RT :ANGLE]
END
?
```

Hint: Check the spelling of names. Finally:

```
?STAR 100 90 4
THERE IS NO NAME TIMES, IN LINE
    REPEAT :TIMES [FD :SIDE RT :ANGLE]
AT LEVEL 1 OF STAR.
?PO STAR
TO STAR :SIDE :ANGLE :TMES
    REPEAT :TIMES [FD :SIDE RT :ANGLE]
END
?
```

Hint? What hint?

CHAPTER 4
Turtles Alive with LOGO

All uncertainty is fruitful...
so long as it is accompanied by the wish to understand
Antonio Machado, Juan de Mairena

This chapter is an exercise in the use of procedures in LOGO, and the centre of attention is the turtle.

With the procedures given in this chapter, the turtle travels around on its own, going where it wishes, controlled (if that be the word) by the uncertain vagaries of a random number generator. The reason why the turtle is self-propelled, and is the epitome of uncertainty, is that we can then concentrate on the design of a more restricted range of facilities.

The most important ideas to follow in this chapter are the notions of 'sequence', 'repetition', and 'control', and the introduction of the notion of 'recursion' means that this chapter discusses one of the most powerful constructs available to programmers. If you can understand recursion, you are well on the way to understanding, for example, many of the ideas in artificial intelligence. Recursion implies the embedding of a procedure within a procedure of the same type, just as, in thought, ideas are embedded in ideas and, in language, clauses are embedded in clauses. In humans, intelligence is a great deal to do with the way in which thought and language are used. For a computer to understand even part of thought and language requires a language which can tackle embedding, that is, a language which can use recursive procedures: LOGO is such a language.

First, however, to rather more mundane matters.

A random look

Many games (computer and other) depend for their success on the notions of 'chance' and of 'randomness'. A whole discipline (statistics) and a field of application (probability) have grown around investigating whether things can or have happened by chance. So what is it to be random?

Here is a procedure named MOVE

```
TO MOVE
```

```
    FD 15
END
```

which, if entered as MOVE, sends the turtle forward a very small amount, indeed very inconsequential amount. To enter a succession of MOVEs is to make the turtle go further forward.

Next enter

```
?RANDOM 360
RESULT: 293
?RANDOM 360
RESULT: 111
?
```

where the results, when you enter RANDOM 360, will possibly differ from those I have given. RANDOM is a LOGO primitive which produces a number chosen 'at random' from the integer range 0 to :NUMBER−1

However, the number is not really chosen at random, but follows a fixed sequence. The numbers appear to be random because you do not know the sequence, and the sequence takes a long time to repeat.

Enter GOODBYE to restart the system, and then

```
?REPEAT 10 [PR RANDOM 360]
293
111
100
256
38
102
78
189
301
?
```

and we find that the numbers 293 and 111 are again the first two outputs: so to say GOODBYE and ask for the first 10 random numbers from 0 to 359 gives the same series of results.

A small modification has the effect of making the start of the sequence appear 'random' (that is, 'unpredictable'). Reset the system again, and then enter

```
?RANDOMIZE
?REPEAT 10 [PR RANDOM 360]
151
307
285
337
217
205
130
27
251
324
```

The new primitive, RANDOMIZE, starts the sequence at some unpredictable value, and thus the series as a whole then becomes unpredictable, or — in other words — the series becomes randomlike. The usual name for the procedure which produces such a sequence is a pseudo-random number generator. The pseudo-random number generator does not produce numbers which are truly random, but the results of its operation are effectively just as useful.

I have checked the appropriateness of the C64 LOGO random number generator by making 1000 calls to the procedure, and counting up the number of times each number 'appeared'. As it is a 'nice' number to choose, I used RANDOM 10 as my example call. This produces numbers from 0 to 9. The procedures that I designed to be used for this evaluation are given in Chapter 8, as they demonstrate some interesting programming techniques. Here are the results for four trials:

NUMBER OF OCCURRENCES

Value	Trial 1	Trial 2	Trial 3	Trial 4
0	102	115	112	95
1	110	108	96	97
2	115	107	90	99
3	97	90	98	106
4	73	95	96	108
5	97	94	109	94
6	105	82	104	91
7	100	97	106	114
8	93	107	90	102
9	108	105	99	94

Generally speaking, it would be difficult to find any patterning in these

results, and so we can be fairly confident that the random numbers produced by C64 LOGO are adequate to the purpose intended.

A wobbly walk

Now compare the effect of these two sets of commands on the operation of the turtle:

```
?RT 45 LT 45
?RT 45 RT 315
?
```

In the first line, the turtle turns right and then left, ending up facing in the direction in which it started. In the second line, the turtle turns right, and continues turning right until it faces directly upwards, as before. Effectively, therefore, the command LT 45 is the same as RT 315, even though the turtle turns in different directions (either the left or the right) to arrive at the same orientation. (The perceptive will have noted that 45 + 315 is 360, which is one complete revolution.)

Now enter the definition of MOVE again, because with all those GOODBYEs the previous definition was lost. Then try this sequence

```
?RANDOMIZE
?RT RANDOM 360 MOVE RT RANDOM 360 MOVE RT
        RANDOM 360 MOVE
?
```

and the turtle goes for a very short wobbly walk (also often called a random walk). The walk is so short that it is worth repeating the last line, and, rather than typing the line in again, we use a special LOGO facility.

This special facility is a means of repeating the last line: either you use CTRL-P or SHIFT plus the up/down cursor key. I prefer using the SHIFT and cursor key because they are next to each other on the right. The repeated line does not include an automatic carriage return, and so you have to depress the RETURN key after each repeat. Once the line is correct, depress f5 — to give the whole screen over to graphics — and then keep pressing SHIFT and the up/down cursor key, followed by RETURN, and the turtle will wobble across the screen.

It is possible that, after a time, you may come to the conclusion that there must be a simpler way. There usually is.

The MOVE procedure simplifies the movement of the turtle, so why not simplify the turning of the turtle? All we have to accomplish is the replacement of RT RANDOM 360 by one word — by a procedure to which we will give an apt name, so why not WOBBLE? Your task, therefore, is to

define a procedure to be called WOBBLE which will turn the turtle in random directions. You have to define the procedure, not me, and it is all up to you.

See if the procedure you have designed works, by trying the effect of the two lines.

```
?DRAW
?WOBBLE MOVE WOBBLE MOVE WOBBLE MOVE
?
```

where the second line uses your WOBBLE procedure. What should happen if all is well is that the turtle should wobble away in exactly the same way as before. By deft use of the repeat line facility (SHIFT plus up/down cursor key), it is possible to drive the turtle to distraction. We have not progressed very far, however, for we are still mechanically hitting keys.

If SHIFT plus up/down cursor key is a repeat facility, why not use REPEAT? Here goes:

```
TO SHORT.WALK
    REPEAT 200 [WOBBLE MOVE]
END
```

and to tell LOGO to go for a SHORT.WALK gets the turtle wobbling. However, the wobble comes to an end after a short while, and if we stop the wobble before it ends, by use of CTRL-G, we read

```
STOPPED!, IN LINE
    REPEAT 200 [WOBBLE MOVE]
AT LEVEL 1 OF SHORT.WALK.
?
```

but more often than not we want to continue beyond 200 movements of the turtle.

Further to travel

What we need to do is to travel (which is a wobble and a move), and then travel again (which is a wobble and a move), and then travel again (which is a wobble and a move), and then... What we need is a procedure TRAVEL, where to TRAVEL we WOBBLE MOVE and then TRAVEL. Here is how the procedure is defined:

```
TO TRAVEL
    WOBBLE MOVE TRAVEL
END
```

where, if we enter TRAVEL, the turtle goes on its travels for a wobbly random walk. The walk continues for as long as you have patience.

When you become impatient, you stop the turtle's travels by pressing CTRL-G, and are told something akin to

STOPPED!, IN LINE
WOBBLE MOVE TRAVEL
AT LEVEL 435 OF TRAVEL.

which in my case produced the picture shown in **Figure 4.1.** In another travel (that shown in **Figure 4.2**) the picture differed and the outcome of using CTRL-G was

STOPPED!, IN LINE
WOBBLE MOVE TRAVEL
AT LEVEL 1091 OF TRAVEL.

Note that the two levels are different, in that the first level is 435 and the second 1091. Note, in addition, that the number of lines drawn in the first case is much less than the number of lines shown in the second example—this is partly indicated by the fact that the second figure is darker.

There is now only the simple question of what the meaning of the term LEVEL is, which has been met before (as Level 1) and has reappeared in a new guise.

The procedures WOBBLE and MOVE, say, are effectively only the same as sequences of LOGO words and inputs. For example,

WOBBLE MOVE

is the same as

RT RANDOM 360 FD 15

(Note that I have now given the definition of WOBBLE, which I left you to work out earlier in this chapter). So, therefore, I will treat TRAVEL as if it were simply composed of other procedures

```
Level Action
0     TRAVEL
1     WOBBLE MOVE TRAVEL
2                 WOBBLE MOVE TRAVEL
3                             WOBBLE MOVE TRAVEL
4                                         WOBBLE MOVE TRAVEL
```

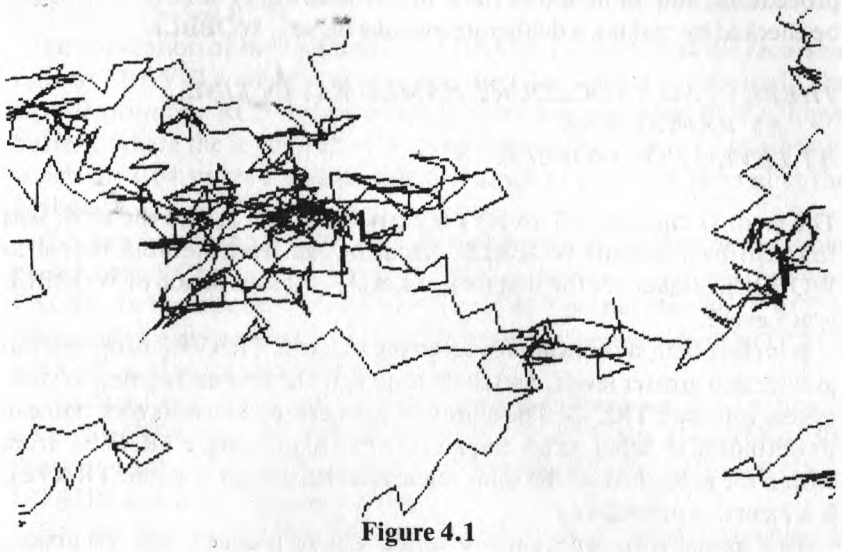


Figure 4.1

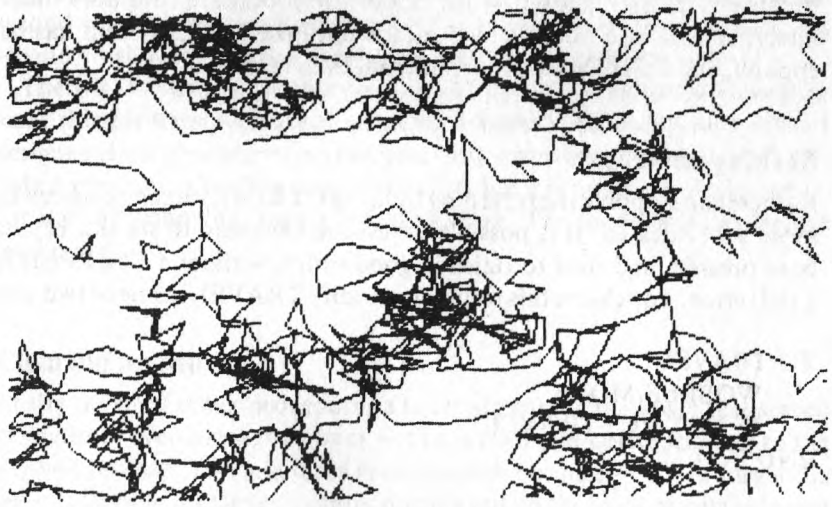


Figure 4.2

At Level 0 there is the first call to TRAVEL, and Level 0 is, in fact, the normal level at which you type in information to LOGO. When the procedure is called, its action occurs at one level on from the call. Thus, a call to TRAVEL at Level 0 produces an action within the procedure at Level 1. This is why, normally, if there is an error in a procedure, or the procedure is stopped by the user, the line reproduced is that at Level 1.

Within the action of the procedure at Level 1, there are other calls to

procedures, and the action of those procedures will be at Level 2. This can be checked by making a deliberate mistake in, say, WOBBLE

THERE IS NO PROCEDURE NAMED RY, IN LINE
RY RANDOM 360
AT LEVEL 2 OF WOBBLE.

The error (I changed RT to RY) occurred at Level 2, and the error was found in the procedure WOBBLE. As can be seen from the table, the call to WOBBLE is made for the first time at Level 1, thus the action of WOBBLE is at Level 2.

It is clear that, by continually referring to itself, TRAVEL progresses to greater and greater levels, and there is no way the process is going to stop, unless you use CTRL-G. The ability of a procedure to use its own name in its definition is called recursion, and the call to procedure TRAVEL from within the procedure of the same name is called a recursive call: TRAVEL is a recursive procedure.

The special form of recursion we have here is called 'tail' recursion, because — as you can see — the tail of the procedure continues indefinitely. In this case 'indefinitely' means until the LOGO system has had enough, and C64 LOGO is more tolerant than most.

Making choices

Rather than stopping the travelling by using CTRL-G, we can be somewhat more sophisticated. It is possible to test the keyboard to see if a key has been pressed, and then to finish in good order, without a STOPPED IN LINE error. To achieve this result, we modify TRAVEL in one of two ways

```
TO TRAVEL?
    WOBBLE MOVE
    IF NOT RC? TRAVEL?
END

TO TRAVEL?
    WOBBLE MOVE
    TEST RC?
    IFTRUE STOP
    IFFALSE TRAVEL?
END
```

where both methods stop the turtle's travels at the hitting of a key. The TRAVEL?s are both recursive procedures, but with a difference — we can control the extent of the recursion from within the procedure. Instead of

having to use a 'system' facility (that is, CTRL-G), we use a program facility (that is, the test RC?).

The operation of both versions of TRAVEL? is such that the recursive calls to TRAVEL? are activated unless there has been a key pressed. The LOGO primitive RC? checks to see if there is a character in the 'input buffer', where the input buffer is comprised of the memory locations in which the C64 stores the most recent characters you have entered at the keyboard.

If a key has been pressed, that is, a character is waiting in the input buffer, then RC? provides the value TRUE, otherwise it outputs the value FALSE. In the case of the first version, if RC? is true then NOT RC? is false (and vice versa), so that if a key has not been depressed then NOT RC? is TRUE, and so the turtle TRAVEL?s.

The decision in the second version is easier to follow because first there is a TEST of RC?, and IFTRUE then the procedure STOPs whilst IFFALSE then the turtle TRAVEL?s. (There are shortened forms IFT and IFF—IFTRUE and IFFALSE respectively.)

For the first version of TRAVEL?, the stopping of the travelling is implied by the procedure (as the procedure just ENDs), whereas for the second version there is an explicit STOP. Though more long-winded, the second version has an extra clarity, because less is assumed.

There is, however, something wrong with both these procedures, and the same thing is wrong with both of them. I made a mistake in logic when I designed these procedures, so can you work out where I went wrong? It is only a small mistake, that is certainly true, but it is not one I can ignore — for it is an annoying, rather irritating, bug. Try to work it out before reading on.

Clearing up input

At the finish of either procedure, a letter appears in the lower text screen (sometimes even more than one) — this is the letter corresponding to the key you pressed. This produces great anguish because — if you forget the letter is there — that letter is automatically put on the front of anything you type. A silly error, but how can it be avoided? Do you always have to remember to use the DEL key?

The RC? procedure only checks to see if a key has been pressed (that is, that there is a character in the input buffer), the procedure does not read that character. The character is thus left in the input buffer, and so it appears at the end of the procedure, when the drawing finishes. That is the fiddling character which gets in the way. Some means has to be found of clearing out the input buffer, ready for sensible input at the end of the procedure.

The solution is mercifully simple

```
TO TRAVEL?
    WOBBLE MOVE
    IF NOT RC? TRAVEL? ELSE CLEARINPUT
END
```

```
TO TRAVEL?
    WOBBLE MOVE
    TEST RC?
    IFTRUE CLEARINPUT STOP
    IFFALSE TRAVEL?
END
```

and we use the LOGO primitive CLEARINPUT which clears the input buffer. The new improved version of TRAVEL? — kills most known bugs — is much cleaner in operation, as there are no dead characters hanging around to clutter up the keyboard. There is an even cleaner way if we use the joystick button.

I will only give one version of the procedure:

```
TO JOY.TRAVEL?
    WOBBLE MOVE
    TEST JOYBUTTON 1
    IFTRUE STOP
    IFFALSE JOY.TRAVEL?
END
```

and there are two differences to the corresponding TRAVEL? The first difference is that RC? is replaced by JOYBUTTON 1, a procedure which checks to see if the button on joystick 1 has been pressed, which produces the result TRUE or FALSE. The second difference is that there is now no need to clear the input buffer.

The joysticks are known as either 0 or 1, depending on the control port into which they are plugged. Joystick 0 is plugged into port 1, and joystick 1 plugs into control port 2. Normally, I use joystick 1 in such applications because control port 1 (that used by joystick 0) conflicts with use of the keyboard, and that is one confusion I will gladly forgo.

Already, I hope, you can see that the joystick makes life somewhat cleaner. And now to draw pictures using a joystick.

CHAPTER 5
Joyful Spritely LOGO

I drew a circle with a turcle.
Debbie, aged 19, on first meeting C64 LOGO

The joy in this chapter comes from the use of a joystick to control drawing on the screen.

If you do not have a joystick available, then do not despair, Chapter 7 shows how the action of a joystick can easily be emulated in LOGO. Even if you have a joystick, this chapter is worth studying: in fact, it is true that the use of the keyboard to control the turtle does have advantages for the production of certain effects — for example, some of the figures in Chapter 7 are more easily produced using the keyboard.

It is well worth studying Chapter 7 for another reason. The conversion from joystick to keyboard techniques is an extremely interesting and satisfying programming exercise. It is for this reason that there is no complete conversion of the joystick routines to the keyboard, because the completion of the project is left to you — so you need to read the present chapter, too. The solutions are not that difficult, and you will learn a great deal from the attempt — as long as you are ready to make mistakes.

Incidentally, I feel that the use of the joystick adds a great deal to the manipulation of graphics in general, and to the turtle graphics of LOGO in particular. Therefore, as a joystick is not vastly greater in price than many games, I think a joystick is a worthwhile purchase for any computer user. And when a mouse becomes available...

Joystick and turns

We are going to design a set of routines which will allow the turtle to draw on the screen, under control of a joystick.

If you don't have a joystick, read this chapter in conjunction with Chapter 7. The keyboard emulation of the joystick in Chapter 7 is an important topic in itself, and should be studied by those with a joystick as well as those without. Those without a joystick will find the discussion of

arithmetic and nested procedures in Chapter 7 of great general importance: do not ignore sections of this book because they do not appear to be directly relevant at this moment.

Start planning by inventing a procedure to do it all:

```
TO TURTLE
    TURN MOVE TURTLE
END
```

and with one procedure we have cracked the problem. All we do is TURN, then we MOVE, and then we TURTLE again. There is one slight problem — what are the definitions of TURN and MOVE? There seems no reason to knock a winning procedure, so we will keep to exactly the same moving combination as before:

```
TO MOVE
    FD 15
END
```

because we need modify only the turning of the turtle. The definition of the turning procedure is not at all complicated:

```
TO TURN
    RT 45 * JOY
END
```

and to understand the term '45 * JOY' we will first have to examine a joystick, and how it sends information to LOGO. Though it is possible to use many different varieties of joystick with the C64, I will describe a Commodore joystick.

The joystick is little more than a black handle which moves, and a red button which can be pressed. If the joystick is examined with the button away from you, between the button and the joystick handle there are the words 'commodore' and 'TOP'. The word TOP is on a ring which surrounds the handle, and on the same ring there are seven markings (with TOP being the eighth), where these indicate the eight positions of the joystick handle.

Actually, if you gyrate the handle, it appears as if the joystick can be in many more than these eight positions, but these positions are the only ones that the computer can distinguish on the joystick. Each of these eight valid directions is identified by a number.

If you have a joystick, plug it into control port 2 and enter this procedure:

```
TO READJOY
```

```
    PR JOYSTICK 1 ; PRINTS VALUE OF JOYSTICK 1
    IF JOYBUTTON 1 STOP ; IF BUTTON PUSHED, STOPS
    READJOY ; THE SAME AGAIN
END
```

which, when activated, starts by printing vast numbers of −1s, until the joystick handle is moved from its upright position: the value communicated by the joystick to LOGO, when the handle is at rest, is −1. The sequence stops when the button is pressed.

Moving the handle round in a clockwise direction results in the numbers on the screen changing from 0 (directly forward, ie TOP) round to 7 (that is, north west). This information is summarised in **Figure 5.1**, to assist those who may not have a joystick as well as those who have. It is this information which has to be communicated in some sensible way to the TURN procedure, via the (as yet undefined) procedure JOY.

If we consider the eight positions taken by the joystick as being matched with the degrees to be turned, the matching is that shown in **Figure 5.2.**

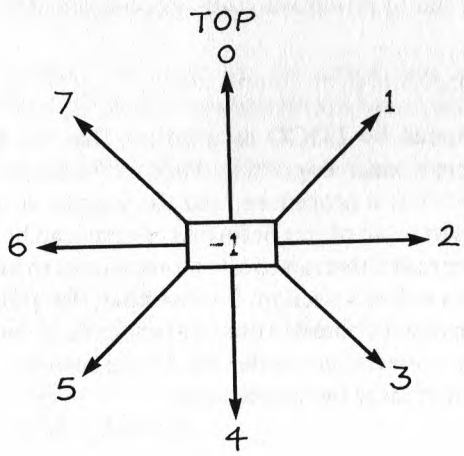


Figure 5.1: The Values Output by Joystick :N.

For those who are interested, *LLL* (Appendix) provides reference details on the use of joysticks. It also gives information about a set of joystick routines on the utility disk — the file on which the routines reside is named JOY, but is not in any way similar to my procedure of that name. So what does my JOY accomplish?

Procedures and OUTPUT

If we examine the crucial (and only) line in the definition of TURN, we

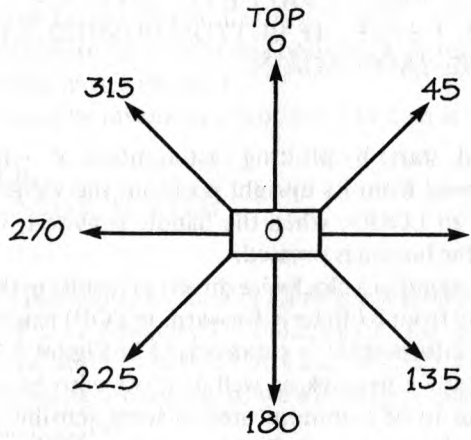


Figure 5.2: The Angles Turned by 45*JOY.

notice that JOY has to produce a value. Look again at that line:

```
RT 45 * JOY
```

which is interpreted by LOGO as meaning that the turtle turns right through a number of degrees given by 45 times the value of JOY. It is true, however, that JOY is a procedure, and the content of a procedure is its action: JOY has to be an object before its content can be a value.

It seems rather restrictive not to allow procedures to have an optionally-provided value as well as an action. For example, the command RANDOM has an action, in that it chooses a pseudo-random number, but it also has a value, where the value is that number which has been chosen. How to solve the problem? Enter these two procedures:

```
TO T1
END

TO T2
    OUTPUT [THIS IS OUTPUT FROM T2]
END
```

and then explore.

Here goes, adventurers in LOGO:

```
?T1 ; LINE 1
?[AN EXAMPLE] ; LINE 2
```

RESULT: [AN EXAMPLE]
?T2 ; LINE 3
RESULT: [THIS IS OUTPUT FROM T2]
?

where the explanation is very simple.

Line 1: In this line there is a call to the procedure T1, a procedure which does not do anything (check the definition). Nothing happens, and a query is printed on the next line ready for you to enter new instructions to LOGO.

Line 2: This is that next line. Here you enter a list, where the list consists of two elements. The two elements are the two words AN and EXAMPLE, and the response of LOGO is to say that the result from the previous line is to produce the list [AN EXAMPLE].

Line 3: A call is made to the procedure T2, and, on the next line, LOGO informs you that the result is a list, [THIS IS OUTPUT FROM T2]. If you look at the content of the definition of T2 you will see that there is one line, and that line instructs

OUTPUT [THIS IS OUTPUT FROM T2]

This is how a procedure not only has an action, but also, by use of OUTPUT, may have a value: a procedure always has an action, and sometimes it has a value.

Somewhere or other, therefore, JOY will have to OUTPUT, or, shortened, OP:

```
TO JOY
    LOCAL "JOYVAL ; LINE 1
    MAKE "JOYVAL JOYSTICK 1 ; LINE 2
    TEST :JOYVAL = -1 ; LINE 3
    IFT OP 0 ; LINE 4
    IFF OP :JOYVAL ; LINE 5
END
```

and not only does JOY use OP, but it also utilises the words LOCAL and MAKE.

Line 1: This sets up a LOGO object whose name is JOYVAL, and which only exists for that call of the procedure. The content of JOYVAL is limited to that procedure, and outside the procedure the object known as JOYVAL is unknown. A LOCAL object JOYVAL is a secret agent as far as procedures which call JOY are concerned. Like all good secret agents, though you may not know it exists, its effects are felt. The quote mark (") at the beginning of the name JOYVAL is to make clear to LOGO that it is the name to which reference is being made.

Line 2: The local object named JOYVAL is made to contain the value which is produced by JOYSTICK 1. That is, the value of the current state of the joystick is stored temporarily, for the life of the procedure call, in the object JOYVAL.

Line 3: A TEST is made to find if the value stored in JOYVAL is equal to −1. That is, a test is made to see if the handle is in the centred position.

Line 4: If the result of the TEST was TRUE, then the value OutPut by the procedure is 0. That is, if the handle is centred, then the turtle does not turn.

Line 5: If the joystick was not centred then the procedure outputs the value contained in the object JOYVAL, which still contains the value produced by the joystick at the instant it was assigned in line 2.

Thus we have it. TURTLE calls the procedures TURN MOVE TURTLE, where the call to TURTLE is recursive.

The procedure MOVE only uses a LOGO primitive (ie FD), and the procedure TURN uses the LOGO primitives RT and *, plus a procedure JOY, which outputs a numerical value.

The procedure JOY uses a fair number of LOGO primitives, LOCAL MAKE JOYSTICK TEST IFT OP IFF, and a local object named JOYVAL. We now have a complete system.

The structure of the TURTLE system is shown in **Figure 5.3**, and various examples of the use of the system are given in **Figures 5.4, 5.5** and **5.6.** None of the uses is exceptional, and it is worth noting the repetition of motifs in all three: the octagons (which have the appearance of circles), the squares, and the eight-pointed stars. Why is this so?... Well, it all comes down to the STAR procedure, though I will leave you to work out why.

One thing which can be said of the TURTLE system is that you have to concentrate all the time on what is happening. There is no rest for you,

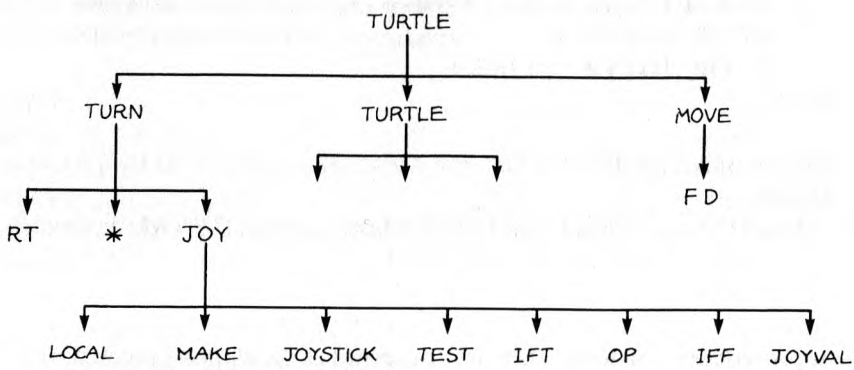


Figure 5.3

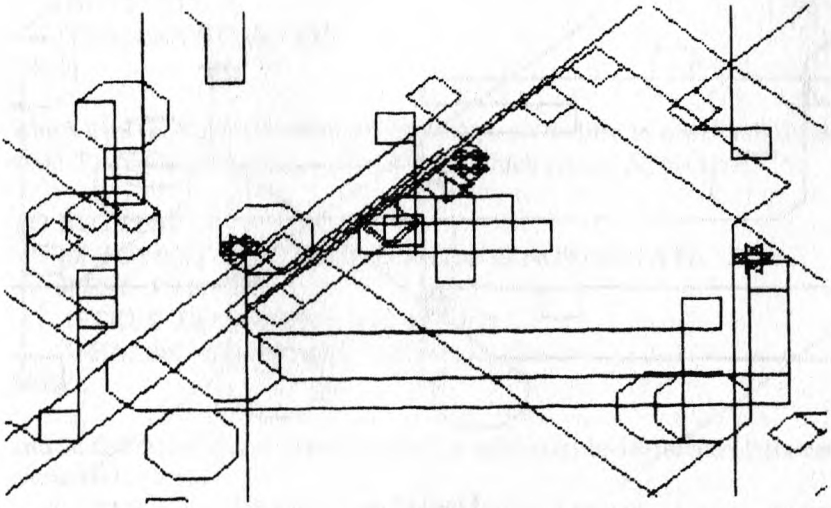


Figure 5.4

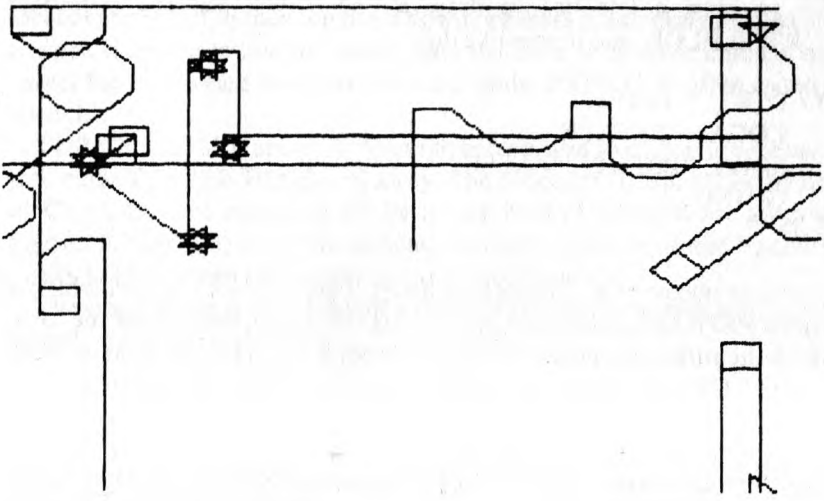


Figure 5.5

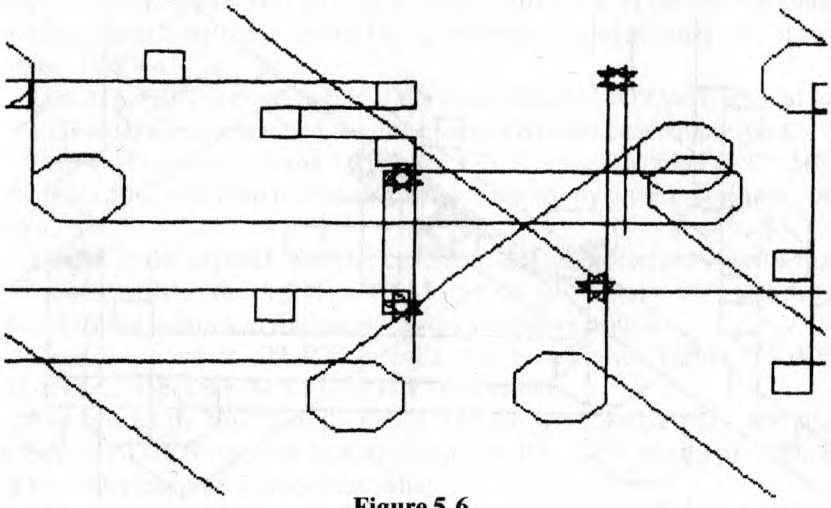


Figure 5.6

wicked or not. I will not show you how to stop the turtle, but perhaps it does not have to draw all the time — read on.

A dotted trail

I propose a system which lets the turtle travel as before, but which allows you to draw a line or not draw a line. I will call the procedure TURTLE.BLOB, and it goes like this:

```
TO TURTLE.BLOB
    LOCAL "PSTATE
    MAKE "PSTATE "FALSE
    GOING
END
```

a cunning means of giving nothing away. There is a local object which is named PSTATE, and this object is going to contain the state of the 'pen' which the turtle is supposed to use to draw. PSTATE is defined as being local to TURTLE.BLOB because it has no use outside that procedure.

Making the object local is also protection, in case there is another PSTATE in another completely unconnected procedure, as you don't want the two examples of PSTATE to interact or interfere. The initial value of PSTATE is made equal to the word (or name) FALSE, which is the LOGO identifier for 'false' (see the previous chapter).

Once the local object has been declared and given a value, the turtle can be set GOING:

```
TO GOING
    BUTTEST
    TURN MOVE GOING
END
```

which uses the classic method of repeating a procedure by use of tail recursion. The novel procedure is BUTTEST, which can be defined by:

```
TO BUTTEST
    IF JOYBUTTON 1 MAKE "PSTATE NOT :PSTATE
    TEST :PSTATE
    IFTRUE PENDOWN ; OR, IFT PD
    IFFALSE PENUP ; OR, IFF PU
END
```

and its definition shows some interesting asides on the distinction between name and content.

BUTTEST is within the procedure TURTLE.BLOB, and all objects which are local to TURTLE.BLOB also exist for BUTTEST, because — in a sense — BUTTEST is local to TURTLE.BLOB. If the button on joystick 1 has been pressed, then the content of the object named PSTATE is made equal to NOT the previous content of that object. This means that if the content of PSTATE was FALSE, then it becomes TRUE — and vice versa.

The local object is thus used as a means of switching from one state of the pen to another, and this is accomplished by the TEST. If the content of the local object is TRUE then the pen is down, whereas if the content is FALSE the pen is up. Note that an object does not have to contain a numerical value, for in this case it contains a word value. PSTATE is often called a 'switch'.

TURN and MOVE are no different to before, and so all that is necessary is to enter TURTLE.BLOB and away. The procedure is still halted by use of CTRL-G. Two outcomes of the use of TURTLE.BLOB are given in **Figures 5.7** and **5.8**, where the delicate dotted lines are produced by continually holding down the button on the joystick control.

There are better ways of doing what I've just done, so why not invent some?

More about disks and files

First check that you have the procedures TURTLE TURTLE.BLOB TRAVEL working correctly.

By dint of checking, you will discover that TRAVEL uses the user-created procedures 'WOBBLE MOVE', TURTLE uses 'TURN MOVE JOY' with a LOCAL object JOYVAL which only lasts for the length of the

procedure, and TURTLE.BLOB uses 'GOING BUTTEST TURN MOVE' plus the LOCAL object PSTATE. The sum total of the procedures needed to play with the turtle is thus [TRAVEL WOBBLE MOVE TURTLE TURN JOY TURTLE.BLOB GOING BUTTEST].

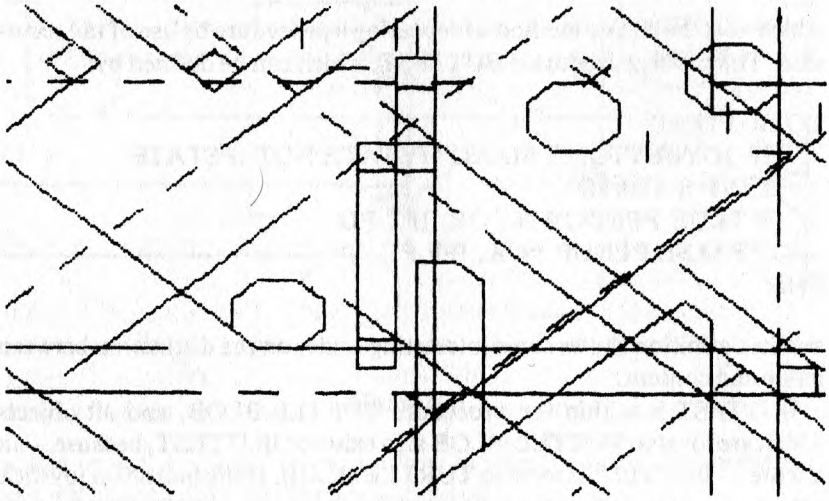


Figure 5.7

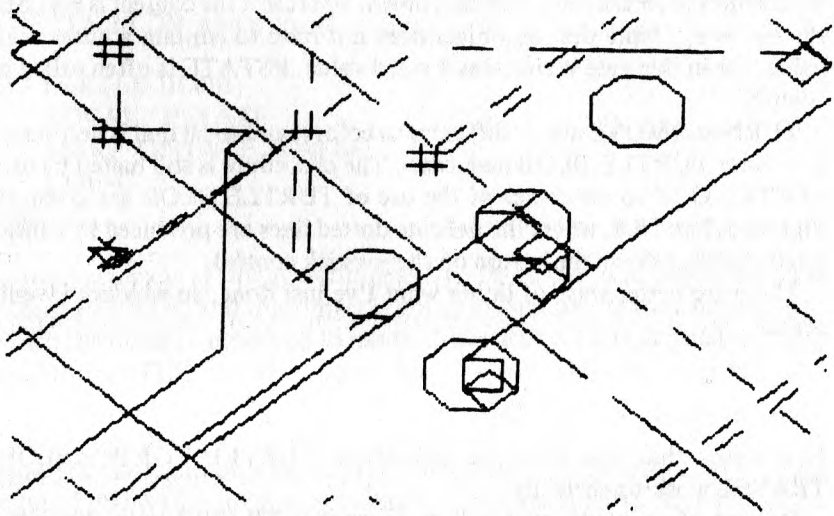


Figure 5.8

It is possible to save everything we have created if we wish, but a check with

```
?POTS PO NAMES
```

will probably reveal so much garbage that you do not wish to keep it. Help is at hand, where it is very imortant that both the round parentheses '( )' and square brackets '[ ]' are in the correct positions:

```
(SAVE "TURTLE.TRICKS [TRAVEL WOBBLE MOVE TURTLE
TURN JOY TURTLE.BLOB GOING BUTTEST])
```

The screen goes blank, and a disk file named TURTLE.TRIC.LOGO is revealed by the use of a CATALOG.

Even though you named your file TURTLE.TRICKS (only 13 characters), you cannot have all the characters. So that LOGO knows that this file is a file of procedures, it adds an 'extension', .LOGO. The filename, with extension, is 16 characters which is the maximum admissable length for a disk file on the C64. Say GOODBYE and then:

```
?READ. "TURTLE.TRICKS
TRAVEL DEFINED
WOBBLE DEFINED
MOVE DEFINED
TURTLE DEFINED
TURN DEFINED
JOY DEFINED
TURTLE.BLOB DEFINED
GOING DEFINED
BUTTEST DEFINED
?
```

and the order in which the procedures are loaded into your workspace corresponds to the order in which you informed LOGO they were to be SAVEd. You now know that the list which followed "TURTLE.TRICKS was the list of procedures to be saved, and that was the order in which they were saved.

The LOGO command SAVE normally only takes one parameter, the name which directly follows the command. This is where the round parentheses come in. Square brackets are used to set the limits to a list, whereas round parentheses are used to group elements together. (On such things see also *LLL*, Graphics and Appendix.) So the SAVE command normally has only one parameter (a name), but it can, optionally, have a second parameter (a list). The command knows whether there are one or

two parameters by the presence or absence of the parentheses. The parentheses enclose the unit: the unit in this case extends from 'SAVE' to 'BUTTEST]', and so two parameters are enclosed. This technique can also be used with PRINT

```
?PR 2 3
2
RESULT: 3
?(PR 2 3 )
2 3
?
```

Normally the printing command only takes one parameter, that is, the first in line, and it leaves the second in line alone. The parentheses enclose the PR and all the parameters to be printed.

Perhaps now we can look at some sprites...

Loading sprites

Place the LOGO utility disk in the drive and type

```
?READ "SPRITES ; LINE 1
```

Sometimes the screen goes blank at this point.

```
TS? DEFINED
EACH1 DEFINED
ASK DEFINED
SMALLX DEFINED
BIGX DEFINED
EXP2 DEFINED
TB? DEFINED
READSHAPES DEFINED
?PO NAMES ; LINE 2
?PO READSHAPES ; LINE 3
TO READSHAPES :FILE
BLOAD WORD :FILE ".SHAPES
END

?READSHAPES "ANIMALS ; LINE 4
?PO NAMES ; LINE 5
"DINOSAUR IS 1
"KANGAROO IS 2
"BUG IS 3
```

```
"DOLPHIN IS 4
"HORSE IS 5
"CAT IS 6
"BUTTERFLY IS 7
"STARTUP IS [BLOAD "ANIMALS.SHAPES]
?
```

and, if you have not already done so, read in your file of procedures called TURTLE.TRICKS.

Line 1: This line reads in a LOGO file named SPRITES.LOGO which contains the procedures given in the list of definitions which follow.

Line 2: The file SPRITES.LOGO does not contain any objects, which is shown by the null response to the request to PO NAMES.

Line 3: One of the procedures defined from SPRITES.LOGO is named READSHAPES, and in this line the content of this procedure is requested. The result follows, and it can be seen that READSHAPES takes one parameter, that known as FILE. This name is concatenated with the word .SHAPES — which is a file extension of a different name — to produce a WORD by use of the procedure of that name. The resulting WORD is then taken as the name of a binary file which is loaded into memory. WORD works in this way:

```
?PR WORD "HERE "TO
HERETO
?
```

Line 4: The procedure READSHAPES is now used, on a file on the disk named ANIMALS.SHAPES, which (because the extension .SHAPES is automatically added by the procedure) we call by using the name ANIMALS as the parameter. Binary information is taken off the file and loaded into the C64's memory, and the loading is into the memory locations where the C64 contains information about the shapes of sprites. We have loaded some definitions of sprite shapes into the C64's memory.

Line 5: We now print out the names of all the objects known to LOGO, and find we have a list of names of animals (whether a butterfly or a bug is an animal is a deep philosophical question). It seems as if the value of a BUG is 3, and a BUTTERFLY is 7, so we can check by

```
?(PR :BUG :BUTTERFLY)
3 7
?
```

and, though we do not know what to do with a bug and a butterfly, we know their numbers.

Follow this sequence carefully:

```
?DRAW HT ; THE TURTLE DISAPPEARS
?TELL :BUTTERFLY ; WE ARE NOW USING THE BUTTERFLY
?ST ; AND NOW WE CAN SEE IT
?FULLSCREEN TRAVEL ; AND HOW IT FLUTTERS
```

and we have a fluttering butterfly. The procedure, TRAVEL, is exactly that utilised before with the turtle. The turtle is a special sprite whose number is 0.

We can see how similar the performance is by stopping the butterfly. As we are still in a fullscreen mode, press f1. Next enter

```
?PD FULLSCREEN TRAVEL
```

and the butterfly draws a track on the screen. The fluttering of the butterfly is equivalent to the wobbling of the turtle.

There are certain distinctions to be made between the sprites numbered 1 to 7, in comparison to sprite 0 (the turtle).

1) The sprites cannot turn, though they can move in all directions: the butterfly, for example, always seems to point in the same direction, even though it may move sideways.
2) The sprites have to be 'turned on' — they start by being invisible and have to be made visible. A sprite is selected by the TELL command, for example, sprite number 4 is selected by TELL 4. Stop the butterfly, therefore, and enter 'TELL 4 ST TRAVEL' and the butterfly remains where it was stopped.
3) When a sprite starts, its pen is up, and not down as is the case of the turtle.

And so now try

```
?TELL 4 HT
?TELL 7 HT
?TELL 0 HT
?TELL 3 ST
?TURTLE
```

This will produce a beetle which you can attempt to guide across the screen. Sprite 4 is hidden, sprite 7 is hidden, and the turtle (sprite 0) is hidden; sprite 3 is shown, and then that sprite is told to obey the injunction TURTLE.

It is well worth playing with some of the other procedures using different sprites, and next we will have all the sprites moving together. You may like to start reading the Sprites section in *LLL*.

CHAPTER 6

Simultaneous Sprites in LOGO

People who say that I repeat too much
do not really listen;
they cannot hear that
every moment of life is full of repeating.
Gertrude Stein

Have you ever come across a *List of things to do* on which the first item is 'Make a list of things to do'?

A list in LOGO is a sequence of items, as is a shopping list. Sometimes, in a LOGO list, the order of the items does not matter (as is the case with most shopping lists); sometimes, however, the order of the items is crucially important (as is the case with most lists of instructions). The order of the items in a REPEAT list is crucial, as can be shown by comparing the result of REPEAT 4 [FD 50 RT 90] with the slightly different sequence in REPEAT 4 [RT 90 FD 50]: both uses of REPEAT produce a square, but in different directions.

In the next couple of sections we will prepare the way for simultaneous TELLing of sprites, by examining how lists can be treated as pieces of program text.

RUNning lists

First MAKE a LOGO object contain a list, where we give the object the name L1:

```
?MAKE "L1 [FD 50 RT 90]
?REPEAT 4 :L1 ; DRAWS A SQUARE HERE
?:L1
RESULT: [FD 50 RT 90]
?
```

When LOGO comes across REPEAT, it treats the contents of L1 just as if the list contained there was a sequence of program instructions. The con-

tent of the object L1 is a list of instructions to draw the side of a square.

If we now progress slightly by assessing the effects of

```
?DRAW REPEAT 1 :L1 ; ONE SIDE IS DRAWN
?RUN :L1 ; ONE SIDE IS DRAWN
?
```

we find that two sides of the square are drawn, one side as a result of REPEAT 1 and the next side as a result of RUN. Thus, the LOGO command RUN is equivalent in effect to a REPEAT 1 sequence; and RUN executes the contents of the list which follows, as if the list contained program instructions.

RUN is an exceedingly powerful facility, and one which is not generally available in more mundane languages (and should not be confused with the BASIC command RUN). To start to see why the facility is so powerful, and how useful RUN can be, define a procedure

```
TO SQ :SIDE
   REPEAT 4 [FD :SIDE RT 90]
END
```

If we then use the procedure SQ, in combination with a few other trimmings, as follows

```
?MAKE "L2 80
?MAKE "L3 [SQ :L2]
?RUN :L3 ; DRAWS A SQ OF SIDE 80
?
```

a square of side 80 is drawn. In the analysis of this short session, so that we are able to follow more closely what happened and why, we can use another helpful C64 LOGO command — that known as TRACE.

The TRACE command is used in this manner

```
?TRACE RUN :L3
TRACING ON
EXECUTING SQ 80
REPEAT 4 [FD :SIDE RT 90]
ENDING SQ
?
```

thus when LOGO first encounters RUN :L3 it expands :L3 into the list [SQ :L2]. The name SQ is recognised as the name of a procedure, and LOGO

expands :L2 into the number 80. Thus, the list which is RUN is taken to be [SQ 80]. This expansion comprises the first line of the TRACE.

In executing the procedure (the next line of the trace), the parameter :SIDE is taken to be 80, and the square is drawn with side 80. It is at this stage of the trace that the tracing routine stops, to wait for a keypress (the turtle disappears on TRACE and reappears on a keypress). If you examine the turtle when it reappears, you can see it waiting for you to stir it into activity.

We will now try a trifle more complexity, because it would be rather pleasing to be able to treat the name of the procedure and the value of the parameter as distinct elements. Here goes:

```
?MAKE "L4 "SQ
?:L4
RESULT: SQ
?SENTENCE :L4 :L2
RESULT: [SQ 80]
?
```

The new command SENTENCE, or (in short) SE, forms a list from its two inputs and, if there are more or less than two inputs, then we use parentheses.

```
?(SE :L4)
RESULT: [SQ]
?(SE :L4 :L2 :L4 :L2)
RESULT: [SQ 80 SQ 80]
?(SE :L4 :L2 :L4 :L2 ; MISSING)
SE NEEDS MORE INPUTS
?SE :L4 :L2 :L4 :L2 ; NO ( )
YOU DON'T SAY WHAT TO DO WITH [SQ 80]
?
```

These outcomes can be compared with those from PRINT, which also uses parentheses to group items. In the case of PRINT (or PR) parentheses are not needed for one item, but they are necessary for more than one item.

As a final little experiment in this section (still with TRACE active, can you stand the excitement?) try

```
?RUN SE :L4 :L2 ; ANOTHER SQUARE
EXECUTING SQ 80
```

```
    REPEAT 4 [FD :SIDE RT 90]
ENDING SQ
?
```

with the story as before, only we had to use SEntences in this story.

Remember to switch the tracing off by use of NOTRACE when you have finished your explorations, otherwise you will become heavily bogged down in words on the screen.

Procedures as parameters

Have a close look at this procedure:

```
TO DRIVE :PROCEDURE :PARAMETER
    RUN SE :PROCEDURE :PARAMETER
END
```

and then, when that is safely stored away, enter (you need to TRACE again):

```
?DRAW DRIVE :L4 :L2
EXECUTING DRIVE SQ 80
RUN SE :PROCEDURE :PARAMETER
EXECUTING SQ 80
REPEAT 4 [FD :SIDE RT 90]
ENDING SQ
ENDING DRIVE
?
```

where it is important to note that the drawing does not commence until the line 'REPEAT 4 [FD :SIDE RT 90]'. The procedure DRIVE has two parameters, where the first is the name of a PROCEDURE, and the second is the list of parameters associated with the PROCEDURE (which was, of course, given as first parameter to DRIVE).

We can start to perform quite sophisticated games (fun-and-games rather than video-games):

```
?SE "REPEAT [4 [FD 50 RT 120]]
RESULT: [REPEAT 4 [FD 50 RT 120]]
?RUN SE "REPEAT [4 [FD 50 RT 120]]
?
```

and a triangle appears (with no information from TRACE, as the line

executed instantly). Before I go any further I need at least two more procedures, which are:

```
TO TRI :SIDE
    REPEAT 3 [FD :SIDE RT 120]
END

TO PENT :SIDE
    REPEAT 5 [FD :SIDE RT 144]
END
```

To guess what I am to do, here is a sample sequence:

```
?DRAW
?DRIVE "TRI 30 LT 120 DRIVE "SQ 60 LT 120 DRIVE "PENT 90
EXECUTING DRIVE TRI 30
RUN SE :PROCEDURE :PARAMETER
EXECUTING TRI 30
REPEAT 3 [FD :SIDE RT 120]
ENDING TRI
ENDING DRIVE
EXECUTING DRIVE SQ 60
RUN SE :PROCEDURE :PARAMETER
EXECUTING SQ 60
REPEAT 4 [FD :SIDE RT 90]
ENDING SQ
ENDING DRIVE
EXECUTING PENT 90
RUN SE :PROCEDURE :PARAMETER
EXECUTING PENT 90
REPEAT 5 [FD :SIDE RT 144]
ENDING PENT
ENDING DRIVE
?
```

— the result is shown as **Figure 6.1.**

What has happened is that the names of the procedures (that is, each first parameter of DRIVE) has in turn generated the corresponding procedure, with each second parameter of DRIVE being used for the parameter to the named procedure.

Try to follow what happens here (it is not all that difficult):

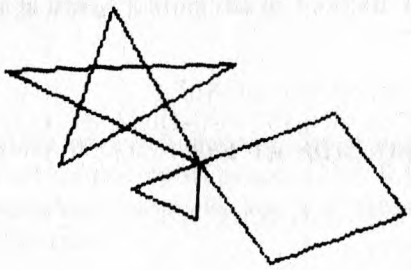


Figure 6.1

```
?MAKE "L5 [TRI SQ PENT]
?ITEM 1 :L5
RESULT: TRI
?ITEM 2 :L5
RESULT: SQ
?ITEM 3 :L5
RESULT: PENT
?DRAW DRIVE ITEM 3 :L5 100 ; A DELIBERATE CHOICE
EXECUTING DRIVE PENT 100
RUN SE :PROCEDURE :PARAMETER
EXECUTING PENT 100
REPEAT 5 [FD :SIDE RT 144]
ENDING PENT
ENDING DRIVE
?DRAW DRIVE ITEM ( 1 + RANDOM 3 ) :L5 100 ; A RANDOM
        CHOICE
EXECUTING DRIVE TRI 100
RUN SE :PROCEDURE :PARAMETER
EXECUTING TRI 100
REPEAT 3 [FD :SIDE RT 120]
ENDING TRI
ENDING DRIVE
?
```

Well, it seems obvious that the command ITEM takes two parameters, where the first is the number of an item in a list, and the second is the name of the list. Thus ITEM 1 :L5 extracts the first item in the list L5, and that item is TRI. In the case of the first use of DRIVE (the deliberate choice) we select the third item, that is, PENT. When DRIVE is used the second time, the selection is random, and your result may differ.

We are going to use random selection of items to produce some random shapes, with random sizes.

Random procedures, random parameters

The two procedures are

```
TO RANDSHAPES :INLIST :SCALE
    DRIVE CHOOSE :INLIST RANDOM :SCALE
    PU SETXY RANDOM 320 RANDOM 260 LT RANDOM
        360 PD
    IF NOT RC? RANDSHAPES :INLIST :SCALE ELSE
        CLEARINPUT
END

TO CHOOSE :INLIST
    OUTPUT (ITEM 1 + (RANDOM COUNT :INLIST) :INLIST)
END
```

and I will explain first the content of CHOOSE, because it is simpler.

The CHOOSE procedure takes one parameter, which is a list (we hope), and selects an item at random. The command RANDOM itself takes one parameter, a number which is used to set limits on the resulting random number. The number used to set the limits is (for this procedure) COUNT :INLIST, where COUNT is a LOGO command which outputs the number of items in a list.

Here is something to try

```
?CHOOSE :L5
RESULT: SQ
?CHOOSE "ALONGWORD
RESULT: G
?CHOOSE CHOOSE :L5
RESULT: Q
?
```

and — as you can see — the ITEM and COUNT commands also work on names. Furthermore, we can try

```
?CHOOSE 1234567
RESULT: 5
?
```

thus we can see numbers are a special type of name (see Chapter 8).

The other procedure, RANDSHAPES, takes two parameters: the first is the content of an object, which should be a list of procedure names (in our case it will be L5); and the second is the scale size of the shapes (that is, the maximum length of side). The first line of the procedure calls DRIVE to

draw a shape (with a random item from INLIST, and a random size of side).

The next line of RANDSHAPES means that the turtle lifts the pen and moves to a set of random X and Y coordinates: try to work out why only positive coordinate values are given. The pen is lowered.

If a key has not been pressed, we execute RANDSHAPES again; if it has, we clear the input buffer.

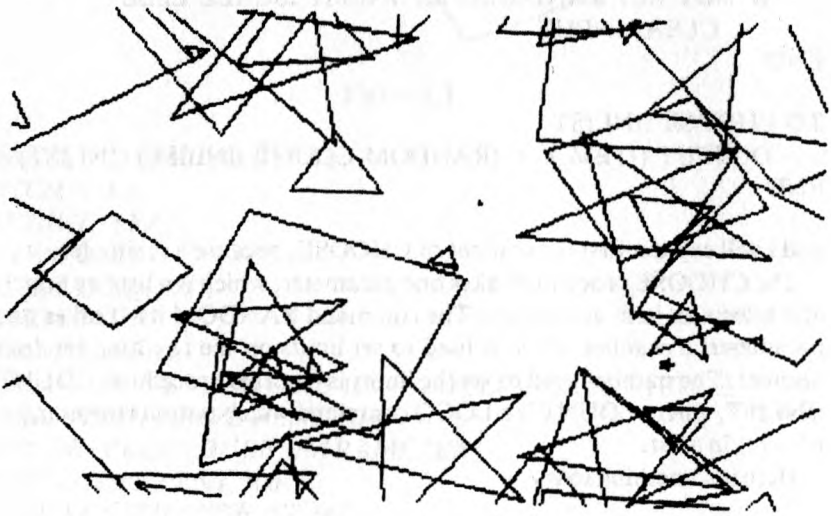


Figure 6.2

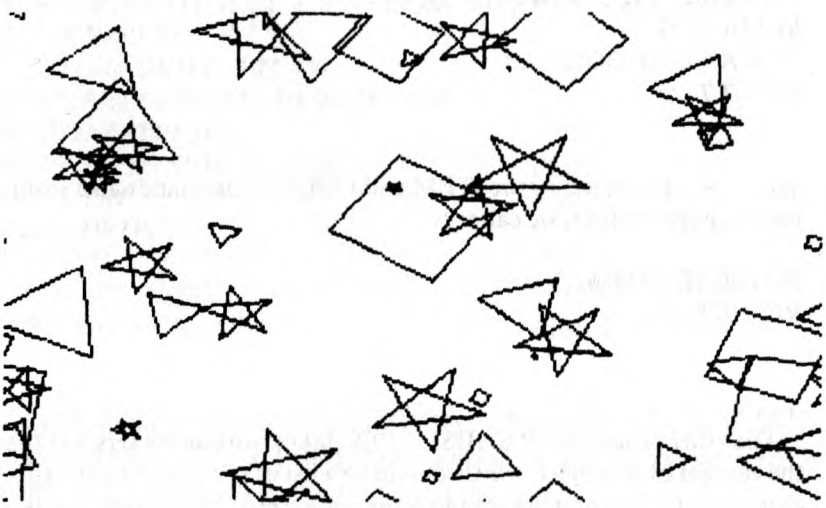


Figure 6.3

The results of two activations of RANDSHAPES are shown in **Figures 6.2** and **6.3**, where the scales differ. You may like to make rather more sophisticated versions of DRIVE and RANDSHAPES.

Spritely activity

First, we have to load the sprite procedures into the computer's memory. Thus we proceed by entering

```
?READ "SPRITES ; SEE LLL, SPRITES
TS? DEFINED
EACH1 DEFINED
EACH DEFINED
ASK DEFINED
SMALLX DEFINED
BIGX DEFINED
SMALLY DEFINED
BIGY DEFINED
EXP2 DEFINED
TB? DEFINED
READSHAPES DEFINED
?
```

and by loading the file SPRITES.LOGO, which is what is accomplished by the READ "SPRITES command, we now have available several new procedures — those listed above.

We will start to investigate these procedures, but first use DRAW HT to clear the drawing screen, and hide the turtle. Get back to the text screen by using f1.

The first procedure we will study is READSHAPES, because that shows an interesting use of the WORD procedure. The procedure also uses the BLOAD procedure (*LLL*, Appendix), which loads a binary file from disk into memory. In this case, the portion of memory into which the information is loaded is that occupied by the sprite shapes.

(Though in no way necessary, it might be of interest to examine the two Sprites chapters in the C64 *Programmer's Reference Guide*, to see how the sprites are stored and activated. After reading all that, you will realise why it is so easy in LOGO to use sprites; and why it is so difficult in BASIC.)

Continue the session with the examination of READSHAPES:

```
?PO READSHAPES
TO READSHAPES
BLOAD WORD :FILE ".SHAPES
END
```

```
?WORD "ROT "TER
RESULT: ROTTER
?
```

so that, it can be seen, the file extension .SHAPES is added to the end of the name we provide. One good reason for using file extensions is that file extensions save possible confusions between sprite shapes known as ANIMALS.SHAPES and procedures known as ANIMALS.LOGO.

Tell the animals

After finding out how READSHAPES is defined, we will use the procedure to read in the already defined ANIMALS.SHAPES — my favourite selection from the sets of shapes available on the utility disk.

```
?READSHAPES "ANIMALS ; EQUIVALENT TO BLOAD
        ANIMALS.SHAPES
?TELL 7 ST ; SEE LLL (SPRITES) FOR ANIMALS.SHAPES
?BIGX ; SHAPE 7 IS A BUTTERFLY, AND NOW IT IS WIDER
?BIGY ; AND TALLER
?SMALLX ; AND THIN AGAIN
?SMALLY ; AND SHORT
?
```

The BUTTERFLY is my favourite sprite shape (closely followed by the BUG), and the shrinking and stretching of the butterfly are a delight to see. We will return to elastic shapes in the next section after we have learnt to talk to the animals.

If you want to tell the shapes what to do by individual name, LOGO does not stop you. It is possible to use the file ANIMALS.LOGO to achieve something only Dr Doolittle thought feasible.

```
?READ "ANIMALS ; THE FILE ANIMALS.LOGO
?PO NAMES
"DINOSAUR IS 1
"KANGAROO IS 2
"BUG IS 3
"DOLPHIN IS 4
"HORSE IS 5
"CAT IS 6
"BUTTERFLY IS 7
"STARTUP IS [BLOAD "ANIMALS.SHAPES]
?MAKE "TURTLE 0 ; IN CASE IT FEELS LEFT OUT
?DRAW HT ; AND THE BUTTERFLY DISAPPEARS NOT THE
        TURTLE
```

```
?TELL :TURTLE HT ; NOW IT GOES
?TELL :BUG ST ; AND THE BUG APPEARS
?BACKGROUND 3 PENCOLOR 6 ; NICE COLOURS? (BG 3 PC 6)
?FD 100 ; GET THE BUG OUT OF THE WAY
?TELL :BUTTERFLY ST ; A RATHER ANAEMIC BUTTERFLY
?PC 9 FD 50 ; MUCH MORE COLOURFUL
?TELL :DOLPHIN ST PC 0
```

and there should now be three sprites on the screen at once — all motionless. At the top is the BUG, coloured blue, in the middle of the screen is the DOLPHIN, coloured black, and in the middle of the two is the BUTTERFLY, coloured brown (though the colours vary with the television).

If you want to use colour names instead of numbers (the numbers are in the Appendix of *LLL*), then READ "COLORS to allow you to refer to light grey as LGRAY.

If you now use DRAW, the picture is revolting. The background is that murky grey colour with the turtle hiding the dolphin. If you issue an instruction to HT, it is the dolphin which disappears, not the turtle. Get the dolphin back by ST, and then

```
?WHO ; THIS COMMAND HAS NOT BEEN DOCTORED
RESULT: 4
?TELL :TURTLE HT TELL :DOLPHIN BG 3 ; GOODBYE
      TURTLE
?
```

The command WHO outputs a number, and the number corresponds to that of the currently active sprite.

Even though we have reset the graphics screen and the turtle has automatically appeared, the dolphin is still the sprite which obeys instructions. Here is a task for you — design a procedure which will reset the graphics screen but will not leave the turtle on the screen, unless it is the active sprite. The use of WHO is important in our analysis of the 'elastic' commands for sprites — see the next section.

Controlling the content of memory

The use of C64 sprites in most languages other than LOGO is extremely difficult, because sprite manipulation involves extensive peeking and poking in the C64's memory. In LOGO it is so easy to use procedures such as SMALLX that it is interesting to find how they work. First print out the definitions of the two procedures for changing in the X direction:

```
?PO [SMALLX BIGX] ; FOR INTEREST
TO SMALLX
```

```
    .DEPOSIT 53277 BITAND 255 — EXP2 WHO .EXAMINE
        53277
END

TO BIGX
    .DEPOSIT 53277 BITOR EXP2 WHO .EXAMINE 53277
END

?
```

The LOGO command .DEPOSIT takes two parameters: the first parameter is the value of a memory location in the C64 (for example, location 53277) and the second is the value which has to be deposited in that memory location (for example, BITAND 255 — EXP2 WHO .EXAMINE 53277).

The next new LOGO command is BITAND, which outputs the bitwise AND between its two inputs (BITOR outputs the bitwise OR). The exact details of BITAND and its operation are not important for present purposes, but what are the two parameters in this case? The parameters are '255 — EXP2 WHO' and then '.EXAMINE 53277'. The first parameter is the result of subtracting EXP2 WHO (whatever that is) from 255, and the second parameter is .EXAMINE 53277.

.EXAMINE takes one input, corresponding to a memory location number, and returns the content of that location — it 'examines' the location. Thus the value to be deposited in location 53277 is some function of the sprite number (ie WHO) in conjunction with the original content of that location. The twin commands .DEPOSIT .EXAMINE correspond to POKE and PEEK in some other languages.

(In the *Programmer's Reference Guide* (Chapter 5, C64 Input/Output Assignments) we find that location 53277 is the location which controls the horizontal sizes of sprites. The complicated arithmetic in SMALLX or BIGX, which need not concern you, is there to simplify your manipulation of sprites. You are protected from such difficulties in LOGO.)

Finally, what is the definition of the procedure EXP2?

```
?PO EXP2 ; IT APPEARS IN THE TWO PREVIOUS
        PROCEDURES
TO EXP2 :N
OP ITEM 1 + :N [1 2 4 8 16 32 64 128]
END

?EXP2 7 ; THIS CORRESPONDS TO SPRITE 7
RESULT: 128
?
```

and the technical-minded will be able to relate this to the contents of bit-switches in location 53277 — but who needs to bother?

Semi-simultaneous sprites

Here is a list of instructions for a sprite, where the list provides instructions which operate on the current sprite:

```
?MAKE "S.LIST [RT WHO + 1 FD( WHO + 1 ) * 10]
?RUN :S.LIST ; AND THE DOLPHIN MOVES A SMALL
        DISTANCE
?TELL BUTTERFLY RUN :S.LIST ; THE FLUTTERBY MOVES
        MORE
?
```

Why not all at once? Why not, indeed...

```
TO MOVE.EM.ALL :L
    BG 1
    MOVE.IT :L 0
END

TO MOVE.IT :INLIST :N
    TELL :N ST
    PC :N + 2
    RUN :INLIST
    MOVE.IT :INLIST REMAINDER (:N + 1) 8
END
```

and, when activated by MOVE.EM.ALL, all the sprites move at once — or rather, they move one at a time. The turtle still has the pen down, and you can see its slow trudge. Make a small alteration to the procedures:

```
TO MOVE.EM.ALL :L :LIMIT
    BG 1
    EACH [0 1 2 3 4 5 6 7] [PU HT]
    MOVE.IT :L 0 :LIMIT
END

TO MOVE.IT :INLIST :N :L
    TELL :N ST
    PC :N + 2
    RUN :INLIST
```

```
    MOVE.IT :INLIST REMAINDER (:N + 1) :L :L
END
```

The procedures are now ready, but first to hide all the sprites by 'EACH [0 1 2 3 4 5 6 7] [HT]'. To use three sprites we enter 'MOVE.EM.ALL :S.LIST 3', and there is more obvious movement — they appear slightly more 'simultaneous'. With only two sprites, we get as close as we can to simultaneity — which is not all that close.

Into the new version of MOVE.EM.ALL, I have insinuated the procedure EACH, which is one of the procedures from the SPRITES file. What does it do?

```
?PO [EACH EACH1] TRACE EACH [6 7] [ST PD 5D 50]
TO EACH :E.WHO :E.WHAT
    LOCAL "O.WHO MAKE "O.WHO WHO
    EACH1 :E.WHO
    TELL :O.WHO
END

TO EACH1 :E.WHO
    IF EMPTY? :E.WHO STOP
    TELL FIRST :E.WHO RUN :E.WHAT
    EACH1 BF :E.WHO
END

TRACING ON
EXECUTING EACH [6 7] [ST PD FD 50]
LOCAL "O.WHO MAKE "O.WHO WHO
EACH1 :E.WHO
EXECUTING EACH1 [6 7]
IF EMPTY? :E.WHO STOP
TELL FIRST :E.WHO RUN :E.WHAT
EACH1 BF :E.WHO
EXECUTING EACH1 [7]
IF EMPTY? :E.WHO STOP
TELL FIRST :E.WHO RUN :E.WHAT
EACH1 BF :E.WHO
EXECUTING EACH 1 [ ]
IF EMPTY? :E.WHO STOP
ENDING EACH1
TELL :O.WHO
ENDING EACH
?NOTRACE
TRACING OFF
?
```

I will not explain how EACH works, but I will tell you what the two parameters are: the first parameter is a list of sprite numbers, and the second is the action to be taken by the sprites. The same action can have different results if, for example, WHO is used within the listed actions.

The first parameter does not have to be a list: to activate all sprites use 'EACH "01234567 :LIST', where the quote mark " is very important if the first sprite is sprite 0. If 01234567 is entered without the quote then LOGO interprets it as 1234567, and thus the zero sprite is lost. For example,

```
?FIRST "01234567
RESULT: 0
?FIRST 01234567
?RESULT: 1
?
```

List processing can produce powerful results.

It will be worthwhile to study the routines provided on the utility disk.

Saving sprite pictures

If you want to make your LOGO system hang around doing nothing, and be unable to do anything, then try to SAVEPICT with sprites on the screen. All you can do is switch off.

In fact

Warning

Do not use SAVEPICT with sprites.

This disastrous event can possibly be explained by reference to the way in which pictures are stored. (See also the details supplied in *LLL*, in the Appendix.)

If, after saving a picture (but *not a picture with sprites*) you CATALOG the disk, then you will notice two files with the picture name: they have file extensions .PIC1 and .PIC2. Both are binary files (copies of memory): the first is a copy of the actual memory used to store the screen display (.PIC1), and the second is a copy of colour memory, used to store information about the colours on the screen. If you print out a copy of a picture on to a printer, it is the first file that is used (printers tend to have a restricted colour sense).

When a picture with sprites is saved, the screen memory is saved, and all is well until the LOGO system attempts to save the contents of colour memory. Then, for some reason or other, the presence of sprites on the screen renders the system inoperative. That this is the case is suggested by reloading LOGO and CATALOGing the disk on which the picture was

saved: there is one file, that with the .PIC1 extension.

You are only rarely likely to wish to save a picture with sprites, and then the best solution is to hide all sprites, before saving the picture. It is always possible to read a picture (named XXX) off disk, without using READ PICT: you simply enter BLOAD "XXX.PIC1.

In a similar manner, it is possible to save the graphics screen — with sprites on-screen — by 'BSAVE "XXX.PIC1 8192 16384'. For the C64 LOGO memory map and the location of the graphics screen, and for details of BSAVE, see *LLL*, Appendix. Note that details are also given on the saving of specific sprite shapes.

It would be an interesting exercise to define two procedures: one, to load screen memory from a .PIC1 file; and, two, to save a .PIC1 file.

PART 2
Specific Applications

CHAPTER 7

Keyboard Control

This chapter discusses two major sets of routines: the first set of procedures emulate (more or less) those given for the joystick in Chapter 5; the second set provide a means by which the progress of the turtle can be more finely controlled.

Recognising the keyboard

The principal procedure for keyboard emulation of TURTLE is called, for phonetic reasons, TERTLE

```
TO TERTLE
    TERN MOVE TERTLE
END
```

where MOVE is as before, and TERN is a slightly altered TURN. For ease of reference, I repeat the definition of MOVE, as well as giving that for TERN.

```
TO TERN
    RT 45 * JOYK
END

TO MOVE
    FD 15
END
```

The difference between TERN and TURN comes with the substitution of JOYK for JOY, and both the procedures should be easily followed.

The new procedure has a definition

```
TO JOYK
    TEST RC?
    IFFALSE OUTPUT 0
```

```
    IFTRUE OUTPUT KEYREAD
END
```

in which there is a call to a procedure KEYREAD. In JOYK, a test is made to see if a key has been pressed (ie the LOGO primitive RC?): if a key has not been pressed, the procedure outputs 0, otherwise it outputs the value of the procedure KEYREAD.

KEYREAD is a clever procedure:

```
TO KEYREAD
    LOCAL "KEY
    MAKE "KEY RC
    IF :KEY = "0 STOP
    MAKE "KEY ( ASCII :KEY ) - 48
    TEST ALLOF :K > - 1 :KEY < 8
    IFTRUE OUTPUT :KEY
    IFFALSE OUTPUT 0
END
```

and requires rather more study because, for example, it converts characters read from the keyboard into special numbers.

To clarify the nature of the conversion of keyboard characters into coded numbers, enter a sequence

```
?PR ASCII "0
48
?PR CHAR 48
0
?
```

where the meaning of ASCII is that it provides the ASCII value for the character which follows. I have placed a quote mark (") in front of 0, but this is not necessary in this particular case (simply because 0 is a numeral). The quote mark is there to remind you that the 0 is a character, not a value.

The code number for the numeral "0 is 48, and this is presented as the information that the ASCII value of 0 is equal to 48. ASCII values are standard values (the American Standard Code for Information Interchange), and these values are used to enable you to talk to the C64 about numerals, letters, and symbols, in a coherent manner. To illustrate this coherence, try

```
?PR ASCII "E
69
```

```
?PR ASCII ""
34
?
```

— the ASCII code for E is 69, and the code for " is 34. To every character there is a code number and, therefore, for every code number there is a character.

The ASCII code number for a character is found by use of the LOGO primitive CHAR. The character corresponding to the code number 48 is 0, and the way in which this is discovered is via the procedure CHAR.

Here is a fun procedure to print out all the characters, and their corresponding code numbers, from 0 to 255. When the procedure has finished its antics, it leaves you not only in lower case but also with text of a different colour:

```
TO ASCPR
   ASC 255
END

TO ASC :N
   IF :N < 0 STOP
   ( PR :N CHAR :N )
   ASC :N - 1
END
```

The procedure ASCPR sets the start of the series at ASCII code 255, and the main production of code numbers is provided by the procedure ASC. Note that at times, as the numbers unfold, funny things happen to the screen, because some control characters are output. If you check with the C64 manual and Programmer's Reference Guide (under ASCII and video codes) you will find out more about these strange effects. Try some for yourself.

When you enter PR CHAR 19, for example, the cursor is 'homed' and at 'PR CHAR 147' the screen is 'cleared'. To clear the screen from within a LOGO procedure, therefore, you PRINT the CHARacter corresponding to the code number 147.

In the procedure KEYREAD, the keyboard reads a character and not a number. Reading a character is more generally applicable, because we do not want a procedure that can accept numbers only, in case a letter or symbol key is hit by accident. In order to generate numbers from the ASCII codes of characters, we subtract 48, as 48 is the ASCII code for the character "0.

The conversion of characters to their corresponding values will generate many differing codes, and, if the resulting numbers are between 0 and 9 (inclusive), then a numeric key has been pressed. Any converted value outside these limits means that a key other than a number key has been pressed.

Using the keyboard

Having examined the use of ASCII codes, we now have to see how their use helps us to control the turtle. Return to the definition of KEYREAD given above, and repeated here for convenience:

```
TO KEYREAD
    LOCAL "KEY
    MAKE "KEY RC
    IF :KEY = "0 STOP
    MAKE "KEY ( ASCII :KEY ) - 48
    TEST ALLOF :K > -1 :KEY < 8
    IFTRUE OUTPUT :KEY
    IFFALSE OUTPUT 0
END
```

and the KEY to the whole routine is the LOCAL object KEY. The object KEY is made to contain the character in the input buffer (that is, by use of RC, and if that character is equal to the character zero (that is, "0) then the routine stops. We finish the movement of the turtle, therefore, by pressing the numeric key 0.

After the conditional test, the object KEY is now made to contain the ASCII code corresponding to the character which was the previous content of the object (that is, :KEY), less the value 48. You may remember that 48 was the ASCII code for the character "0, thus — if a numeric key was pressed — the value stored in KEY runs from 0 to 9. The parentheses are placed around ASCII :KEY so that LOGO realises that this is a unit.

Investigate what this means:

```
PR ASCII A
THERE IS NO PROCEDURE NAMED A
?
```

which shows that, to refer to the name of A, one does not use A alone, one has to

```
PR ASCII "A
65
?
```

that is, 'quote' A to produce "A. Next try to

```
PR ASCII "A - 48
- DOESN'T LIKE A AS INPUT
?
```

and this means that the subtraction sign ' – ' behaves, in a sense, as a procedure, and is a procedure with two inputs (each side of the sign). As far as the subtraction procedure is concerned, the sequence '"A – 48' does not make sense because you cannot subtract a number from a letter. When the LOGO interpreter is trying to tease out the meaning of the line, it thinks that the input for ASCII is '"A – 48'. That is not correct; what was meant was 48 less than the ASCII code for A.

The next line is more hopeful:

```
?PR ( ASCII "A ) - 48
17
?
```

as long as the parenthesis ')' is separated from the "A; otherwise there is a different error (try it to see). The parentheses have grouped the expression in the way we want to make sense. We use the parentheses to group where LOGO would not otherwise group, in a similar manner to the way in which such parentheses group in, say, the PRINT command.

In the next line there is a complex TEST, which could be rewritten, to good effect, as

```
TEST ALLOF ( :KEY > - 1 ) ( :KEY < 8 )
```

where the parentheses group elements in the way LOGO would group, but make the grouping clearer. The LOGO procedure ALLOF has two inputs, where both inputs have to produce truth values: the result of the application of ALLOF can be illustrated by

```
?ALLOF "TRUE "TRUE
RESULT: TRUE
```

```
?ALLOF "TRUE "FALSE ; ALSO ALLOF "FALSE "TRUE
RESULT: FALSE
?ALLOF "FALSE "FALSE
RESULT: FALSE
?ALLOF TRUE TRUE ; NOTE CAREFULLY
THERE IS NO PROCEDURE NAMED TRUE
?
```

and we can, if we wish, produce a 'truth table' which shows the combinations of TRUE and FALSE and their results. Those used to other languages, and standard logical names, might recognise the ALLOF procedure as the logical AND. The result of ALLOF is only true if all of its inputs are true.

At this point you may care to try out similar tests of the C64 LOGO procedure ANYOF, which corresponds to a logical OR. This will enable you to see for yourself how the two commands differ.

Notice (in the final line of the ALLOF examination) that we have to refer to TRUE and FALSE by their names, that is, they have to be 'quoted'. The ALLOF procedure can have more than two inputs, as long as the entire sequence is enclosed in parentheses

```
?( ALLOF "TRUE "TRUE "TRUE )
RESULT: TRUE
?ALLOF "TRUE "TRUE "TRUE
YOU DON'T SAY WHAT TO DO WITH TRUE
?
```

If it is true that the value contained in the object KEY is not only greater than -1 but also less than 8, then that value is output to the calling procedure. The calling procedure is JOYK which then outputs to TERN, and then turns are made. If the result of the test is false, then the key pressed is out of bounds, and thus the value 0 is output by the procedure.

The test could be performed in a slightly different manner, using ANYOF:

```
TEST NOT ANYOF ( :KEY < 0 ) ( :KEY > 7 )
```

and I leave you to work out its consequences.

TOPLEVEL control

As I said earlier (in Chapter 5), use of the keyboard for controlling the turtle does have some attractions, and some people prefer to use the key-

board. If you examine **Figures 7.1, 7.2** and **7.3**, you can see how they are of a different nature to the joystick drawings — they somehow seem more precise. There is only one particular problem with TERTLE and that comes when the procedure is stopped by pressing the 0 key.

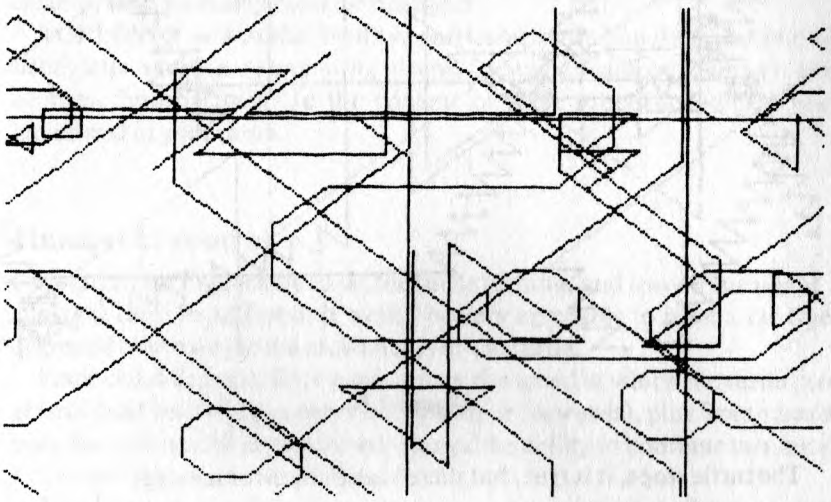


Figure 7.1

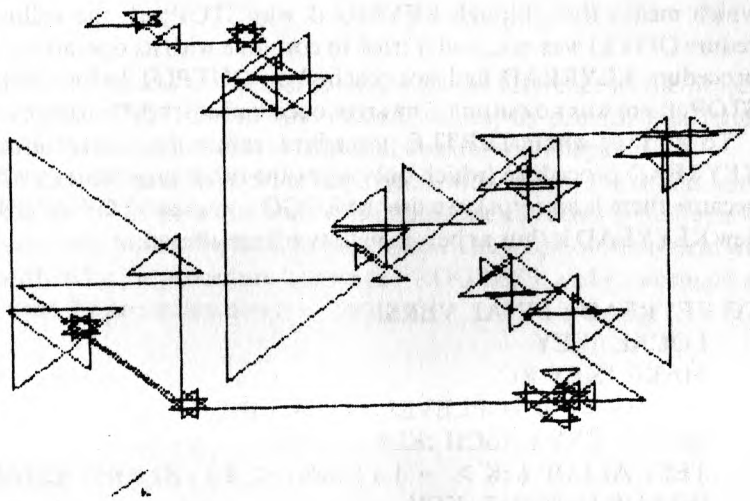


Figure 7.2

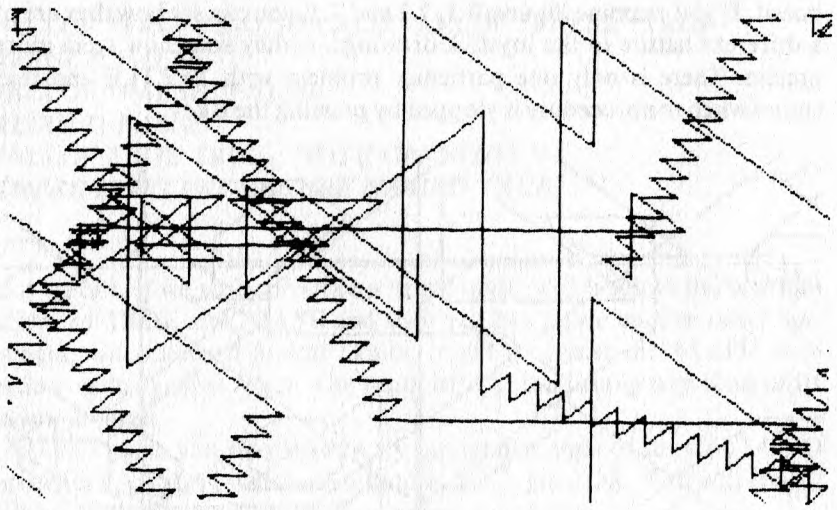


Figure 7.3

The turtle stops, it is true, but there is also the error message

KEYREAD DIDN'T OUTPUT, IN LINE
IFTRUE OUTPUT KEYREAD
AT LEVEL 79 OF JOYK.

which means that, though KEYREAD was STOPped, the calling procedure (JOYK) was not, and it tried to continue with its operation. As the procedure KEYREAD had not reached an OUTPUT before it met the STOP, there was no output. The error occurred at level 79 in my case.

To stop the whole TERTLE procedure, rather than just stopping the KEYREAD procedure (which only stops the other procedures by default because there is an error), we use the LOGO command TOPLEVEL. The new KEYREAD is thus as before with two lines altered:

```
TO KEYREAD ; FINAL VERSION
    LOCAL "KEY
    MAKE "KEY RC
    IF :KEY = "0 TOPLEVEL ; NEW LINE
    MAKE "KEY ( ASCII :KEY ) - 48
    TEST ALLOF ( :K > - 1 ) ( :KEY < 8 ) ; SLIGHT CHANGE
    IFTRUE OUTPUT :KEY
    IFFALSE OUTPUT 0
END
```

In the place of STOP we have TOPLEVEL, and TOPLEVEL instructs the LOGO interpreter to stop everything and jump to the top level of operation (that is, Level 0). Rather than jumping back to the previous level (as happens with STOP), TOPLEVEL stops everything, and it stops everything cleanly, with no annoying error messages.

TOPLEVEL is a useful facility, particularly for handling exceptional behaviour such as terminating deeply nested (usually recursive) procedures. Improvements to the content of these procedures are in your hands and in your mind.

Fineness of control

It would be very agreeable to set the turtle turning, and leave it turning at a constant rate. In addition, it would be very agreeable to have a far finer degree of control over the movements of the turtle.

Finer control means finer control over the speed at which the turtle progresses (and the ability to move backwards or forwards), plus finer control over the turning abilities of the turtle, and the ability to continue turning at a constant rate.

This fineness of control is summed up in one procedure, the procedure known as DO:

```
TO DO
    RT :ANGLE FD :DIST
END
```

in which the fineness of control is shown by the reference to the objects :ANGLE and :DIST, instead of constant values. By providing different values for the content of the two objects, we alter what happens.

Of course, we have to provide the objects with values, but that is not a problem, and — as we do not want to contaminate any other procedures — the objects will have to be local to the system. The scope of the system will be identified by the procedure known as PROGRESS, and so we might as well define the procedure now:

```
TO PROGRESS
    LOCAL "DIST
    LOCAL "ANGLE
    MAKE "DIST 0
    MAKE "ANGLE 0
    DECIDE
END
```

and we have used the classic LOGO technique of hiding something as yet unknown as a procedure: DECIDE is our secretive procedure this time.

DECIDE will have to be a procedure which accepts keyboard input and acts on the basis of that input, though how this is to be performed is as yet unknown to you (or me). When I designed this system, and had come to the conclusion that the crucial procedure was to be DO, I still had no idea what was to come next: PROGRESS came next.

I felt that I had better clear up how DIST and ANGLE were given their initial content,and what that initial content was. PROGRESS grew out of that desire, but I still did not know what was to follow, because I was not even sure what keys were to be recognised as being relevant. In such cases, minimise the potential for too great complication by modularising (that is, by using procedures as much as possible).

Keyboard analysis

I knew that, if no key had been pressed, the turtle was to continue on its journey uninterrupted, where by 'uninterrupted'. I meant that the content of ANGLE and DIST were to remain the same. This meant that the procedure DECIDE wrote itself:

```
TO DECIDE
    LOCAL "KEY
    TEST RC?
    IFFALSE DO
    IFTRUE KEYPRESS
    DECIDE
END
```

and — as the turtle movements repeated — the procedure became tail recursive.

Though it was not actually used in the body of the definition, I felt sure that somewhere there would need to be a local object called, say, KEY. I put one in. The first thing seemed to be to test for a key being pressed, and so there was TEST RC?. It was clear that, if a key had not been pressed, then the procedure DO was activated — after all, nothing had changed as far as ANGLE and DIST were concerned.

That bit was easy, the next bit was hard. What to do if a key had been pressed? I did not know, and did not want to mess up my procedure working it all out. Thus I modularised (that is, put off the awful day) and I said that if a key had been pressed then call KEYPRESS. A classic case of leave it until later (but it did not go away).

After all that, all there was left to be done was DECIDE again.

Still KEYPRESS had to be faced, and it was only then that I began to consider what keys I was to use, and why. I fancied using the J L keys to produce more of a turn left and right respectively, as those keys are popular keys, and well-placed. To speed (or slow) the turtle I decided to use the I M keys as they were well-situated with respect to the J L keys: the four keys made a little cross.

Near to the four keys there was P, and so I used that key to stop operations; to press P stopped the turtle. Well away, so that it could not be pressed by accident, was the Z key, and this is how the travelling was terminated. After all that explanation, the content of KEYPRESS is obvious.

```
TO KEYPRESS
    MAKE "KEY RC
    IF :KEY = "Z TOPLEVEL
    IF :KEY = "P KEYWAIT
    IF :KEY = "J MAKE "ANGLE :ANGLE - 1
    IF :KEY = "L MAKE "ANGLE :ANGLE + 1
    IF :KEY = "I MAKE "DIST :DIST + 1
    IF :KEY = "M MAKE "DIST :DIST - 1
    DO
END
```

The local object KEY contains the character in the input buffer: if it is Z, we stop everything by returning to TOPLEVEL; if the key is P, then we wait (by use of a procedure KEYWAIT); and the other keys add or subtract units from the size of the content of the local objects ANGLE SIZE. Then one DOes: if any other key has been pressed, the procedure does not take any notice.

What about KEYWAIT? Here it is, and I leave you to work out how it operates...

```
TO KEYWAIT
    IF NOT RC? THEN KEYWAIT ELSE MAKE "KEY RC
END
```

Figures 7.4, 7.5, 7.6 and **7.7** show the many different types of effect possible using PROGRESS — it takes a bit of practice, though, to arrive at my peak of excellence....

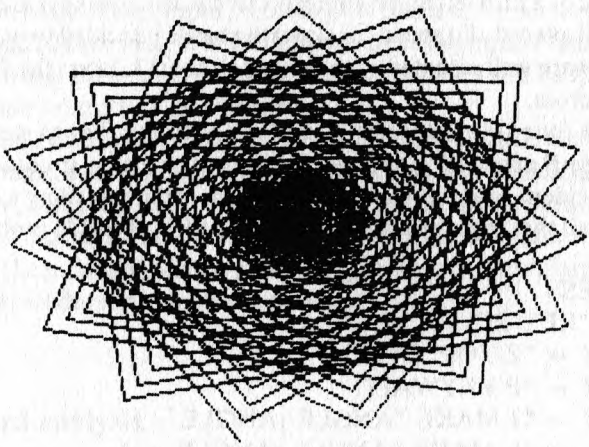


Figure 7.4

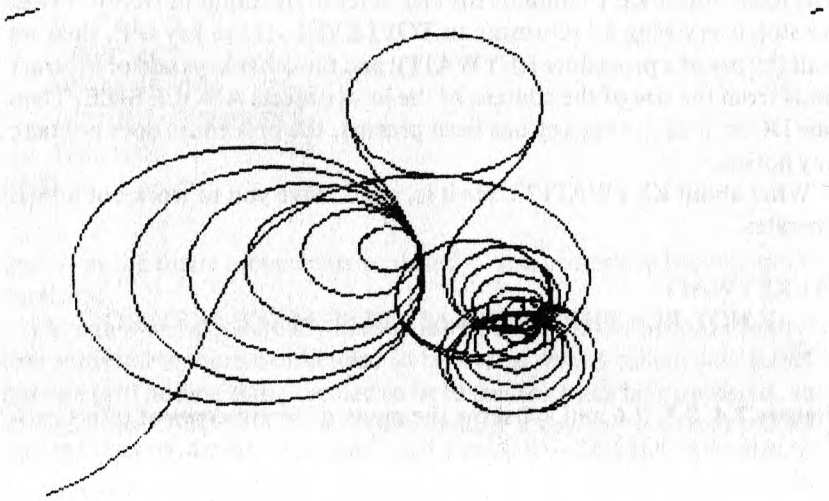


Figure 7.5

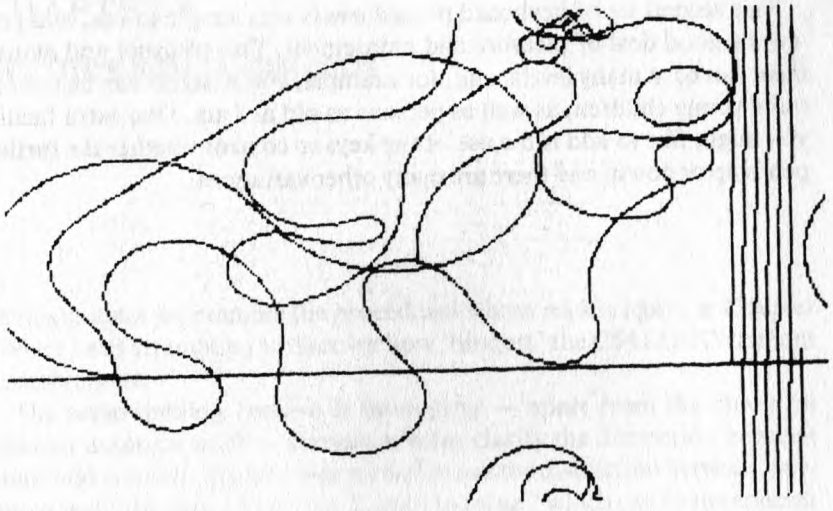


Figure 7.6

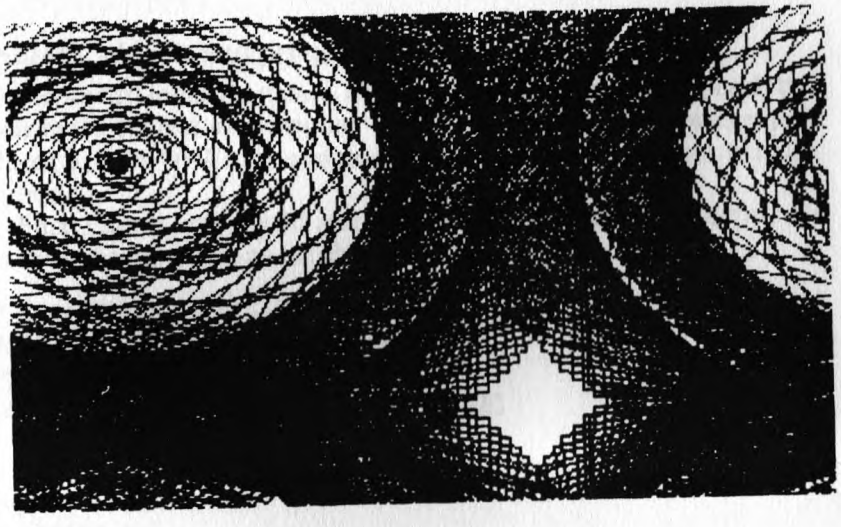


Figure 7.7

This second set of keyboard procedures is very simple to use, and provides a good deal of pleasure and amusement. This pleasure and amusement can be at many levels, and, for example, PROGRESS can be used by quite young children, as well as persons as old as I am. One extra facility you might like to add is the use of the keys to control whether the turtle's pen is up or down: and there are many other variations.

CHAPTER 8

Names and Content

In this chapter we examine the procedures whose results I gave in Chapter 4 when I was attempting to discover how 'random' the C64 LOGO random numbers were.

The programming exercise is interesting — apart from the check on random numbers itself — because it helps clarify the distinction between name and content. In particular it emphasises the distinction between, say, the name of the object known as **2**, and the value *2* which can be the content of any object: the object **2**, as we shall see, need not have the content *2*.

Note that, for this chapter alone, we will use a new convention to distinguish between object, procedure, and content. The convention only applies to the printed text, and is not relevant to the listings of procedures and happenings. Any object is shown in **heavy type**, any procedure name is shown in ***italic heavy type***, and any value (or content) is shown in *italic type*.

Names, objects and procedures, revisited

When we first looked at names, objects, and procedures (in Chapter 3) this was in the context of explaining the meanings of various messages produced by LOGO. We are now going to examine the distinctions in much finer detail.

First, initialise the LOGO system by ***GOODBYE***, and note that we are talking of a procedure name. Enter this sequence:

```
?.CONTENTS
RESULT: [STARTUP FALSE TRUE]
?MAKE "A "B .CONTENTS
RESULT: [B A STARTUP FALSE TRUE]
?"A
RESULT: A
?:A
RESULT: B
?A
THERE IS NO PROCEDURE NAMED A
?
```

and then provide this definition

```
TO A
    PR [PROCEDURE A]
END
```

which allows us to write some funny THINGs (?):

```
?.CONTENTS
RESULT: [PROCEDURE B A STARTUP FALSE TRUE]
?"A
RESULT: A
?:A
RESULT: B
?THING "A
RESULT: B
?A
PROCEDURE A
?
```

Thus we now have an object **A** with a content *B*, and a procedure *A* whose content is *PR* [*PROCEDURE A*].

Note that ***THING*** **"A** outputs the content of the object **A**, and is equivalent to *:A*. ***THING*** has uses, for example, when we ask for the content of an object named as the content of an object. The use of this facility will become apparent, but as a small step try

```
?MAKE "A "B
?MAKE "B "C
?THING "A
RESULT: B
?:A
RESULT: B
?THING THING "A
RESULT: C
?THING :A
RESULT: C
?::A
THERE IS NO NAME :A
?
```

There are many consequences of the use of ***THING***, some of which appear in the random number checking procedures.

If we simply present LOGO with the unadorned letter 'A', the LOGO

interpreter looks through its list of procedures, and finds *A* whose content is *PR [PROCEDURE A]*. If LOGO finds the letter A with a prefix, it either thinks that you are referring directly to the letter as the name of some object (ie **"A**) or to the content of the object (ie *:A*). The content of the object is *B*.

When we come to numbers, things are rather different, and sometimes confusingly so.

Re-initialise the system, and try the following:

```
?.CONTENTS
RESULT: [STARTUP FALSE TRUE]
?1; LINE 1
RESULT: 1
?.CONTENTS
RESULT: [STARTUP FALSE TRUE]
?MAKE "1 "B
?.CONTENTS
RESULT: [B 1 STARTUP FALSE TRUE]
?:1
RESULT: B
?TO 1 ; LINE 2
CAN'T TO 1
?
```

and it is possible to see the difference between the simple letter A and the simple number 1. It is also possible to see the similarities.

If we start with the similarities, we can see that the object **"1** can have the same content as **"A**, that is, they can both contain the name *"B*. In this sense, therefore, the two objects are of the same nature: thus **1** is only another object. Or is it?

Take **Line 1** — in the case of the equivalent line using 'A' by itself without modification of any nature, the response is 'THERE IS NO PROCEDURE NAMED A'; whereas in the case of '1', the response is 'RESULT: 1', yet '1' does not appear in the .CONTENTS list. The LOGO interpreter recognises that '1' is a number, and a number is its own result. Another way you may care to consider the special nature of '1' is as a primitive procedure *1* which outputs the value *1*.

This might explain why it is impossible to create a procedure *1* by conventional methods **(Line 2)**, though, if you try to redefine another primitive such as, say, ***FD***, the error message is different. Try it.

Thus the numbers are special LOGO items which are somewhere between LOGO primitives, and LOGO objects. This means that sometimes we have to be careful of how numbers are treated.

As a final experiment, how about

```
?MAKE "FD "B
?:FD
RESULT: B
?
```

and thus we see that any LOGO primitive such as ***FD*** can be used as a LOGO object **FD**, and contain a value (in this case, ***B***), and so '1' is best seen as a LOGO primitive procedure. Mind you, I can make something really marvellous:

```
?X
THERE IS NO PROCEDURE NAMED X
?MAKE "X "B
?TO X ; AND HERE IS A DEFINITION
    OUTPUT "X
END
?X
RESULT: X
?:X
RESULT: B
?
```

because I have defined a procedure *X* such that, when used, *X* OUTPUTs *X*. X now behaves very much like 1.

Checking random numbers

A sequence of characters forms a LOGO word, and to make a word from a sequence of characters the LOGO primitive ***WORD*** is used. Try the following, being very careful to leave a space between the final letter and the parenthesis ')'.

```
?WORD "XX "9999
RESULT: XX9999
?( WORD "XX "9999 "ZZ )
RESULT: XX9999ZZ
?( WORD "XX )
RESULT: XX
?
```

This shows how sequences of characters can be put together. One important characteristic to note is the use of parentheses to group primitives which can have optional numbers of parameters.

As there is this possible ambiguity in LOGO, as to whether '1' is **1**, ***1***, or *1*, the random checking routines use ***WORD*** to make the status clear when **1** is what is meant. Here are the routines:

```
TO ADDUP :X
    LOCAL "N
    MAKE "N RANDOM :X
    MAKE ( WORD :N ) ( 1 + THING ( WORD :N ) )
END

TO CLEAR :N
    MAKE ( WORD :N ) 0
    IF NOT :N = 0 CLEAR :N - 1
END

TO CHECK :NUM :REP
    CLEAR :NUM - 1
    REPEAT :REP [ADDUP :NUM]
END
```

and I will leave you to work out the full implications of these routines. Not to be totally unfair, however, I will explain that in ***CHECK :NUM :REP***, the first parameter is the number of categories for the random number, and the second parameter gives the number of trials.

I will also explain ***CLEAR*** **:N**. The parameter gives the number of random categories (which will run from **0** to **:N − 1**), and the assignment 'MAKE (WORD :N) 0' corresponds to

```
?MAKE ( WORD 10 ) 0
?:10
RESULT: 0
?
```

which, though in a sense ***WORD*** is redundant, is acting on the safe side. Running the procedures without ***WORD*** can — at times — produce strange arithmetic, so play safe. To run a check for 10 categories (from 0 to 9), with 1000 tries, we enter 'CHECK 10 1000', and to check on results we enter 'PO NAMES'.

CHAPTER 9

Aspects of Graphical Design

In the production of statistical graphical presentations, we have to know a fair amount about how the graphical display of shapes in C64 LOGO can be managed. It is a rare statistical display that can wrap round: thus we need to know far more about the dimensions of the screen in terms of graphical units.

In this chapter, therefore, we will study the design of one shape in detail, based on the Greek letter sigma. This exploration will help us to find out a fair amount about the graphical display in C64 LOGO. At the end of this chapter you should be able to draw a square which actually looks square...

Drawing lines again

In many more complex statistical presentations the injunction to 'add up' is given by use of the Greek capital letter sigma. The shape of sigma is shown in **Figure 9.1**, and the shape was drawn using a LOGO routine.

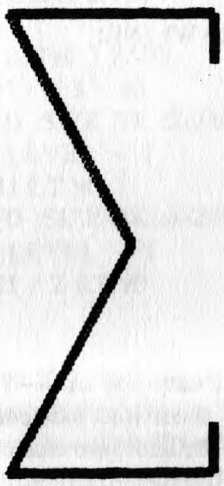


Figure 9.1

Contrary to most of the good advice given by myself and others, I started the routines to draw a sigma from the basic building block — a thick line. I do not believe in rules for the sake of rules, and in this case I wanted to sort out the lines, good thick lines.

I started with the idea of drawing a thick line by using four ordinary thin lines next to each other. All the lines were to be parallel to each other, in the direction I will call the 'main axis'. The endpoint of the main axis was at the starting point of the turtle's travels.

The turtle was to start at the endpoint and then draw two pairs of lines either side of the main axis (in the correct direction, of course). Each pair of lines could be regarded as having its own 'central axis', where both central axes were parallel to the main axis. The starting point of the central axis of each pair of lines was to be 1 unit away from the start (left and right). The two pairs of lines were identical in as far as the drawing operation was concerned, but each pair was initiated from either side of the main axis.

If we assume that we commence the drawing of each pair of lines at the endpoint of the pair's central axis, to reach that endpoint for the first pair, we turn to the left through 90 degrees, and move one unit forward. To face the turtle in the correct direction along the central axis, we then turn the turtle through 90 degrees to the left.

Assuming that drawing the two lines leaves us at the same place, with the same heading, to move to the endpoint of the other central axis, the moves are: a 90 degree turn to the right; then a move of two units forward; and a 90 degrees turn to the left will leave the turtle facing in the appropriate direction. We then draw the second set of two lines, turn left through 90 degrees, move forward one unit, and turn through 90 degrees to the right to face in the original direction.

The procedure is called DRAWLINE:

```
TO DRAWLINE :SIZE
    LT 90 FD 1 RT 90
    TWO.LINE :SIZE
    RT 90 FD 2 LT 90
    TWO.LINE :SIZE
    LT 90 FD 1 RT 90
END
```

and, as ever, I have taken the easy way out — I have not drawn the two lines, I have merely wished them into existence by means of the call TWO.LINE :SIZE. Wishing the lines into existence does not work unless we give the correct words for the spell to LOGO.

Really, drawing two lines is almost like drawing the two pairs of lines, but with thinner lines, and the tops of the lines joined together. The :SIZE

remains the same, but all the other distances are halved, thus

```
TO TWO.LINE :SIZE
    LT 90 FD 1 / 2 RT 90
    FD :SIZE
    RT 90 FD 1 RT 90 ; NOTE THAT THE TOPS ARE JOINED
    FD :SIZE
    RT 90 FD 1 / 2 RT 90
END
```

and, though it is slower, here is another way in which this procedure could be written, which shows the structural similarity between the procedures:

```
TO TWO.LINE :SIZE ; NEW VERSION, COMPARE DRAWLINE
    LT 90 FD 1 / 2 RT 90
    FD :SIZE BK :SIZE ; IMPORTANT CHANGE
    RT 90 FD 1 LT 90
    FD :SIZE BK :SIZE ; IMPORTANT CHANGE
    LT 90 FD 1 / 2 RT 90
END
```

This congruence has potential.

Recursive line drawing

Try to work out how this procedure works (it is an example of controlled recursion), and then attempt to use it.

```
TO DLINES :SIZE :WIDTH :LEVEL
    LT 90 FD :WIDTH / 2 RT 90
    IF :LEVEL = 0 FD :SIZE BK :SIZE ELSE DLINES :SIZE
        :WIDTH / 2 :LEVEL - 1
    RT 90 FD :WIDTH LT 90
    IF :LEVEL = 0 FD :SIZE BK :SIZE ELSE DLINES :SIZE
        :WIDTH / 2 :LEVEL - 1
    LT 90 FD :WIDTH / 2 RT 90
END
```

If you enter DLINES 100 50 3 and then watch, you will see a very active turtle, drawing a comb with 16 teeth. Enter 'RT 180 DLINES 100 50 4', and the comb is the other way up — with 32 teeth (count them). The new comb has exactly the same dimensions as that previously drawn (50 * 2 = 100 wide and 100 high).

If you now try 'DRAW DLINES 100 50 5 RT 180 DLINES 100 50 6', the

first comb has 64 teeth (and you cannot count them) which almost fully fill the rectangle/square; the second comb has 128 teeth, and fills in the shape with solid colour. 128 lines within the space of 100 units, is distinct overkill (there are 28 lines too many). I will not explain how DLINES works, but I suggest you try using TRACE, and a small value for the :LEVEL parameter. Remember to use NOTRACE to stop the information once you have had enough.

I also suggest you try 'DLINES 50 1 0' and 'DLINES 50 2 1', to compare with the results of DRAWLINE. Those who investigate this procedure will learn greatly useful information about the applicability of recursion.

Something for you to try is to create a procedure 'TO FILL.BOX :HEIGHT :WIDTH'. The procedure will take the desired 'width' of the box, calculate the :LEVEL necessary to ensure that the box is filled (as in the case of DLINES 100 50 6), and then call DLINES with the appropriate parameters.

Of course, you might think of a different way. Look at the next chapter but one, when you have produced your version.

Half a sigma

As you can see, the Greek sigma is a symmetrical letter: so I will only construct routines for the upper half, and then modify those routines for the lower half. The best place to start drawing a symmetrical shape is at the centre, and, therefore, when the half is complete, that is where the turtle will return.

The first line to be drawn is thus that extending from the centre, to the left. The inclinations of the other two lines (that is, the top line, and the short line at right angles to the top line) are easy to establish, but the angle for the first line is more problematic. We could try many different angles, but that could become rather tiresome.

The solution is work out how far forward, and how far to the side, the line is to go. In this example I think two units forward, and one to the side seem about right. How, therefore, are we to use this belief? If you examine **Figure 9.2**, the implication is that the Opposite side is half the length of that of the Adjacent side. To find the angle in the apex enclosed by the Opposite side and the Hypotenuse is our task.

We find this angle by use of the primitive ATAN. The tangent of the angle enclosed by A and H is given by O/A, and this value we know to be 1/2. In many computer languages (but not C64 LOGO), if we know the value of an angle we can call a function which gives the tangent of the angle (usually shortened to 'tan').

As the preservation of angles is very important to all the rest of the routines given in this chapter, use the C64 LOGO primitive .ASPECT.

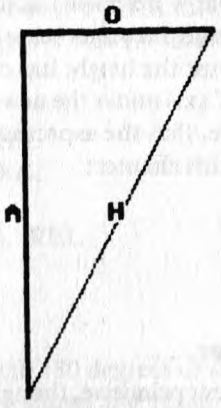


Figure 9.2

Normally the dimensions of the LOGO plotting screen vary from −160 to +159 along the X axis, and from −129 to +130 along the Y axis. These figures contradict those given in *LLL* (in the Computation chapter), where the limits are given as −155 to +155 for X, and −120 to +120 for Y. In fact, *LLL* is deficient in information about the admissable dimensions.

The maximum dimension for the Y axis can be established by DRAW HT FD 130 and the line reaches the top of the screen. A further FD 1 produces a tiny dot at the bottom of the screen, which indicates that the maximum height has been exceeded, and wrapround has occurred. The minimum can be established by DRAW HT RT 180 FD 130 which produces a tiny wrapround at the top of the screen.

The dimensions of the X axis can be established by DRAW HT RT 90 FD 159 followed by FD 1, and then DRAW HT LT 90 FD 160 plus FD 1.

The range of units for the X axis is thus 320, and the range of units for the Y axis is 260 and this can be compared to the number of characters on the screen (40 characters across by 25 lines). It can be seen that eight graphical units (on the X axis) are equivalent to one character width, but 10.4 graphical units (on the Y axis) are equivalent to one character height. If the height of one character was also eight units then the limits on the Y axis would be −99 to +100. (Work out the reason why.)

In *LLL* (Appendix) the normal value for the primitive .ASPECT is given as .768, and note that

```
?199/259
RESULT: 0.768339
?
```

and thus the default value of .ASPECT is equal to the limits for eight graphical units per character height, relative to the existing limits. Try to

draw a square, and see how it appears as a rectangle. Change the relative scaling by .ASPECT 1, and draw that same square: it looks 'squarer' — the width has not altered, just the height has changed. It is now up to you to check the limits to the Y axis under the new aspect ratio.

We assume, therefore, that the aspect ratio has been altered to 1 in the routines for the rest of this chapter:

```
?.ASPECT 1
?
```

Half a sigma is better...

C64 LOGO has no tangent primitive, though, by use of SIN (sine) and COS (cosine), a procedure can be so defined — see *LLL*, Computation. C64 LOGO does, however, have an arctangent function ATAN, where the ATAN primitive is used to find the value of an angle when we know the relative size of the sides of a triangle. Try this sequence

```
?DRAW TWO.LINE ; DRAWS A SLIGHTLY THICK LINE
?ATAN 1 2 ; OPPOSITE AND ADJACENT RELATIVE SIZES
RESULT: 26.5652
?RT ATAN 1 2 ; EQUIVALENT TO RT 26.5652
?TWO.LINE 50 ; DRAW THE LINE TO MAKE THE
        DIRECTION CLEAR
?
```

to find that the line seems to be in about the correct direction.

Enough of angles, now to the drawing of half of a sigma.

```
TO HALF.SIGMA :SIZE :RATIO
    LOCAL "X MAKE "X XCOR
    LOCAL "Y MAKE "Y YCOR
    LOCAL "H MAKE "H HEADING
    LOCAL "ANGLE MAKE "ANGLE ATAN 1 2
    LT :ANGLE DRAWLINE :RATIO * :SIZE FD :RATIO *
        :SIZE
    RT (90 + :ANGLE) DRAWLINE :SIZE FD :SIZE
    RT 90 DRAWLINE :SIZE / 4
    PU SETXY :X :Y SETH :H
END
```

As I believe in the benefits of study, study this routine to work out why it works, and what the mysterious :RATIO parameter is. To assist slightly, I give you **Figure 9.3** which accentuates a few interesting angles. Two

HALF.SIGMAs combined produce a FULL.SIGMA, so all that is necessary is to turn through 180 degrees between drawing the two HALF.SIGMAs:

```
TO FULL.SIGMA :SIZE :REL
    HALF.SIGMA :SIZE :REL
    RT 180
    HALF.SIGMA :SIGMA :REL
    RT 180
END
```

and at the end is a turn through 180 degrees to set the turtle facing in the original direction. **Figure 9.4** shows the result of the sequence FULL-.SIGMA 60 4 / 3, and it is wrong.

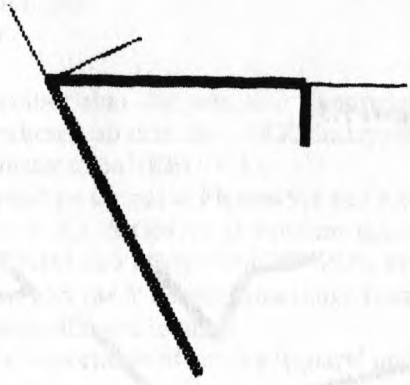


Figure 9.3

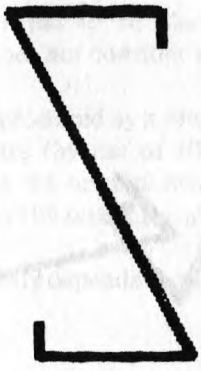


Figure 9.4

Quite simply, the two halves are not only rotated through 180 degrees, they are also turned right to left and vice versa. A mistake, but not a failure — rather a chance to improve my understanding. Looking at the original HALF.SIGMA, which works correctly, I need to turn all the RTs into LTs and all LTs to RTs.

I know that, for example, RT :ANGLE is the same as LT (– :ANGLE), so, to reflect the shape, all that is needed is to make the angles negative. If I am going to make angles negative, I want to be able to choose when I make them minus — I need some 'switch' which I can set to determine which way round the shape is. Here are the new procedures

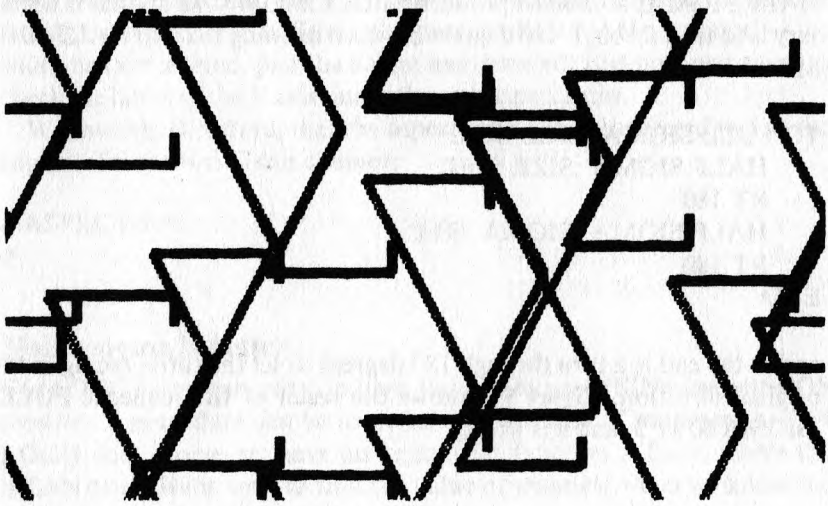


Figure 9.5

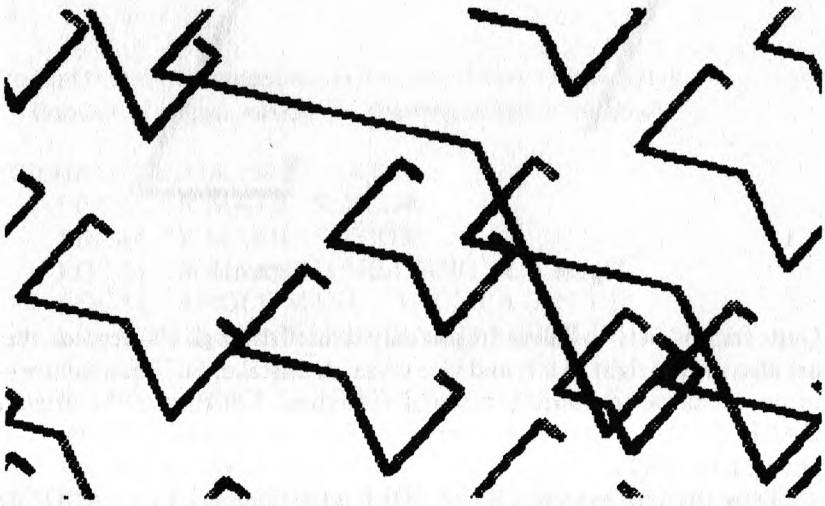


Figure 9.6

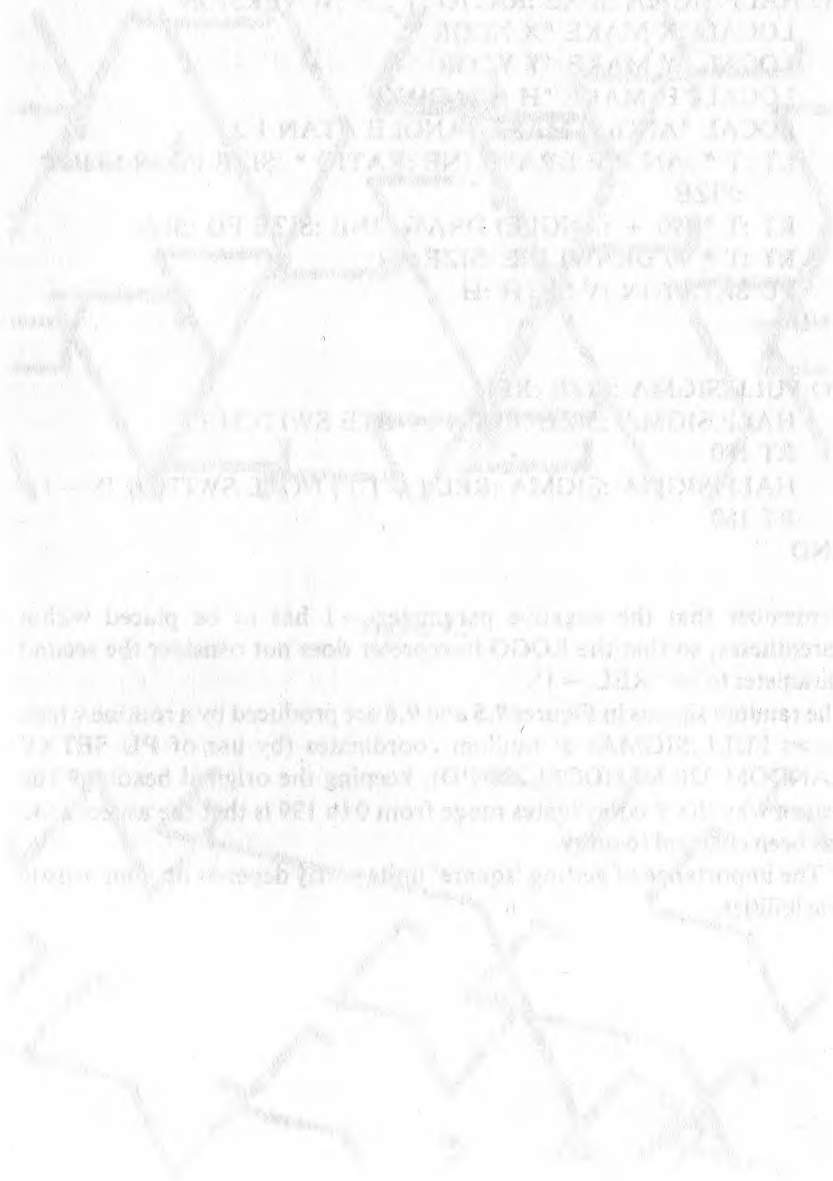
```
TO HALF.SIGMA :SIZE :RATIO :T ; NEW VERSION
    LOCAL "X MAKE "X XCOR
    LOCAL "Y MAKE "Y YCOR
    LOCAL "H MAKE "H HEADING
    LOCAL "ANGLE MAKE "ANGLE ATAN 1 2
    LT :T * :ANGLE DRAWLINE :RATIO * :SIZE FD :RATIO *
        :SIZE
    RT :T * (90 + :ANGLE) DRAWLINE :SIZE FD :SIZE
    RT :T * 90 DRAWLINE :SIZE / 4
    PU SETXY :X :Y SETH :H
END

TO FULL.SIGMA :SIZE :REL
    HALF.SIGMA :SIZE :REL 1 ; NOTE SWITCH IS 1
    RT 180
    HALF.SIGMA :SIGMA :REL ( -1 ) ; NOTE SWITCH IS -1
    RT 180
END
```

Remember that the negative parameter −1 has to be placed within parentheses, so that the LOGO interpreter does not consider the second parameter to be ':REL − 1'.

The random sigmas in **Figures 9.5** and **9.6** are produced by a routine which draws FULL.SIGMAs at random coordinates (by use of PU SETXY RANDOM 320 RANDOM 200 PD), keeping the original heading. The reason why the Y coordinates range from 0 to 199 is that the aspect ratio has been changed to unity.

The importance of getting 'square' units partly depends on your artistic sensibilities.



CHAPTER 10

Simple Statistical Programming

The ultimate objective of this chapter is to develop procedures to calculate the means, standard deviations, the Pearson correlation between, and regression equation for, two statistical variables. In addition, procedures are suggested to draw the scattergram of values and produce the regression line on the same display.

In the process of performing these calculations, we will be involved in certain more advanced LOGO features, including the manipulation of lists. It is only fair to warn the reader that parts of this chapter are fairly technical, but one reason for its inclusion is to show how to tackle a more complex numerical application in LOGO. The drawing of the scattergram (at the end of the chapter) is a useful technique whose application goes beyond that of statistics.

The calculation of regression and correlations using LOGO has been chosen because, as far as I am aware, such details are unavailable elsewhere.

The first thing to learn in statistics is how to add numbers together — in computational terms, statistics is little more than the addition of values. For those interested, the importance of summation is explained at greater length in my *Pocket Guide to Statistical Programming* (Pitman — it has equivalent BASIC routines for those who wish to make the comparison).

First, however, the idea of summation.

Summation and precedence

In statistics, the addition of a list of numbers is the most frequent operation to be performed. In LOGO the key way of storing data is by use of the list, and so statistics using LOGO will involve lists of information and the utilisation of operations on lists.

First, a procedure to calculate the sum of all the elements in a list,

```
TO SUM :INLIST
    IF NOT ( COUNT :INLIST ) = 1 OUTPUT ( FIRST
        :INLIST + SUM BUTFIRST :INLIST ) ELSE OUTPUT
        FIRST :INLIST
END
```

which works by OUTPUTting the value of the FIRST plus the SUM of the rest of the list (all items BUTFIRST). If there is only one item left in the list, then that is the value which is OUTPUT. Should work, should it not? Find out...

```
?SUM [1 2 3 4 5 6]
+ DOESN'T LIKE [5 6] AS INPUT, IN LINE
IF NOT ( COUNT :INLIST ) = 1 OUTPUT ( FIRST :INLIST +
        SUM BUTFIRST :INLIST ) ELSE OUTPUT FIRST
        :INLIST
AT LEVEL 5 OF SUM.
?
```

and it all seems very strange. What does it all mean?

As this type of confusion is very common in LOGO, and arises from the workings of the LOGO interpreter, it is very important to try to analyse the full import of the error message. The (infix) arithmetical operator '+' expects two numbers either side (it is called 'infix' because it is fixed in-between two 'operands'). We have come across this confusion before, in Chapter 5, and have seen how the use of parentheses can clarify our intent.

A telling argument against using infix operators is the confusion produced by the interpreter, as shown in SUM. Note that the mistake in SUM was genuine, and was not a 'pretend' error — this mistake was a real Boris Boob. The LOGO interpreter has a very strict idea about priority — not always what we think the order is (and I include 'me' in 'we'). The explanation of the meaning behind the error message can be more clearly shown by this short session:

```
?FIRST [3 4]
RESULT: 3
?FIRST [3 4] + 2
+ DOESN'T LIKE [3 4] AS INPUT
? ( FIRST [3 4] ) + 2
RESULT: 5
?
```

In the initial line, the FIRST item of the list [3 4] is output, with the result that the value is 3. In the second line, the sequence is the same, other than that 2 is added to FIRST [3 4]. The difference in outcome is tremendous, because the infix operator '+' tries to add [3 4] to 2, and '+' does not like a list as an input. The infix operator takes precedence over the 'prefix' operator FIRST. If the prefix operator and inputs are enclosed in parentheses, then they are protected from the ravages of the infix operator (third line).

The solution to the problem with SUM is very simple, once you know how:

```
TO SUM :INLIST
    IF NOT ( COUNT :INLIST ) = 1 OUTPUT ( ( FIRST
        :INLIST ) + SUM BUTFIRST :INLIST ) ELSE OUTPUT
        FIRST :INLIST
END
```

all we have to do is enclose the list and prefix operator in parentheses, (FIRST :INLIST).

There is no potential for error, or at least less potential for error, if we define a prefix operator to ADD, plus a further procedure EQUAL?.

```
TO ADD :N1 :N2
    OUTPUT :N1 + :N2
END

TO EQUAL? :P1 :P2
    OUTPUT :P1 = :P2
END
```

These two procedures allow us to rewrite SUM without the use of any parentheses — the use of prefix operators removes possible confusions, as there is no need to calculate precedence, and so parentheses are not required.

Here is the revised procedure, using prefix operators alone, and TEST.

```
TO SUM :INLIST
    TEST NOT EQUAL? COUNT :INLIST 1
    IFTRUE OUTPUT ADD FIRST :INLIST SUM BUTFIRST
        :INLIST
    IFFALSE OUTPUT FIRST :INLIST
END
```

and, though at first it seems strange to use the prefix operators/procedures, it is rather more consistent.

The sigma procedure

Summing information for two lists of variables is somewhat more complex: for example, the items in each list are paired, multiplied together, and all the products are then summed. The summation operation in statistics is known as the sigma function, and so SIGMA is exactly the same as SUM. How are we to change the name?

Watch

```
?TEXT "SUM
RESULT: [[:INLIST] [TEST NOT EQUAL? COUNT :INLIST 1]
        [IFTRUE OUTPUT ADD FIRST :INLIST SUM
        BUTFIRST :INLIST] [IFFALSE OUTPUT FIRST
        :INLIST]]
?DEFINE "SIGMA TEXT "SUM
?PO SIGMA
TO SIGMA :INLIST
TEST NOT EQUAL? COUNT :INLIST 1
IFTRUE OUTPUT ADD FIRST :INLIST SUM BUTFIRST :INLIST
IFFALSE OUTPUT FIRST :INLIST
END
?ER SUM
?
```

The primitive TEXT, when applied to a quoted word, gives the definition of the procedure known by that name. To ask for the definition of a procedure, where the name given is not that of a procedure, results in the empty list '[ ]'. The new procedure DEFINE takes a list as input, and makes the named word refer to a procedure with that definition. ER SUM erases the procedure name SUM. Unfortunately, as the astute will have noticed, the routine refers to SUM within the body of the definition, and so you will have to use ED (or EDIT) to alter that word. We end by producing

```
TO SIGMA :INLIST
    TEST NOT EQUAL? COUNT :INLIST 1
    IFTRUE OUTPUT ADD FIRST :INLIST SUM BUTFIRST
        :INLIST
    IFFALSE OUTPUT FIRST :INLIST
END
```

The TEACH system on the utility disk (*LLL*, Appendix) uses the DEFINE facility, and the three procedures are well worth studying. In addition to the use of DEFINE, there are interesting uses of PRINT1 (print without ending with a carriage return) and REQUEST (reads in a line of typing, terminated by a RETURN). Use PO to study the contents of the procedures in question.

Combining lists

Actually, as can be clearly seen, the strange-looking statistical operator sigma (examine the previous chapter to find what it looks like) acts in exactly the same way as summing a list. The next stage is to find a way to form a combined list, but what kind of combined list? One example is this

LIST 1	LIST 2	COMBINED LIST
A	Z	A # Z
B	Y	B # Y
C	X	C # X
D	W	D # W

where both lists continue indefinitely, as does the combined list. The # sign indicates some infix operator (such as + − * /) and the values of the items in the combined list will vary depending on the operator.

Follow the sequence:

```
?2 * 3
RESULT: 6
?[2 * 3]
RESULT: [2 * 3]
?RUN [2 * 3]
RESULT: 6
?( SE "2 "* "3 ) ; REMEMBER THE SPACE BETWEEN 3 AND )
RESULT: [2 * 3]
?RUN ( SE "2 "* "3 )
RESULT: 6
?MAKE "P1 2 MAKE "P2 3 MAKE "OP "*
?( PR :P1 :OP :P2 )
2 * 3
?( SE :P1 :OP :P2 )
RESULT: [2 * 3]
?RUN ( SE :P1 :OP :P2 )
RESULT: 6
?
```

which — if studied carefully — should help you to understand the workings of the procedures INOP and COMBINE. Remember that OP is the short form of OUTPUT, and that BF is the short form of BUTFIRST.

```
TO INOP :P1 :OP :P2
    OUTPUT RUN ( SE :P1 :OP :P2 )
END

TO COMBINE :OPER :L1 :L2
    TEST NOT EQUAL? 1 COUNT :L1
    IFTRUE OP ( SE INOP FIRST :L1 :OPER FIRST :L2
        COMBINE :OPER BF :L1 BF :L1 BF :L2 )
```

```
    IFFALSE OP INOP FIRST :L1 :OPER FIRST :L2
END
```

and the simplest way to investigate the workings of COMBINE is first to try

```
?COMBINE "* [1 2 3][1 2 3]
RESULT: [1 4 9]
?TRACE COMBINE "* [1 2 3][1 2 3]
TRACING ON
EXECUTING COMBINE * [1 2 3][1 2 3]
TEST NOT EQUAL? 1 COUNT :L1
EXECUTING EQUAL? 1 3
OUTPUT :P1 = :P2
OUTPUT: FALSE
ENDING EQUAL?
IFT OP ( SE INOP FIRST :L1 :OPER FIRST :L2 COMBINE
        :OPER BF :L1 BF :L2)
EXECUTING INOP 1 * 1
OUTPUT RUN ( SE :P1 :OP :P2 )
OUTPUT: 1
ENDING INOP
EXECUTING COMBINE * [2 3] [2 3]
TEST NOT EQUAL? 1 COUNT :L1
EXECUTING EQUAL? 1 2
OUTPUT :P1 = :P2
OUTPUT: FALSE
ENDING EQUAL?
IFT OP ( SE INOP FIRST :L1 :OPER FIRST :L2 COMBINE
        :OPER BF :L1 BF :L2 )
EXECUTING INOP 2 * 2
OUTPUT RUN ( SE :P1 :OP :P2 )
OUTPUT: 4
ENDING INOP
EXECUTING COMBINE * [3] [3]
TEST NOT EQUAL? 1 COUNT :L1
EXECUTING EQUAL? 1 1
OUTPUT :P1 = :P2
OUTPUT: TRUE
ENDING EQUAL?
IFT OP ( SE INOP FIRST :L1 :OPER FIRST :L2 COMBINE
        :OPER BF :L1 BF :L2 )
IFF OP INOP FIRST :L1 :OPER FIRST :L2
EXECUTING INOP 3 * 3
```

```
OUTPUT RUN ( SE :P1 :OP :P2 )
OUTPUT: 9
ENDING INOP
OUTPUT: 9
ENDING COMBINE
OUTPUT: [4 9]
ENDING COMBINE
OUTPUT: [1 4 9]
ENDING COMBINE
RESULT: [1 4 9]
?NOTRACE
TRACING OFF
?SIGMA COMBINE "* [1 2 3] [1 2 3] ; IE SIGMA [1 4 9]
RESULT: 14
?
```

Before you go any further, try to understand what the above output means. However, if you cannot follow the session, do not worry too much, carry on with the next session, and then return to try again.

Means, standard deviations and covariances

Now we have to come to terms with the calculation of statistical quantities. In statistics, the mean is equal to the sum of all values, divided by the number of values to be summed. The definition of a procedure MEAN has to be, therefore,

```
TO MEAN :L
    OUTPUT DIV SIGMA :L COUNT :L
END

TO DIV :P1 :P2
    OUTPUT :P1/:P2
END
```

and I have thrown in a free prefix procedure to divide two numbers: otherwise, the line in MEAN would be OUTPUT (SIGMA :L)/(COUNT :L) which is not as neat as the prefix line. To find the mean of the numbers [34 56 89], we enter MEAN [34 56 89], to find the answer is 59.6666.

Another useful quantity in statistics is the mean square value. That is, each value is squared, and the sum of the squares is found; the sum of squares is then divided by the number of values; and so you find the mean of the squares. To produce a procedure to find the mean square value

```
TO MSQ :L
```

```
    OUTPUT MEAN COMBINE "* :L :L
END
```

and, for example,

```
?MSQ [1 2 3 4] ; IE THE MEAN OF [1 4 9 16]
RESULT: 7.5
?PR ( 1 + 4 + 9 + 16 )/4
7.5
?
```

There is only one other key quantity necessary for bivariate statistical analysis — the mean crossproduct.

The mean crossproduct is calculated on two lists of values: each pair of values are multiplied together, the crossproducts are summed, and the mean value found. The procedure is thus

```
TO XPROD :L1 :L2
    OUTPUT MEAN COMBINE "* :L1 :L2
END
```

and to test it we try

```
?XPROD [-2 -1 0 1 2] [-2 -1 0 -1 -2]
RESULT: 0
?
```

As I have noted, we now have all the necessary pieces for the calculation of the means, standard deviations, bivariate correlation and regression. First, though, what are the items in the heading to this section? The standard deviation of a list is simply the square root of the result of taking the mean squared value less the square of the means, and

```
TO SDEV :L :M ; :L IS THE INPUT LIST, AND :M IS THE MEAN
    OUTPUT SQRT MINUS MSQ :L MULT :M :M
END

TO MINUS :P1 :P2
    OUTPUT :P1 - :P2
END

TO MULT :P1 :P2
    OUTPUT :P1 * :P2
END
```

where the first parameter of SDEV is the appropriate list, and the second parameter is the mean of the values in that list. The full import of this use of the second parameter will become apparent later, but, for now, to calculate the standard deviation of the values [1 2 3 4 5]

```
?SDEV [1 2 3 4 5] MEAN [1 2 3 4 5]
RESULT: 1.41421
?
```

and that leaves only the covariance.

The covariance routine is very like the SDEV procedure:

```
TO COV :L1 :L2 :M1 :M2
    OUTPUT MINUS XPROD :L1 :L2 MULT :M1 :M2
END
```

because the covariance is the mean crossproduct minus the product of the two ordinary means.

Correlation and regression

The next stage is to calculate the correlation and regression equation for two variables.

The correlation is formed by dividing the covariance by the product of the two standard deviations:

```
TO CORR :COVAR :SD1 :SD2
    OUTPUT DIV :COVAR MULT :SD1 :SD2
END
```

whereas there are two numbers to be output from the procedure to calculate the regression equation.

One number is the regression coefficient (that is, the slope of the line) and the other is the regression constant (that is, where the line cuts the Y axis on the graph) — the formulae are given in my *Pocket Guide to Statistical Programming*, and most books on statistics also have formulae for this topic.

The regression coefficient is produced by taking the covariance between the two variables and dividing it by the square of the standard deviation of the variable to be displayed on the axis X. The regression constant is produced by subtracting the product of the X mean and the regression coefficient from the Y mean. Note how I use a LOCAL variable to make the workings of the procedure more obvious:

```
TO REGRESS :COVAR :SD1 :M1 :M2
```

```
    LOCAL "COEFF
    MAKE "COEFF DIV :COVAR MULT :SD1 :SD1
    OUTPUT SE MINUS :M2 MULT :COEFF :M1 :COEFF
END
```

If the OUTPUT line is parenthesised, the workings are clearer, but the parentheses are unnecessary in actual operation. Even so, the line could be written 'OUTPUT (SE (MINUS :M2 (MULT :COEFF :M1)) (:COEFF))'. A procedure to output the three values (correlation, regression constant, and regression coefficient, in that order) is now easily written:

```
TO CORREG :COVAR :SD1 :SD2 :M1 :M2
    OUTPUT SE CORR :COVAR :SD1 :SD2 REGRESS
        :COVAR :SD1 :M1 :M2
END
```

and it can be seen that, to calculate the three values, all the subsidiary calculations are necessary, even down to the two means. Now to implement a 'statistics system'.

A statistics system

The statistics system will calculate away on two input lists and return a list whose elements will be as follows:

ITEM	ELEMENT
1	CORRELATION
2	REG CONSTANT
3	REG COEFF
4	COVARIANCE
5	STAND DEV X
6	STAND DEV Y
7	MEAN X
8	MEAN Y

The aim is to have a sequence such as

```
?MAKE "BIVALUES BIVARIATE [1 2 3 4 5] [7 5 9 0 10]
?
```

so that the eight possible values are the content of the object named BIVALUES. The BIVARIATE procedure will make extensive use of LOCAL names because this simplifies the construction of the routine.

```
TO BIVARIATE :LX :LY
    LOCAL "MX
    LOCAL "MY
    LOCAL "SX
    LOCAL "SY
    LOCAL "CXY
    LOCAL "R
    MAKE "MX MEAN :LX
    MAKE "MY MEAN :LY
    MAKE "SX SDEV :LX :MX
    MAKE "SY SDEV :LY :MY
    MAKE "CXY COV :LX :LY :MX :MY
    MAKE "R CORREG :CXY :SX :SY :MX :MY
    OUTPUT (SE :R :CXY :SX :SY :MX :MY)
END
```

Here is a typical example of the kind of information which is subjected to an analysis by correlation and regression (taken from *Econometric Methods (2nd ed)* by J. Johnston):

YEAR	CASUALTIES (000) :Y	VEHICLES (0 000) :X
1947	166	352
1948	153	373
1949	177	411
1950	201	441
1951	216	462
1952	208	490
1953	227	529
1954	238	577
1955	268	641
1956	268	692
1957	274	743

It seems clear that, as the number of vehicles increases, the number of casualties increases. The correlation coefficient is a measure of that 'closeness', and the closer the correlation is to 1 (or -1, if the relationship is in the other direction) then the nearer the values lie to a straight line.

We use the above data in this way:

```
?MAKE "X1 [352 373 411 441 462 490 529 577 641 692 743]
```

```
?MAKE "Y1 [166 153 177 201 216 208 227 238 268 268 274]
?MAKE "BIVALUES BIVARIATE :X1 :Y1
?:BIVALUES
RESULT: [0.967572 55.854 0.31196 4806.93
        124.132 40.0221 519.182 217.818]
?( PR [CORRELATION IS] FIRST :BIVALUES )
CORRELATION IS 0.967572
?
```

and if you look at the list 'RESULT: [0.967572 55.854 0.31196 4806.93 124.132 40.0221 519.182 217.818]', you can then work out what is what. To help you slightly, the first item is the correlation. This is a very high correlation (almost 1), which confirms that the two sets of numbers are closely related: as the numbers of vehicles increase, the number of injuries also increase.

The next step is to plot those values, and the regression line, so that we can actually see what the relationship is like.

The scattergram

The two lists of values (that is, casualties and vehicles) can be plotted out on graph paper, and all that has to be decided is the scales to be used. If we use .ASPECT 1, our canvas is 320 units wide by 200 units high, and I suggest we use the portion from −150 to +150, and −90 to +90. This means that there is then room for you to enter coordinates and labels, if you so desire.

To go from the values in the two lists to coordinates on the screen, we need to use a linear transformation. A linear transformation means simply that there is an exact one-to-one straight line relationship between values on the screen axes, and values in the lists. You use a linear transformation to work out distances between points on the ground, given distances on a map.

If the minimum X value in the list is :MINVAL, and the maximum is :MAXVAL, then there is a 'distance' of MINUS :MAXVAL :MINVAL between the two values. This corresponds to the distance between the maximum for the X axis :MAXCOORD and the minimum :MINCOORD, that is, MINUS :MAXCOORD :MINCOORD. Thus, one unit of X value is equal to 'DIV MINUS :MAXCOORD :MINCOORD MINUS :MAXVAL :MINVAL' units on the horizontal axis (the 'abscissa').

This insight gives the hint for a general linear transformation procedure which will also work for the vertical axis (the 'ordinate'). The procedure is

```
TO LINTRAN :INVAL :MINVAL :MAXVAL :MINCOORD
        :MAXCOORD
```

```
    OUTPUT ADD :MINCOORD MULT MINUS :INVAL
        :MINVAL DIV MINUS :MAXCOORD :MINCOORD
        MINUS :MAXVAL :MINVAL
END
```

so, for example, to find the screen value (limits from −90 to +90) corresponding to an X value of 4, where the limits on the X value are 0 to 9:

```
?LINTRAN 4 0 9 ( -90 ) 90
RESULT: -10
?
```

Thus, to draw a scattergram of values from two lists, we define a procedure

```
TO SCATTERGRAM :X :Y :LX :UX :LA :UA :LY :UX :LO :UO
    IF EMPTY? :X STOP
    CROSS LINTRAN FIRST :X :LX :UX :LA :UA LINTRAN
        FIRST :Y :LY :UY :LO :UO
    SCATTERGRAM BUTFIRST :X BUTFIRST :Y :LX :UX
        :LA :UA :LY :UX :LO :UO
END
```

where the sequence of parameters for SCATTERGRAM is

:X	LIST OF X VALUES
:Y	LIST OF Y VALUES
:LX	LOWEST X VALUE
:UX	UPPERMOST X VALUE
:LA	LOWEST ABSCISSA (X AXIS) VALUE
:UA	UPPERMOST ABSCISSA (X AXIS) VALUE
:LY	LOWEST Y VALUE
:UY	UPPERMOST Y VALUE
:LO	LOWEST ORDINATE (Y AXIS) VALUE
:UO	UPPERMOST ORDINATE (Y AXIS) VALUE

To operate we need a procedure CROSS, which is to draw a cross to mark the specified position. I will not provide this procedure, but leave it as a trifling exercise....

Here is how we can use SCATTERGRAM:

```
?.ASPECT 1 DRAW PU HT ; SCREEN CLEARS LEAVING
        TEXT SPACE
?FULLSCREEN SCATTERGRAM :X1 :Y1 300 800 (-150) 150 145
```

```
        280 (-90) 90
?SAVEPICT "D.1
?CS SCATTERGRAM :X1 :Y1 300 800 (-150) 0 145 280 0 90
?SAVEPICT "D.2
?CS SCATTERGRAM :X1 :Y1 300 800 0 150 145 280 (-90) 0
?SAVEPICT "D.3
?
```

and we have saved three pictures. The three are reproduced here as **Figures 10.1, 10.2** and **10.3** and should be compared with the sequence above: note the effect of changing the limits on the admissable abscissa and ordinate values. To call back any of the saved pictures we use 'READPICT "XXX' where XXX is the name of the picture we want.

Use of the linear transformation procedure can give a great deal of flexibility to the drawing of shapes, particularly where the drawing uses SETXY (as does CROSS, a hint).

The regression line

As can be seen, the points plotted are very close to a straight line, and this is reflected in the correlation being near to 1. The line to which the points are close is called the regression line, and we have already calculated the equation of the line.

Corresponding to every possible X value, there is a *predicted* value for Y, and the predicted value is output by this procedure:

```
TO PRED :V :BIV
    OUTPUT ADD ITEM 2 :BIV MULT :V ITEM 3 :BIV
END
```

Remember the second item of the bivariate list is the regression constant, and the third item is the regression coefficient. For example

```
?PRED 743 :BIVALUES
RESULT: 287.64
?
```

and the predicted number of casualties is 287.64 when the number of vehicles is 743 (as it was in 1957). Thus, in 1957, the number of road casualties was less than would be expected, given the other figures. The reason for this is up for discussion....

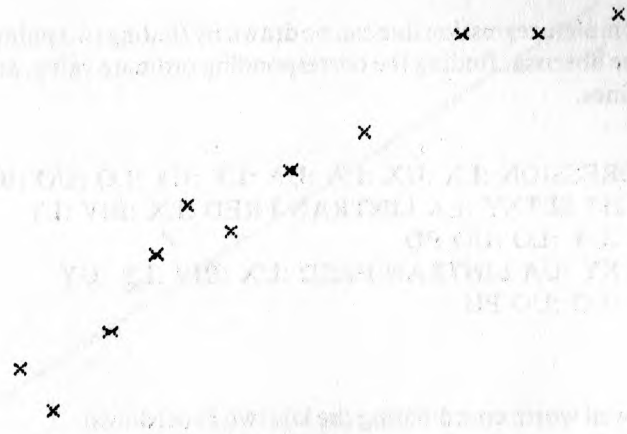


Figure 10.1

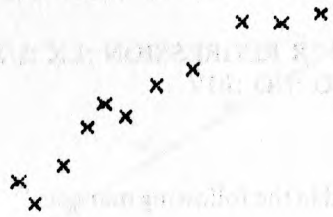


Figure 10.2

Figure 10.3

The complete regression line can be drawn by finding two points at either end of the abscissa, finding the corresponding ordinate value, and joining the two lines.

```
TO REGRESSION :LX :UX :LA :UA :LY :UY :LO :UO :BIV
    PU HT SETXY :LA LINTRAN PRED :LX :BIV :LY
        :UY :LO :UO PD
    SETXY :UA LINTRAN PRED :UX :BIV :LY :UY
        :LO :UO PU
END
```

It seems well worth coordinating the last two procedures:

```
TO BIPLOT :X :Y :LX :UX :LA :UA :LY :UY :LO :UO :BIV
    .ASPECT 1 HT PU CS
    SCATTERGRAM :X :Y :LX :UX :LA :UA :LY :UY
        :LO :UO
    IF EQUAL? RC "R REGRESSION :LX :UX :LA :UA
        :LY :UY :LO :UO :BIV
END
```

which can then be used in the following manner:

```
?BIPLOT :X1 :Y1 300 800 (-150) 150 145 280 (-90)
        90 :BIVALUES
?SAVEPICT "D.4 ; NOTE FUNNY BOTTOM RIGHT
?BIPLOT :X1 :Y1 352 743 (-140) 140 153 274 (-80)
        80 :BIVALUES
?SAVEPICT "D.5 ; VALUES RESTRICTED
?BIPLOT :X1 :Y1 350 750 (-130) 130 150 280 (-70) 70
        :BIVALUES
?SAVEPICT "D.6
?BIPLOT :X1 :Y1 350 750 (-70) 70 150 280 (-40) 40
        :BIVALUES
?SAVEPICT "D.7 ; A TINY PICTURE
?
```

If **Figures 10.4, 10.5, 10.6** and **10.7** are studied (apart from the case where the line went out of bounds), it is clear that, though there is a close relationship between casualties and vehicles, the relationship is not linear. The points seem to follow a curve: the relationship is 'curvilinear'.

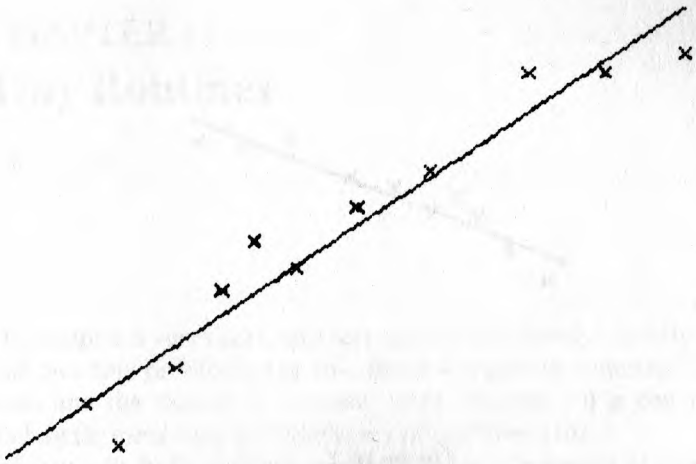


Figure 10.4

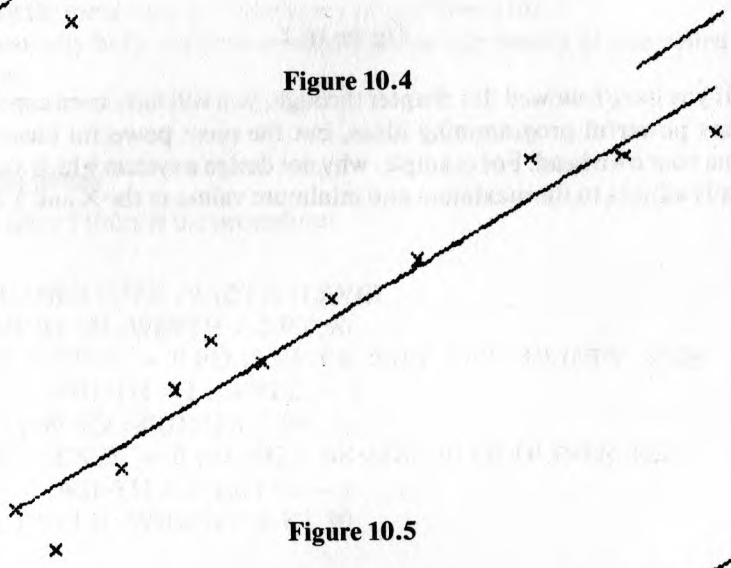


Figure 10.5

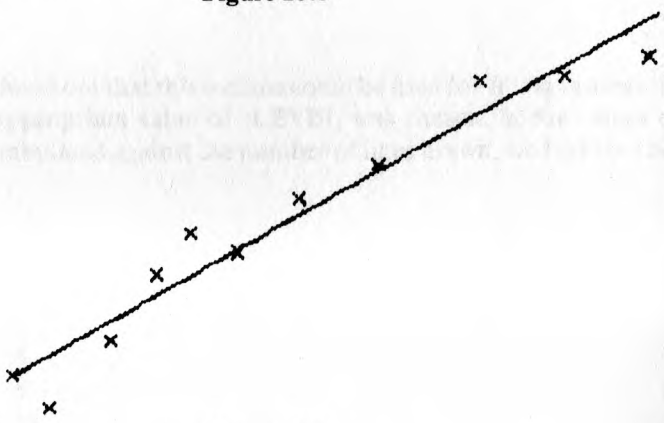


Figure 10.6

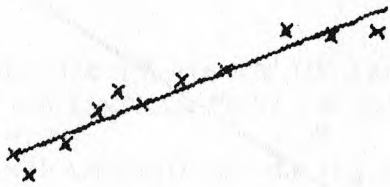


Figure 10.7

If you have followed this chapter through, you will have been exposed to many powerful programming ideas, but the most powerful ideas come from your own head. For example, why not design a system which automatically adjusts to the maximum and minimum values in the X and Y lists?

CHAPTER 11
Tiny Routines

This chapter is very short, and very simple. Effectively it is only concerned with two tiny problems: the first (from Chapter 9) concerns filling in an area, and the second (in a sense, from Chapter 10) is concerned with finding the maximum and minimum values from a list.

Essentially both applications show the simple beauty of controlled recursion.

Filling areas

In Chapter 9 there is the procedure:

```
TO DLINES :SIZE :WIDTH :LEVEL
    LT 90 FD :WIDTH / 2 RT 90
    IF :LEVEL = 0 FD :SIZE BK :SIZE ELSE DLINES :SIZE
        :WIDTH / 2 :LEVEL - 1
    RT 90 FD :WIDTH LT 90
    IF :LEVEL = 0 FD :SIZE BK:SIZE ELSE DLINES :SIZE
        :WIDTH / 2 :LEVEL -1
    LT 90 FD :WIDTH / 2 RT 90
END
```

and it was pointed out that this routine could be used for filling in areas, as long as the appropriate value of :LEVEL was chosen. If the values of :LEVEL are tabulated against the number of lines drawn, we find the following:

:LEVEL	Lines
0	2
1	4
2	8
3	16
4	32

and so on. The progression is in powers of two (that is, at each level, the number of lines doubles). A simple procedure to produce the correct level value is

```
TO LEVEL :N
    IF ANYOF ( :N = 2 ) ( :N < 2 ) OUTPUT 0 ELSE
        OUTPUT 1 + LEVEL :N / 2
END
```

and to draw and fill in a square box of side 100, we have to enter

```
?.ASPECT 1 DLINES 100 50 LEVEL 100 ; TAKES; AGES TO
        FINISH
?
```

Find out what value is output by LEVEL for various inputs, just to check that it works. Then find out why it works, or does not work.

Incidentally, it is probably simpler to alter DLINES by

```
TO DLINES :SIZE :WIDTH
    LT 90 FD :WIDTH / 2 RT 90
    IF :WIDTH < 1 FD :SIZE BK :SIZE ELSE DLINES :SIZE
        :WIDTH / 2
    RT 90 FD :WIDTH LT 90
    IF :WIDTH < 1 FD :SIZE BK :SIZE ELSE DLINES :SIZE
        :WIDTH / 2
    LT 90 FD :WIDTH / 2 RT 90
END
```

There is a slight error in this procedure, what is it? *(Hint:* powers of two.)

Maximum and minimum

How can we find the largest element of a list? (Or the smallest?) We must go through the list, and compare at each stage the largest so far, against the next item. The largest so far is the larger of the two values. If we have only one value, then that is the largest. We have defined a procedure in words, and in LOGO:

```
TO MAX :L
    LOCAL "TEMP
    TEST 1 = COUNT :L
    IFTRUE OUTPUT FIRST :L
    IFFALSE MAKE "TEMP MAX BF :L IF FIRST :L > :TEMP
        OUTPUT FIRST :L ELSE OUTPUT :TEMP
END
```

which we can try out

```
?MAX [1 9 2 8 3 5] ; A LIST OF DIGITS
RESULT: 9
?MAX 192835 ; A NUMBER WITH THE SAME DIGITS
RESULT: 9
?
```

LOGO treats a list like a list, LOGO treats a number as if it were a word, and LOGO treats a word as if it were a list of characters. In some cases LOGO gets carried away. Watch what happens:

```
?1234567890987654321
RESULT: 1.23456E18
?MAX 1234567890987654321
> DOESN'T LIKE E AS INPUT, IN LINE
IFFALSE MAKE "TEMP MAX BF :L IF FIRST :L > :TEMP
        OUTPUT FIRST :L ELSE OUTPUT :TEMP
AT LEVEL 8 OF MAX.
?
```

and notice that E is number eight in the sequence of characters known as 1.23456E18 (do not forget to count the stop/period).

I do not like MAX as well as I might, because it uses LOCAL and is a bit messy. Trying to think of a neater solution, I decided that the key idea was to output the larger of the two numbers, so I defined

```
TO 2MAX :P1 :P2
    IF :P1 > :P2 OUTPUT :P1 ELSE :P2
END
```

which takes two inputs and outputs the larger. The new improved version of MAX is thus

```
TO MAXX :L
    TEST 1 = COUNT :L
    IFTRUE OUTPUT FIRST :L
    IFFALSE OUTPUT 2MAX FIRST :L MAXX BUTFIRST :L
END
```

which may not impress you, but it certainly impresses me. I leave you to improve further, and to produce routines for the minimum value.

I am happy.

CHAPTER 12

Differential Drawing

This chapter contains, mathematically speaking, probably the most advanced topic in this book. However, in the context of turtle graphics, it is one of the simplest parts to program—we examine differential calculus, and the integration of differential equations.

The differential dy/dx

In ordinary mathematics, the 'differential' (for example, dy/dx) is defined as the slope of the tangent to a curve, at any specific point.

Thus, if the differential equation is

```
dy/dx = x^2/4 - 4x - 5
```

then, at any particular value of x, the slope of the tangent to the curve is given by that equation. If the value of x is 3, then the slop is 3*3/4 - 4*3 - 5, that is - 14.75.

The study of differential equations is known as differential calculus (or, in short, 'calculus'). Calculus is a very important topic in more advanced mathematics, and an understanding of calculus is useful, because we can now use calculus, which is very complicated, very simply.

It is simpler (in ordinary mathematics) to work out the differential equation from the original equation from which the differential equation is derived. Sometimes the original equation is easy to find, and the original equation is, in this case:

```
y = x^3/12 - 2x^2 - 5x + K
```

where K is a 'constant of integration'. Working backwards, from the differential equation to the original equation, is known as integral calculus or 'integration'.

Integration is far more complex than differentiation, except in LOGO. To see why this is so, examine **Figure 12.1**. The curved line has a tangent at the specific point shown (that labelled P). The tangent is extended to form the hypotenuse of the triangle shown, and the value of the gradient (that is, dy/dx) at P is equal to the value of Y/X. Therefore, if a turtle is drawing

a curve, at every point the turtle faces in the direction of the tangent to the curve. (Think about it.)

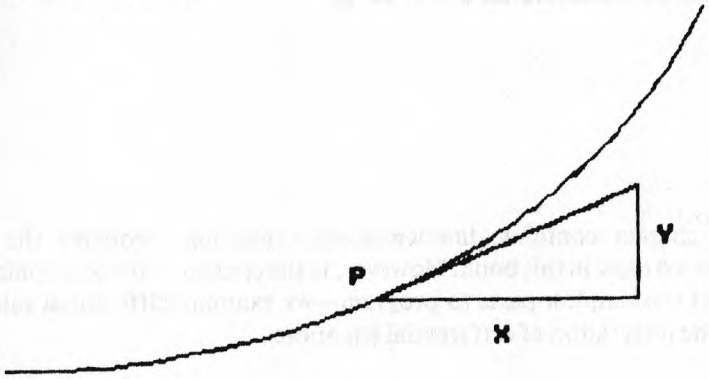


Figure 12.1

To draw the curve which corresponds to the original equation, then all we have to do is to let the turtle follow the directions given in the differential equation. Thus the integration of curves is second nature to the turtle, in fact the turtle is differential geometry personified.

Drawing the original equation

All we have to produce is a procedure which lets the turtle follow the equation of the tangent. The turtle cannot be continually modifying its direction (to point in the direction of the tangent), so we will have to allow it to move a short distance before we modify the direction.

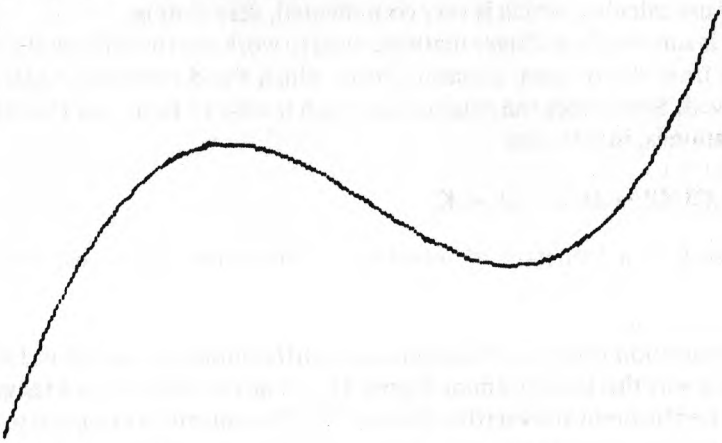


Figure 12.2

As we found with drawing circles, short straight lines produce effects which look remarkably like smooth curves. Look at **Figure 12.2**, which is the original curve produced for the differential

dy/dx = x^2/4 – 4x – 5

given above. The limits on the value of x are – 15 to 35, and the limits on y are 0 to 1000 (we always assume the constant of integration is zero). These limits were found by experiment.

I used prefix mathematical operators for the procedures — to save any problems concerning precedence, and all but one of these are familiar from Chapter 10:

```
TO GT? :P1 :P2 ; THE NEW ONE
    OUTPUT :P1 > :P2
END

TO DIV :P1 :P2
    OUTPUT :P1 / :P2
END

TO MINUS :P1 :P2
    OUTPUT :P1 - :P2
END

TO MULT :P1 :P2
    OUTPUT :P1 * :P2
END

TO ADD :P1 :P2
    OUTPUT :P1 + :P2
END
```

The immediate decision concerns how to set up the differential equation so that the turtle knows what to obey. The example differential equation can be LOGOfied by

```
MAKE "DIFF.EQUATION [:X * :X / 4 - 4 * :X - 5]
```

but it is always possible to pass the equation as a parameter to a procedure. However the operation is performed, the equation is activated by use of RUN.

The procedure we call to perform the integration is

```
TO DIFPLOT :EQ :XL :XU :YS :YU
```

```
    .ASPECT 1 ; I JUST LIKE IT THAT WAY
    DRAW FULLSCREEN HT PU SETXY ( - 150 ) ADD
        MULT 180 DIV :YS :YU ( - 90 ) PD
    DRAWLINE :EQ :XL :XL :XU :YU
END
```

and the interpretation of the parameters for DIFPLOT is as follows

INPUT	FUNCTION
:EQ	THE EQUATION LIST
:XL	THE LOWER X VALUE
:XU	THE UPPER X VALUE
:YS	THE START Y VALUE
:YU	THE UPPER Y VALUE

where the lower Y is always taken to be zero. :YS sets how high up the screen the curve starts. The screen coordinates extend from −150 to +150, and −90 to +90, and so the curve always starts (SETXY) at −150 and then an amount which depends upon the relative sizes of YS and YU. It is worth your effort to disentangle the meaning of 'ADD MULT 180 DIV :YS :YU (− 90) PD.

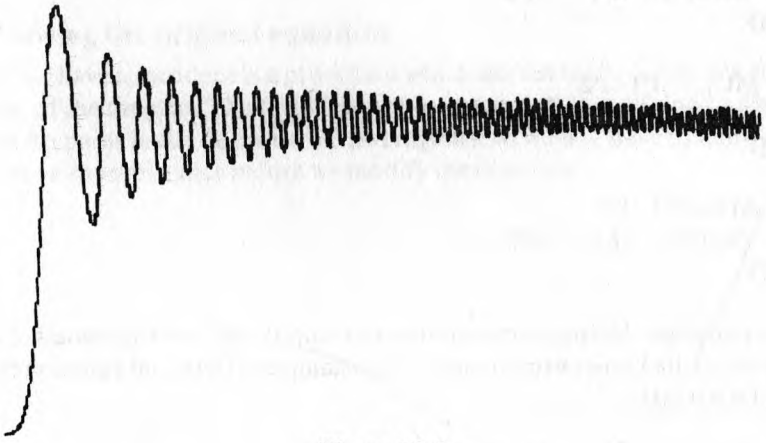


Figure 12.3

In the case of Figure 12.2, the curve was drawn by use of the call 'DIFPLOT :DIF.EQUATION (− 15) 35 0 1000'.

The procedure DRAWLINE is called as the last line of the procedure:

```
TO DRAWLINE :EQ :X :XL :XU :YU
    IF ( ANYOF GT? YCOR 90 GT? - 90 YCOR GT? :X :XU )
        STOP
```

```
    SETH MINUS 90 ATAN MULT DIV 180 :YU RUN :EQ
        DIV 300 MINUS :XU :XL
    FD 5
    DRAWLINE :EQ ( ADD :XL MULT MINUS :XU :XL DIV
        MINUS XCOR ( - 150 ) 300 ) :XL :XU :YU
END
```

The first line checks to see if the turtle is out of bounds, the second line sets the turtle heading in the correct direction (it uses ATAN modified for the scaling of X and Y axes), the third line moves the turtle forward five units, and the fourth line calls DRAWLINE with a new position for the X axis (derived from the value of XCOR).

I will only give one more example of the system: the equation was [SIN :X * :X], and to get sufficient fineness of detail I modified the FD 5 in DRAWLINE to FD .5. The actual call was

```
?DIFPLOT [SIN :X * :X] 0 150 0 7
```

and, even though I used such a short move by the turtle, the resolution of the graphics screen still coarsens the effect. (And the time taken to draw the figure...)

Still, the procedures allow experimentation with a piece of powerful mathematics, so all is not wasted.

CHAPTER 13

Spritely Icons

The Icon System (TIS) is currently under development (but may be available by the time you read this book), and thus is incomplete, with many snags which need ironing out. TIS takes some of its ideas from Marlene Kliman's excellent Programmable Icons (PI) system.

The PI system is designed to help less able children to use a computer, especially those children with restricted physical capabilities, and so is TIS. The intrinsic interest of TIS is such, however, that it has been enjoyed by more intelligent children. Marlene's work is of special interest because she designed the sprite routines for C64 LOGO.

The aim of TIS

Interactive drawing on the graphics screen by use of the turtle system, even using a joystick, can be complex and (in the case of the routines in Chapter 5) requires speedy reaction. Even though it is simpler than for other languages, to program in LOGO still requires some slight effort — far too great an effort for some children.

The turtle is an arresting beast, so is it possible to control that turtle with no need for instant responses, and with no need to be proficient with a keyboard? For young children, and the physically and/or mentally handicapped , the ability to press several keys with ease may never come.

There have been many special input devices designed to assist the less proficient in controlling the computer. Two devices which come to mind are the 'concept keyboard' and the 'button box'. The cheapest device, and one which can be used by many less proficient individuals, is the joystick; especially if the movements of the joystick are kept to 'forward' and to 'back'.

With the advent of the computer language Smalltalk (and its progeny the Apple Lisa and Macintosh), the ways in which the user communicates with the computer have been extended. The Smalltalk systems use the very simple idea of putting pictures on the screen, where each picture indicates an action. In a Smalltalk environment, the pictures are known as icons, where each icon gives a visual clue to its intended action. The user directs a cursor to point to an icon, thus indicating the desired action.

With true Smalltalk systems, the pointing is usually accomplished by a 'mouse', a device (with a button or two) which is moved across a surface. The movements of the mouse are reproduced by the movements of the cursor/pointer on the screen, and actions are activated by pressing a button on the mouse. My mouse will be a joystick, but the pointer will only move up and down along one vertical line, and, though the joystick does have a button, you may find in some cases that the spacebar is more positive: buttons can be temperamental, but spacebars are big and can be thumped.

My aim is, therefore, to produce a visually attractive system which is very easy to use, yet has sufficient intrinsic interest to be worthwhile. The aim is not to provide a finished system, but to provide some ideas which can be developed by those interested. A further aim is to introduce some new programming ideas, which will have relevance for other applications.

TIS — the design of the screen

The screen is to be split into two main functional areas to reinforce the distinction between indications to actions, and the actions themselves.

The functional area to the left will be composed of a vertical line of icons (indications to actions). To the right of the line of icons is a pointer, which moves up and down, pointing to each icon (ie indication) in turn. The pointer will be coloured light grey/gray (so that it is easily seen) and when the button on the joystick is pressed (or, alternatively, the spacebar depressed) the indicated action is activated. Whilst the indicated action is in operation, the pointer changes colour to confirm that the action is taking place.

The functional area in the centre of the screen is the action area, and is a rectangle composed of 64 smaller rectangles (eight by eight). The small rectangles are filled in during the progress of the turtle, as indicated by the content of the icons. The right of the screen is left blank for future enhancements.

In this simple system there are four icons: move forward one rectangle, colouring in the rectangle you have left; turn to the right, without moving; change colour of turtle and rectangles when the turtle moves; and reset system. Other actions might be raise and lower pen, stop, and erase — but that is up to you.

The icons, and the pointer arrow, are all sprites. The sprite shapes have had to be specifically designed for this application, so you may care to improve on the designs. **Figures 13.1, 13.2, 13.3, 13.4,** and **13.5** are screen dumps of the designs of the sprites for TIS. The sprite shapes were copied from where they were stored in memory, and then printed out as characters on the high resolution screen. I designed the procedures to perform this

operation, and these use two procedures from the utility disk, saved under the name STAMPER (*LLL*, Appendix).

Here are my routines which I give without any explanation, apart from remarking that the sprite shapes (1–7) are stored from locations 3136 to 3583. The information on each sprite is stored in chunks of 64 locations where the first 63 locations give shape information, and the 64th location gives status information.

```
TO STAMPSPRITE :S :X :Y
    DRAW TELL 0 EACH "01234567 [HT PU]
    SETXY :X :Y
    POSN ( :S - 1 ) * 64 + 3136 21
END

TO XL :L
    OUTPUT FMOD .EXAMINE :L
END

TO EXL :L
    OUTPUT ( WORD XL :L XL :L + 1 XL :L + 2 )
END

TO FMOD :V
    LOCAL "POW2
    MAKE "POW2 [1 2 4 8 16 32 64 128]
    OUTPUT FWORD 8 ITEM 8 :POW2
END

TO FWORD :N :B
    IF 1 = :N IF 1 = BITAND 1 :V OP "* ELSE OP ".
    IF :B = BITAND :B :V OP WORD "* FWORD :N - 1
        ITEM :N - 1 :POW2 ELSE OP WORD ". FWORD
        :N - 1 ITEM :N - 1 :POW2
END
```

The sprite shapes were created by use of EDSH, which is one of the sprite routines loaded when READ "SPRED is used (*LLL*, Sprites). The sprite editor is a very useful system to master, why not try to rewrite it?

Running TIS

To get the TIS system up and running, I use READ "TIS, and after a wait I see a list of procedures being loaded, some of which are the sprites routines from the utility disk.

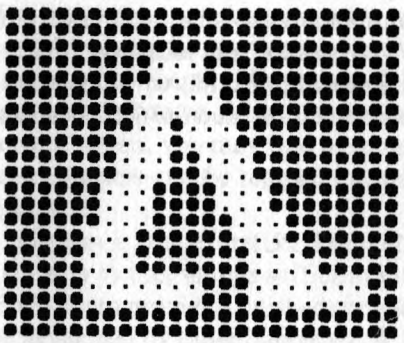


Figure 13.1: Sprite1

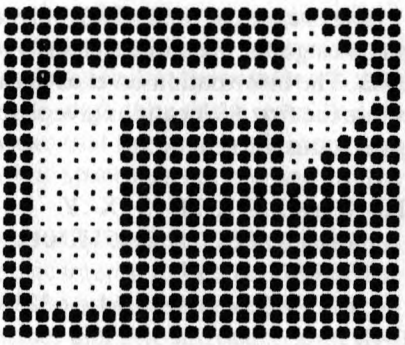


Figure 13.2: Sprite 2

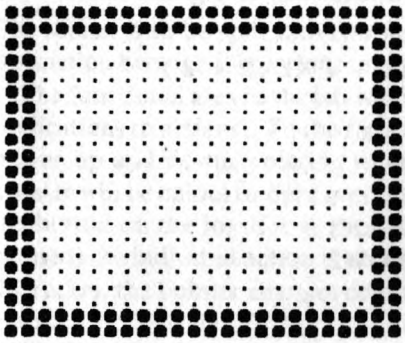


Figure 13.3: Sprite 3

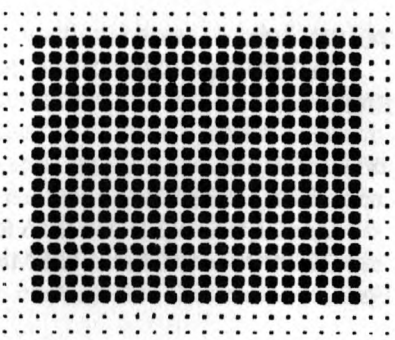


Figure 13.4: Sprite 4

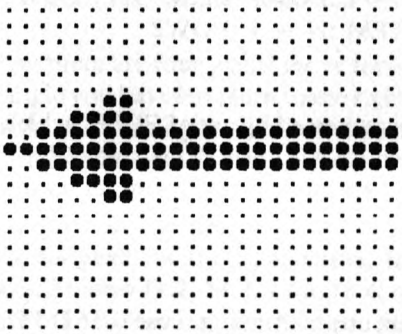


Figure 13.5: Sprite 5

When that is completed, the screen clears and the background becomes dark, and then shapes move to the lefthand side of the screen. The top shape is a pair of walking legs (truly), the next one down is a right turn arrow, below that there is a cyan square, and at the bottom there is a white border around a dark area: pointing at the top shape is an arrow. These are all sprites, the designs of which you can see. This is the 'indications' functional area.

In the centre, a lattice of lines is drawn to produce the large rectangle with its 64 smaller rectangles. The turtle positions itself in the bottom left rectangle of the lattice.

The system is ready for use: moving the joystick handle back or forward moves the arrow up or down, and the arrow wraps round at both top and bottom. To press the joystick button whilst the arrow points to the legs (1), makes the turtle move up a block, colouring in the rectangle it has left. Moving the joystick back until it stops at the right turn arrow (2) and then pressing the button, turns the turtle.

To change colour the cyan square (3) is selected: the button is pressed, and the arrow changes colour. Moving the joystick back or forward produces a change in the colour of the square, a change in colour which is reflected in the turtle colour. Pressing the button again deselects that action, and the arrow is ready to move.

To select the fourth sprite is to restart.

To get back to LOGO requires a CTRL-G, and an f1: if I now enter:

```
?:STARTUP
RESULT: [.OPTION "STAMPCHAR 0 1 READSHAPES
        "TIS TISET]
?
```

and, when I saved the TIS file, I had entered

```
?MAKE "STARTUP [.OPTION "STAMPCHAR 0 1 READSHAPES
        "TIS TISET]
?SAVE "TIS SAVESHAPES "TIS
?
```

On loading a file of procedures, LOGO looks to see if the standard name STARTUP has a content. If the content is a list, then LOGO RUNs that list. Thus when TIS is loaded, the .OPTION procedure is used with "STAMPCHAR (*LLL*, Appendix), the sprite shapes are loaded into memory, and then the procedure TISET is called.

The use of STARTUP is the way in which automatic systems are set up.

Starting TIS

I will now go through the TIS initial procedure, commenting as necessary — by now you should be able to work out a good deal for yourself.

```
TO TISET
    LOCAL "POSN ; THE ARROW POSITION
    MAKE "POSN 3 ; START AT POSITION 3 - THE TOP
    .ASPECT 1 ; SQUARER COORDINATES, EASIER TO
        PLACE ITEMS
    BG 0 ; PAINT IT BLACK
     HT FULLSCREEN ; ALL THE TELLS PLACE SHAPES
    TELL 3 PC 3 ST BIG SETXY - 150  90
    TELL 1 PC 3 ST BIG SETXY - 150  45
    TELL 2 PC 13 ST BIG SETXY - 150  0
    TELL 4 PC 1 ST BIG SETXY - 150 ( - 45 )
    TELL 5 PC 15 ST SMALL SETXY - 100 8O
    TELL 0 PC 13 ST PU SETH 0
    LATTICE ; DRAWS RECTANGLES
    MOVEARROW ; DOES WHAT IT SAYS
END

TO BIG
    BIGX BIGY
END

TO SMALL
    SMALLX SMALLY
END

TO LATTICE
    CS PU SETXY - 56 ( - 99)
    P.LINES 192  16  8
    LT 90
    P.LINES 128  24  8
    PU SETXY - 48 ( - 87 )
    RT 90
    GETCOL ST
END
```

Thus far we have seen that the TISET procedure is fairly easy to follow: the ease is somewhat misleading, because it took me a long time to find numbers which fitted pleasantly together. The construction of LATTICE uses a procedure to draw sets of parallel lines:

```
TO P.LINES :SIDE :DIST :NUM
```

```
    LINE :SIDE ; DRAWS A DOUBLE LINE
    REPEAT :NUM [RT 90 PU FD :DIST PD LT 90 LINE
        :SIDE]
END
```

and the procedure LINE is left to your programming.

The whole system is now ready to go: the action is conducted via the kind offices of MOVEARROW:

```
TO MOVEARROW
    IF RC? CLEARINPUT TELL 5 PC 9 TELL 0 ACTION
        :POSN TELL 5 PC 15 TELL 0 ; NOTE THAT THIS
        LINE USES THE KEYBOARD NOT THE
        JOYBUTTON
    IF 0 = JOYSTICK 1 THEN MAKE "POSN REMAINDER
        :POSN + 1 4 ; UP ARROW
    IF 4 = JOYSTICK 1 THEN MAKE "POSN REMAINDER
        :POSN + 3 4 ; DOWN ARROW
    TELL 5 SETY :POSN * 5 - 55 TELL 0
    REPEAT 150 [ ] ; TIME LAG
END

TO ACTION :NUM
    IF :NUM = 3 FILL ; FILL IN A RECTANGLE
    IF :NUM = 2 RT 90 ; TURN RIGHT
    IF :NUM = 1 CHANGECOL ; CHANGE FILL COLOUR
    IF :NUM = 0 TISET ; RESET SYSTEM
END
```

and procedure MOVEARROW uses ACTION to act. ACTION calls one of four procedures, of which the most interesting is FILL.

Filling rectangles

First the FILL procedure has to save the present position of the turtle, ready for modification, and then to call one of four routines depending upon the direction of movement. Here it is:

```
TO FILL
    LOCAL "X
    LOCAL "Y
    MAKE "X XCOR
    MAKE "Y YCOR
    HT
```

```
    RUN ( SE WORD HEADING ".FILL ) ; HEADING WILL
        BE 0, 90, 180, OR 270
    ST
END

TO 0. FILL
    PU UP.FILL PU
    SETXY :X :Y FD 24
END

TO 180.FILL
    PU RT 180 UP.FILL PU
    RT 180 SETXY :X :Y FD 24
END

TO 90.FILL
    PU LT 90 UP.FILL PU
    RT 90 SETXY :X :Y FD 16
END

TO 270.FILL
    PU RT 90 UP.FILL PU
    LT 90 SEXTY :X :Y 16
END
```

When it comes down to details, the key routine now appears to be UP.FILL, which is a routine to fill in a rectangle from the bottom to the top in three lines of two characters at a time (the rectangle is equal to two characters wide by three characters high, 16 by 24).

Characters are used on the screen because it makes the filling of areas with different colours that much easier (try using lines to fill to see what I mean). The actual LOGO command used is STAMPCHAR, which prints characters at the turtle's position on the screen. When I used the .OPTIONS "STAMPCHAR 0 1 (*LLL*, Appendix) in the STARTUP list, this allowed me to superimpose characters over each other, by overwriting. Normally a new character erases the old character.

It is at this point that things become more complex. First, UP.FILL:

```
TO UP.FILL ; TOTAL RECTANGLE IS 16 BY 24
    STAMPLINE :X - 8 :Y - 12 ; BOTTOM LINE
    STAMPLINE :X - 8 :Y - 4 ; MIDDLE LINE
    STAMPLINE :X - 8 :Y + 4 ; TOP LINE
END
```

where, as usual, difficulties have been relegated to another procedure, where the procedure looks like

```
TO STAMPLINE :X :Y
    LOCAL "XT
    LOCAL "YT
    MAKE "XT XCOR ; SAVE X COORDINATE
    MAKE "YT YCOR ; SAVE Y COORDINATE
    PU SETXY :X :Y ; MOVE TO SPECIFIED
        COORDINATES
    STAMP ; PRINT OVERPRINTED CHARACTERS
    SETXY :X + 8 ; MOVE SIDEWAYS
    STAMP ; MORE CHARACTERS
    SETXY :XT :YT ; RETURN TO ORIGINAL
        COORDINATES
END

TO STAMP ; HAPPEN TO FIND O AND Z OVERWRITE
        WELL
    STAMPCHAR "O
    STAMPCHAR "Z
END
```

The best way to sort out how these routines operate is by using them on screen, and experimenting. If you try SINGLECOLOR and then DOUBLECOLOR you will see more unusual effects.

Changing colours

These are the final procedures:

```
TO CHANGECOL
    TELL 2
    IF 0 = JOYSTICK 1 PC REMAINDER ( ITEM 4
        DRAWSTATE ) + 1 16
    IF 4 = JOYSTICK 1 PC REMAINDER ( ITEM 4
        DRAWSTATE ) + 1 16
    GETCOL
    REPEAT 150 [ ]
    IF RC? CLEARINPUT TELL 5 STOP
    CHANGECOL
END
```

```
TO GETCOL
    LOCAL "T
    TELL 2
    MAKE "T ITEM 4 DRAWSTATE
    TELL 0 PC :T
END
```

and the result of ITEM 4 DRAWSTATE is the current pen-colour (see *LLL*, Appendix).

The CHANGECOL procedure works by cycling through the colours backwards and forwards in numerical sequence (see *LLL*, Appendix), and GETCOL finds the current colour (of sprite 2) and then sets the turtle to that colour. (Note that in the Appendix of *LLL*, they get very confused about their colours.)

With enhancements, TIS can be a very powerful system, and it is already a major programming exercise.

Index

Other titles from Sunshine

SPECTRUM BOOKS

Artificial Intelligence on the Spectrum Computer
Keith & Steven Brain — ISBN 0 946408 37 8 — **£6.95**

Spectrum Adventures
Tony Bridge & Roy Carnell — ISBN 0 946408 07 6 — **£5.95**

Machine Code Sprites and Graphics for the ZX Spectrum
John Durst — ISBN 0 946408 51 3 — **£6.95**

ZX Spectrum Astronomy
Maurice Gavin — ISBN 0 946408 24 6 — **£6.95**

Spectrum Machine Code Applications
David Laine — ISBN 0 946408 17 3 — **£6.95**

The Working Spectrum
David Lawrence — ISBN 0 946408 00 9 — **£5.95**

Inside Your Spectrum
Jeff Naylor & Diane Rogers — ISBN 0 946408 35 1 — **£6.95**

Master your ZX Microdrive
Andrew Pennell — ISBN 0 946408 19 X — **£6.95**

COMMODORE 64 BOOKS

Graphic Art for the Commodore 64
Boris Allan — ISBN 0 946408 15 7 — **£5.95**

DIY Robotics and Sensors on the Commodore Computer
John Billingsley — ISBN 0 946408 30 0 — **£6.95**

Artificial Intelligence on the Commodore 64
Keith & Steven Brain — ISBN 0 946408 29 7 — **£6.95**

Simulation Techniques on the Commodore 64
John Cochrane — ISBN 0 946408 58 0 — **£6.95**

Machine Code Graphics and Sound for the Commodore 64
Mark England & David Lawrence — ISBN 0 946408 28 9 — **£6.95**

Commodore 64 Adventures
Mike Grace — ISBN 0 946408 11 4 — **£5.95**

Business Applications for the Commodore 64
James Hall — ISBN 0 946408 12 2 — **£5.95**

Mathematics on the Commodore 64
Czes Kosniowski — ISBN 0 946408 14 9 — **£5.95**

Advanced Programming Techniques on the Commodore 64
David Lawrence — ISBN 0 946408 23 8 — **£5.95**

Commodore 64 Disk Companion
David Lawrence & Mark England ISBN 0 946408 49 1 **£7.95**

The Working Commodore 64
David Lawrence ISBN 0 946408 02 5 **£5.95**

Commodore 64 Machine Code Master
David Lawrence & Mark England ISBN 0 946408 05 X **£6.95**

Machine Code Games Routines for the Commodore 64
Paul Roper ISBN 0 946408 47 5 **£6.95**

Programming for Education on the Commodore 64
John Scriven & Patrick Hall ISBN 0 946408 27 0 **£5.95**

Writing Strategy Games on your Commodore 64
John White ISBN 0 946408 54 8 **£6.95**

COMMODORE 16/PLUS 4 BOOKS

The Working Commodore C16
David Lawrence ISBN 0 946408 62 9 **£6.95**

ELECTRON BOOKS

Graphic Art for the Electron Computer
Boris Allan ISBN 0 946408 20 3 **£5.95**

Programming for Education on the Electron Computer
John Scriven & Patrick Hall ISBN 0 946408 21 1 **£5.95**

The Working Electron
John Scriven ISBN 0 946408 52 1 **£5.95**

BBC COMPUTER BOOKS

Functional Forth for the BBC Computer
Boris Allan ISBN 0 946408 04 1 **£5.95**

Graphic Art for the BBC Computer
Boris Allan ISBN 0 946408 08 4 **£5.95**

DIY Robotics and Sensors for the BBC Computer
John Billingsley ISBN 0 946408 13 0 **£6.95**

Artificial Intelligence on the BBC and Electron
Keith & Steven Brain ISBN 0 946408 36 X **£6.95**

Essential Maths on the BBC and Electron Computer
Czes Kosniowski ISBN 0 946408 34 3 **£5.95**

Programming for Education on the BBC Computer
John Scriven & Patrick Hall ISBN 0 946408 10 6 **£5.95**

Making Music on the BBC Computer
Ian Waugh — ISBN 0 946408 26 2 — **£5.95**

DRAGON BOOKS

Advanced Sound & Graphics for the Dragon
Keith & Steven Brain — ISBN 0 946408 06 8 — **£5.95**

Artificial Intelligence on the Dragon Computer
Keith & Steven Brain — ISBN 0 946408 33 5 — **£6.95**

Dragon 32 Games Master
Keith & Steven Brain — ISBN 0 946408 03 3 — **£5.95**

The Working Dragon
David Lawrence — ISBN 0 946408 01 7 — **£5.95**

The Dragon Trainer
Brian Lloyd — ISBN 0 946408 09 2 — **£5.95**

ATARI BOOKS

Atari Adventures
Tony Bridge — ISBN 0 946408 18 1 — **£5.95**

Writing Strategy Games on your Atari Computer
John White — ISBN 0 946408 22 X — **£5.95**

SINCLAIR QL BOOKS

Artificial Intelligence on the Sinclair QL
Keith & Steven Brain — ISBN 0 946408 41 6 — **£6.95**

Introduction to Simulation Techniques on the Sinclair QL
John Cochrane — ISBN 0 946408 45 9 — **£6.95**

Developing Applications for the Sinclair QL
Mike Grace — ISBN 0 946408 63 7 — **£6.95**

Mathematics on the Sinclair QL
Czes Kosniowski — ISBN 0 946408 43 2 — **£6.95**

The Working Sinclair QL
David Lawrence — ISBN 0 946408 46 7 — **£6.95**

Quill, Easel, Archive & Abacus on the Sinclair QL
Alison McCallum-Varey — ISBN 0 946408 55 6 — **£6.95**

Assembly Language Programming on the Sinclair QL
Andrew Pennell — ISBN 0 946408 42 4 — **£7.95**

GENERAL BOOKS

Home Applications on your Micro
Mike Grace — ISBN 0 946408 50 5 — **£6.95**

Sunshine also publishes

POPULAR COMPUTING WEEKLY

The first weekly magazine for home computer users. Each copy contains Top 10 charts of the best-selling software and books and up-to-the-minute details of the latest games. Other features in the magazine include regular hardware and software reviews, programming hints, computer swap, adventure corner and pages of listings for the Spectrum, Dragon, BBC, VIC 20 and 64, ZX 81 and other popular micros. Only 40p a week, a year's subscription costs £19.95 (£9.98 for six months) in the UK and £37.40 (£18.70 for six months) overseas.

DRAGON USER

The monthly magazine for all users of Dragon microcomputers. Each issue contains reviews of software and peripherals, programming advice for beginners and advanced users, program listings, a technical advisory service and all the latest news related to the Dragon. A year's subscription (12 issues) costs £10 in the UK and £16 overseas.

MICRO ADVENTURER

The monthly magazine for everyone interested in Adventure games, war gaming and simulation/role-playing games. Includes reviews of all the latest software, lists of all the software available and programming advice. A year's subscription (12 issues) costs £10 in the UK and £16 overseas.

COMMODORE HORIZONS

The monthly magazine for all users of Commodore computers. Each issue contains reviews of software and peripherals, programming advice for beginners and advanced users, program listings, a technical advisory service and all the latest news. A year's subscription costs £10 in the UK and £16 overseas.

For further information contact:
Sunshine
12–13 Little Newport Street
London WC2H 7PP
01-437 4343

Telex: 296275

Printed in England by Commercial Colour Press, London E7.

Logo is the computer language of the future. In schools around the world it is Logo that is introducing young people to computers and, as they progress, helping them to realise the power at their fingertips. With advanced features that set it far in advance of other micro languages Logo is a tool for beginners and experts alike.

The author shows how Logo can liberate the full potential of your Commodore 64. The built in procedures for sound effects, the superb graphics capabilities and the ability to manipulate sprites are fully discussed together with a section on writing your own adventures.

For teachers, students and home micro owners who want to go beyond BASIC, Boris Allan's book is an essential guide book for the journey.

Boris Allan is one of the leading experts and enthusiasts in the field of micro languages. His books and articles have a worldwide audience among computer professionals and hobbyists alike. He is the author of the best selling Graphic Art on the Commodore 64.

GB £ NET +006.95
ISBN 0-946408-48-3
00695
9 780946 408481

SUNSHINE

ISBN 0 946408 48 3

£6.95 net