PAPERMAC

COMPUTER LIBRARY

LOGO

ON THE SINCLAIR SPECTRUM

Graham Field

- A complete operating guide to Sinclair LOGO
- Practical information on every aspect – from getting started to turtle graphics and list processing
- Clear, concise explanations to maximise the simplicity and power of LOGO

LOGO
ON THE SINCLAIR SPECTRUM

Other titles in the Papermac Computer Library:

Women and Computing: The Golden Opportunity
by Rose Deakin

How to Buy Software
by Alfred Glossbrenner

LOGO
ON THE SINCLAIR SPECTRUM

GRAHAM FIELD

MACMILLAN

First published 1985 by
PAPERMAC
a division of Macmillan Publishers Limited
4 Little Essex Street London WC2R 3LF
and Basingstoke

Associated companies in Auckland, Delhi, Dublin, Gaborone, Hamburg, Harare, Hong Kong, Johannesburg, Kuala Lumpur, Lagos, Manzini, Melbourne, Mexico City, Nairobi, New York, Singapore and Tokyo

Series consultant: Ray Hammond

British Library Cataloguing in Publication Data

Field, Graham
LOGO for the Sinclair Spectrum. – (Papermac)
1. Sinclair ZX Spectrum (Computer) –
Programming 2. LOGO (Computer program language)
1. Title
001.64'24 QA76.8.S625

ISBN 0-33-38376-1

Typeset by Bookworm Typesetting, Manchester
Printed in Great Britain by
Richard Clay (The Chaucer Press) Ltd
Bungay, Suffolk

Contents

Acknowledgements

I should like to thank Sinclair Research Ltd and especially Julian Goldsmith for help with the provision of software and documentation and all my colleagues of the ITMA Collaboration for their support and suggestions. Thanks are particularly due to Doreen Johns who typed the manuscript for the care and thoroughness which she brought to this task.

1

What is LOGO?

LOGO is a computer language developed at the Massachusetts Institute of Technology from 1967 onwards as a suitable way for children to learn programming and with the belief that it would contribute to the learning of mathematics and the development of problem-solving skills. Its use in this way has been the subject of considerable successful research at MIT and other places in the United States and at the University of Edinburgh, which also reports an improvement in word handling and language skills of children who are familiar with LOGO. For teachers, parents and even pupils interested in the educational aspects of LOGO there are a number of excellent books available, notably *Mindstorms* by Seymore Papert (Harvester Press, 1980), one of the language's originators.

Despite its design for the use of children, LOGO should be considered a serious computer language, not a toy. Possessing a number of advantages over BASIC, it is certainly a possible replacement for that language for the home or school computer. Although LOGO is easy to learn and use, and therefore suitable for children – or for adults unused to computers – it is also a powerful tool for controlling the full capabilities of the computer.

Turtles and lists

Young children are usually introduced to LOGO by making use of the computer to control the movement of a turtle – a small mechanical cart,

usually connected by wires to the computer, which can be moved around on the floor, preferably on a large sheet of paper. The original turtles were covered by a domed 'shell', hence the name. This device carries on its underside a pen which can be raised or lowered so that as the turtle moves it can draw its path. In order to control the turtle, you must present the computer with an ordered and organized series of instructions, its program, which can be neatly stated in LOGO.

Although the Spectrum has facilities for the control of a floor turtle, we shall be thinking of the term as referring to a small arrowhead displayed on the TV screen, where it can both draw lines like a floor turtle and also change colours and rub lines out.

The fascination and power of LOGO as a computer language derive from the fact that once you have told it how to do something you can give that task a name and then use that name as a new command (called a *procedure*). So you effectively construct your own commands and your own computer language, which develops as your understanding does. The only restriction is the amount of memory space in the computer needed to store your procedures. You can, however, store them on cassette or microdrive, only having in main memory those you need at the time.

LOGO was originally developed and subsequently used by research workers in the field of artificial intelligence – the study of human thinking and how machines can imitate it and throw light on how it works. This study often makes use of a computer language called LISP (a LISt Processing language). A list is a natural and easy way to store information, and *list processing* is the term used to describe the methods of handling lists. It is unfortunately the case that LISP has proved difficult to learn, possibly because it looks more complicated than it is and because it still incorporates some code words used for the first computer on which it was implemented, even though this machine no longer exists. This has led to the idea that list processing is mysterious and difficult to understand. That this is not so will, I hope, become apparent as you work through this book, for LOGO provides straightforward and easily understood list processing commands which children have used to write poetry-generating programs and teaching programs.

It is regrettable that some people have come to regard turtle graphics as the most important aspect of LOGO. There even seem to be a few who believe it is the only important aspect of the language. As a result there have been a number of programs developed which provide turtle graphics – often very worthwhile as programs, and written to a high standard – many of which are quite wrongly called LOGO, or incorporate the word in their names. LOGO for the Sinclair Spectrum does not belong in this category; it provides all the facilities to be expected of a full implementation of the language and adds some – access to machine code routines, for instance – which are not part of LOGO proper but may enable you to write more ambitious programs. These extra techniques are outlined in LOGO 2 but are not covered in this book.

Like other computer languages and indeed all branches of computing, LOGO is described using a lot of jargon words. Many people seem to hate or fear jargon, unless it is connected with their own jobs or areas about which they are knowledgeable. Some jargon is, however, necessary if we want to discuss certain aspects of the language and show how they relate to others or develop techniques which depend on knowledge of how information is represented and organized. It does not help you to drive a car to know that the four things sticking out of the top of the engine are spark plugs – at least, not until you find that the car will not start; then such knowledge will help someone explain to you what steps to take to improve the situation, and it is essential when it comes to ordering spare parts. This book is not written entirely in jargon, nor does it attempt to insulate you from it, but it sets out to give a simple explanation of all the technical terms used when they arise. One item of jargon, or a specialist way of writing, needs to be explained immediately. We have already used the word 'LOGO' to refer both to the language we will be using and the program, which you load in from tape, that understands the language and uses it to control the computer. This should cause no real confusion, as only rarely will it be necessary for you to know which is which.

LOGO and BASIC

Now for those acquainted with BASIC, a short comparison of the two languages. If you are not used to using BASIC you can skip this section.

The two languages differ in a number of ways. Possibly the most obvious to someone meeting LOGO for the first time is that it does not have line numbers; there is no need for them. If you need to change part of what you have written in LOGO you use an *editor*, a special program or part of the LOGO system written for that purpose. Unlike the BASIC line editor, this handles a whole procedure or more at once, and provides more extensive editing features than are available with BASIC. LOGO procedures tend to be shorter, often very much shorter, than BASIC programs; this makes them easier to handle, to write and to understand. Although the editor will take more than a screenful of instructions, it is usually not a good idea to put so much into a single procedure.

Another feature which will cause some surprise if you are used to BASIC is that while you might expect just one BASIC program to be in memory at a time, LOGO can have a number of procedures that may be quite unconnected with each other stored simultaneously in the computer memory.

A difference which will perhaps not be obvious at first is that each BASIC instruction has its own structure and always has to be written in this way. LOGO, on the other hand, has only one structure for its instructions with two

minor variations which do not have to be used but are allowed for the convenience of the user.

If you have got used to entering BASIC keywords by means of single key presses, safe in the knowledge that BASIC will know whether it is expecting a keyword or not, you will be disappointed to learn that the structure of LOGO does not make this approach possible; you will have to adapt to spelling words out more fully. Many of the built-in instructions, however, have abbreviations, and the nature of LOGO is such that you can define abbreviations for any others if you think them necessary. There are more fundamental differences in design of the two languages that can only be appreciated after using them, but I hope enough has been said here to prevent you from thinking of LOGO as just another form of BASIC.

LOGO dialects

If you examine LOGO procedures written out for other computers you may think they look very different from those given in this book or in the manuals with Spectrum LOGO. This is because LOGO seems to have produced more and more different versions or dialects than any other computer language. This is unfortunate, but there are some who would argue that LOGO is not so much a programming language as a way of thinking. Certainly the general structure will be similar, as will be the things which LOGO will do. With a little thought, you will be able to translate LOGO programs that you find elsewhere for other computers. The words may be different but the ideas are the same.

How to use this book

When writing LOGO it is customary for books to use capital letters and to indent sections for clarity. I shall adopt the same practices but it will not be necessary for you to do so as the version of LOGO available for the Spectrum is quite flexible. Lower-case letters may be used without problems and indentation is merely a convenient way of making procedures easy to read. So, when I have something like

```
TO SAYHELLO
    PRINT [WELCOME TO SINCLAIR LOGO AND
           A NEW EXPERIENCE]
END
```

your screen will show:

```
?to sayhello
>print [welcome to Sinclair logo a!
>nd a new experience]
>end
```

The ? and > signs at the beginning of the lines and the ! at the end of the second are provided by LOGO and will be explained at the appropriate time.

As mentioned above, many of the built-in instructions have abbreviations provided. In my examples I shall avoid their use so as to make things as clear as possible. *LOGO 2* shows clearly which instructions have abbreviations, and a full list is given on the reference card which accompanies the manuals.

This book is intended to provide a full introduction to Sinclair LOGO, so Chapters 2 and 3 repeat much of the information from LOGO 1, partly for completeness and partly to provide fuller explanation. If you feel you have already mastered this and are simply seeking for further information or ideas, I nevertheless suggest you read quickly through those chapters, particularly the last part of Chapter 3, to acquaint yourself with what is there before starting Chapter 4.

Chapter 10 can be read at any time, although it uses ideas from a number of previous chapters; you may need to debug (jargon meaning 'find and remove the errors in') your procedures at any stage.

You are advised to try all the examples given, for only by seeing how LOGO reacts to your instructions, and mistakes, can you begin to understand how to use it effectively. If any changes or improvements occur to you, you should try them. All the examples, except possibly the simplest, are deliberately left open to improvement and development. Some ideas for improvements are suggested in the exercises which are given throughout the book. No 'answers' are provided for these; if LOGO accepts it and does what you wanted, then your procedure is 'right'. The exercises should be regarded as suggestions for investigation and experiment only.

Now all that remains is to begin.

2

Getting Started

Unlike BASIC, which is built into the Spectrum, LOGO must be loaded in from cassette tape. Consult your Spectrum introduction manual for the details of connecting up computer and cassette recorder and loading programs before doing this. The procedure, which is also to be found in *LOGO 1*, is as follows. Put the LOGO cassette into the cassette recorder. Type:

LOAD"

(Press J. on the keyboard, followed by symbol-shift and P, twice.) Now press the ENTER key and then PLAY on the cassette recorder. When the program is found on the tape, a moving striped pattern will appear on the screen border and a star-like design will build up in the centre. Eventually, when the program has finished loading, you will see displayed on the screen the message:

WELCOME TO SINCLAIR LOGO

followed by a copyright message. Below this should be a question mark, used as a 'prompt' to show that LOGO is ready for your instructions, and a small flashing square, called the *cursor*, which shows where you are to type them. In addition, in the bottom right-hand corner is a small letter 'l' which indicates that LOGO expects you to type a letter or number which will appear on the screen. You may now stop the cassette recorder and begin.

We will start by experimenting with turtle graphics. Type SHOWTURTLE and press the ENTER key. You will probably type in small letters (lower case) but LOGO will not mind. I will always use capitals (upper case) to indicate what you type in or what LOGO replies. You will see the turtle appear as a small arrowhead in the centre of the screen. At the same time, you will find that the screen splits to give you two lines at the bottom in which to type instructions while the rest of the screen is for the turtle's drawings. From now on I will stop reminding you to press ENTER at the end of an instruction and, except for certain cases which will be explained when we come to them, one line on the page will be one line of LOGO, ended with ENTER.

Try the instruction:

FORWARD 50

You should see the turtle move in the direction that it was pointing. It is important to realize that LOGO expects your instructions to it to consist of words, separated from other words by spaces just as in written English. So if you have been used to a programming language like BASIC and thought you could type:

FORWARD50

you will have found that LOGO could not understand you. Whichever of the above two instructions you used, try the other one now, so that you can appreciate the difference. When the error message is displayed an upward-pointing arrow appears in the bottom right-hand corner; it means that you must press a key (any key) before continuing.

In the same way,

BACK 50

will cause the turtle to move in the opposite direction, the same distance. 50 is a number used to tell the turtle how far to move. This is called the *input* to FORWARD or BACKWARD, while these words, the names of things to do, are called *procedures*. It is difficult to say what units the number 50 is measured in, since that will depend on the size of your television screen, but there are 175 'turtle steps' up the screen and 255 across.

Try going forwards and backwards by varying amounts to get used to the distances involved. Watch what happens when the turtle goes off the edge of the screen.

When you've had enough of movement forwards and backwards in a straight line, it is time to try a different direction. First type:

CLEARSCREEN

This instruction, not unsurprisingly, clears the screen and also moves the turtle back to its starting position in the centre. To turn the turtle we use RIGHT or LEFT according to which way we want it to turn. Each of these has an input, the angle in degrees through which the turtle is to turn. Remember that there are 360 degrees in a complete circle, so that

RIGHT 180

would make the turtle 'about turn', and

LEFT 90

will turn it through a quarter turn. Now try:

```
FORWARD 50
RIGHT 120
FORWARD 50
RIGHT 120
FORWARD 50
```

You might like to add another RIGHT 120 to finish with the turtle pointing in its original direction. It might occur to you that there must be an easier way to draw a triangle; using this method, pentagons (five sides) or decagons (ten sides) would get a bit laborious. After all, all we have done is repeat a pair of instructions three times. Fortunately, LOGO provides an easy way to do just that. Use CLEARSCREEN to start again, and this time put in:

```
REPEAT 3 [FORWARD 50 RIGHT 120]
```

Use square brackets, not round ones (press SYS Y to get the open square bracket and SYS U to get the close bracket). The different kinds of brackets have, as we shall see later, different meanings in LOGO; for the present, look upon the square brackets as enclosing the group of instructions which are to be repeated. The above instruction should again draw a triangle.

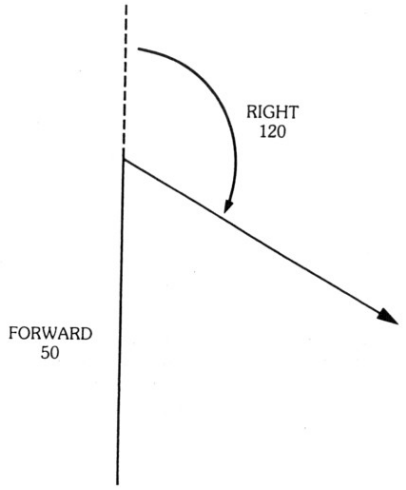

Figure 1 *The effect of FORWARD 50 RIGHT 120 FORWARD 50*

A REPEAT instruction can get rather long. To make them easier for you to read, the line will be split at a convenient place and the remainder indented further than usual. This will be a sign that you are to type it in as a single LOGO line. Don't worry if your typing reaches the end of a line on the screen and begins another; LOGO will not recognize the completion of the instruction until you press ENTER. It will, however, put an exclamation mark at the end of the first screen line to show that it continues below. So, for instance, when in this book you see something like

```
REPEAT 8  [FORWARD 20 RIGHT 90
           FORWARD 10 RIGHT 90
           FORWARD 20 LEFT 45]
```

it should be typed as a single LOGO line with only one use of the ENTER key, at the end.

The great power of LOGO lies not in the ease of repetition but in the ability it gives you to define new instructions. Since you now know how to draw a triangle, all that remains is to tell LOGO how.

Type:

```
TEXTSCREEN
```

This removes the turtle arrowhead and makes all twenty-two lines of the screen available for text. Now type:

```
TO TRIANGLE
```

You will see a new prompt, >, appear. Type in the instruction:

```
REPEAT 3  [FORWARD 50 RIGHT 120]
```

This time the instruction is not obeyed. This is because the word TO has told LOGO that you are telling it how to TRIANGLE (you are in what is called the 'TO' mode of use). Instead another > sign appears. This is because your instructions may consist of a number of lines of text. To signal that you have come to the end, type:

```
END
```

LOGO will respond:

```
TRIANGLE defined
```

and display its ? prompt to show that it is ready for instructions; this is called the *command* mode. Use SHOWTURTLE and then type:

```
TRIANGLE
```

If all is well your triangle should be drawn. If not, try again. Like FORWARD and BACK, TRIANGLE is a procedure and can be used like them as an instruction to LOGO; it may even be incorporated into more complicated commands or other procedures. No book on LOGO can be complete without rotating patterns, so clear the screen again and try this one:

```
REPEAT 12  [TRIANGLE RIGHT 30]
```

See how the instruction TRIANGLE is now understood by LOGO.

Exercise:
Convert this into a procedure, giving it a suitable name.

As an example of how procedures are combined, we will draw a stick man or, to avoid allegations of male bias, a slim woman in trousers. He or she consists of a head, a vertical line as a body, arms and legs. So, starting with the turtle in the home position pointing upwards, we would want to define the shape as:

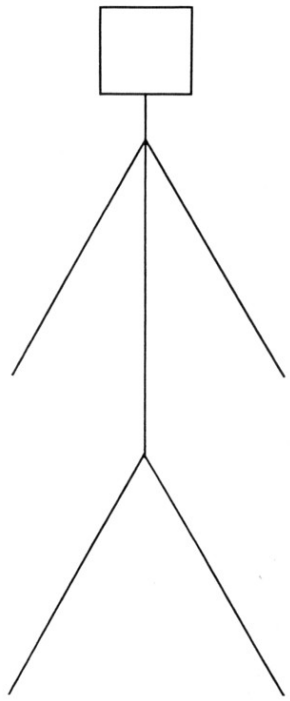

Figure 2 *Design for a stick figure*


```
TO MAN
   HEAD
   BODY
   LEGS
   ARMS
END
```

You can type it in just like that, or ignoring the indentation of the middle lines which I have used for clarity. Don't be worried that neither we nor LOGO know yet how to draw HEAD, BODY, ARMS and LEGS; we will simply write procedures for each of these in turn. If at any stage you are unsure about what a particular instruction does, try typing the instructions of the procedure one by one in command mode to observe their effects.

We can draw the head as a square for convenience. The basic design will be found in *LOGO 1* and it bears a considerable resemblance to the triangle we drew above. When it is finished, however, we will need to position the turtle in the centre of the bottom edge pointing down ready to draw the body; this requires three extra instructions.

```
TO HEAD
   REPEAT 4 [FORWARD 20 RIGHT 90]
   RIGHT 90
   FORWARD 10
   RIGHT 90
END
```

The body is easy:

```
TO BODY
   FORWARD 40
END
```

To draw the legs, as a downward V shape, turn slightly to the left, move the turtle forward and back to draw a line and return to the same position; turn twice the previous angle to the right and repeat the same process. We will finish by turning once more to the left so that the turtle again points vertically down.

```
TO LEGS
   LEFT 30
   FORWARD 30
   BACK 30
   RIGHT 60
   FORWARD 30
   BACK 30
   LEFT 30
END
```

The arms can simply be dealt with by moving back up the body and drawing another pair of legs!

```
TO ARMS
   BACK 35
   LEGS
END
```

Notice how the original problem was broken down into simpler parts, then each of these was considered separately. This is called *top-down programming*. It is a method of solving problems to which LOGO is particularly suited.

If at any time you want to make a copy of a turtle drawing on the printer, enter the command

```
COPYSCREEN
```

This causes the printer to copy everything on the top twenty-two lines of the screen.

Editing

If you have made no mistakes so far you have been very fortunate, but you will need to know how to correct or improve your procedures in the future. You can change a procedure by means of the editor. To use it, type EDIT followed by a space, inverted commas and the name of the procedure you want to change; for instance:

```
EDIT "LEGS
```

The screen will display the LEGS procedure. Try moving the cursor, the small flashing block, around the screen using the CAPS key with one of the keys 5, 6, 7 or 8. Now type some extra characters, for instance PRINT "HELLO. They should be inserted into the procedure at the position of the cursor. Now use DELETE (CAPS 0) to delete them one by one. Pressing ENTER produces a new line at the point where the cursor is. Think of this as producing a 'new line' character which you cannot see but which instructs LOGO to put the following text on the next screen line. This character can also be deleted like any other, causing two lines to join together.

Extra facilities are available in the editor using what is termed E MODE; to get it press CAPS and SYS at the same time; a letter E will appear in the bottom right-hand corner. The next key pressed will then have an effect different from usual. You will find that E MODE, followed by CAPS and keys 5, 6, 7 or 8, moves the cursor by greater amounts. In this case, 5 and 8 move the cursor to the beginning and end of the line respectively while 6 and 7 move it to the beginning and end of the screen. With longer procedures which take up more than one screenful, E MODE B and E MODE E can be used to move the cursor to the beginning and end of the text respectively (remember that E MODE B means select E MODE and then press B). A great convenience is the ability to move lines around inside the procedure. Move the cursor to the beginning of the line RIGHT 60, press E MODE Y. The line disappears. Move the cursor two lines down and press E MODE R to put

'RIGHT 60' in this position. Imagine this pair of instructions as picking a whole line up in one place and putting it down in another.

Don't be afraid to play with the editor at this stage so that you get to know how it behaves. All editors have their peculiarities and time spent developing understanding by using this one now will be well repaid in the future.

You can leave the editor in one of two ways. If you want LOGO to ignore all the changes you have made and return to the original version of the procedure, press CAPS and BREAK/SPACE simultaneously. On the other hand, if you want LOGO to incorporate your changes into the procedure you have been editing, use E MODE C.

You can use all the editing keys outside the editor when typing in instructions or data, provided you always stay within one LOGO line. So if you have reached the end of a long input line and, before you press ENTER, notice there is an error near its start, you can move the cursor back, make the change and replace the cursor at the end of the line without difficulty.

If you type EDIT followed by the name of a procedure which has not yet been defined (remember to include the quotes mark before the name), LOGO will assume you want to define it and enter the editor showing TO followed by the procedure name and a blank screen. This gives you an alternative way of defining a procedure to the use of TO on its own. In writing procedures for you to copy, I shall continue to start them with TO, but it is probably better for you to use the EDIT method because this puts the full resources of the editor at your disposal – including, for instance, the opportunity to go back and insert lines when you realize that you have forgotten them. (If you think it is unlikely you will need this, you probably haven't been programming very long.)

Storing programs on cassette

We have now got to the point where you might want to keep the procedures you have written, for future use. Remember that when your Spectrum is switched off it will 'forget' what is in its memory, including both LOGO and your procedures. Even if you do not want to save what you have written so far, it is better to try doing so now to make sure that everything works and that you understand the process, rather than wait until you have written an earth-shattering collection of procedures that you don't want to risk losing.

Like most computer languages, LOGO makes use of the idea of a *file*, a 'chunk' of information recorded on cassette tape (or microdrive) and identified by giving it a name. File-names, unlike other words in LOGO, cannot be more than seven letters long, a limitation imposed by the Spectrum's filing system.

For instance, to save the procedures to draw the stick figure in a file called PERSON, type:

SAVE "PERSON [MAN HEAD BODY LEGS ARMS]

Start the cassette tape by pressing the PLAY and RECORD keys on the recorder. LOGO meanwhile will tell you to press any key when you are ready. When the tape is moving – be sure that it has got past the plain 'header' section at the start of the tape which is non-magnetic and won't record anything – press a key on the keyboard. As usual, the border of the television screen will flash while the file is being saved. When it stops flashing and the prompt, ?, appears once more, stop the tape recorder.

To get the procedures back from the cassette tape into memory, rewind the tape, type:

LOAD "PERSON

and press PLAY on the recorder. You will see the usual striped pattern around the screen border as the file loads.

Microdrive owners will need to tell LOGO to use the drive rather than cassette. This is done by giving the instruction

SETDRIVE 1

to select microdrive number 1. You need to give this instruction only once as it will remain in force until either you type:

SETDRIVE 0

to change to the cassette system, or you switch off.

You can save more than one file on a cassette, but be sure to leave a gap between the end of one and the start of the next. It is a good idea to write down what files are on the cassette and the approximate tape-counter reading for the start of each one.

Managing the workspace

Part of the computer's memory is used to store the LOGO program itself. Some more is used by LOGO to store information that it must use either to keep track of what it is doing (we will see just how complicated and important that is at a later stage) or to hold intermediate results temporarily (for instance, calculating 2 + 5 – 3 means working out 2 + 5 first and remembering it ready for the next part of the calculation). The rest of the memory is used to store information and procedures for you; it is called your *workspace*. It is important to know what procedures etc. you have stored. Sooner or later every really imaginative programmer will start to run short of memory space. The solution may be to get rid of the procedures you do not want.

Most of the instructions for handling procedures have names which seem amusing but meaningless. Possibly this helps you remember them easily! You can try them out only if you have a few procedures in memory, so either type some in or load them from cassette tape. Remember that we may want to practise deleting some, so either type in procedures you don't want to keep or record them first.

Try the instruction:

POTS

You will get something like:

TO TRIANGLE
TO MAN
TO HEAD
TO ARMS
TO BODY
TO LEGS

As you can see, this instruction, which stands for Print Out Titles, tells you the names of all the procedures that you have defined. LOGO, of course, knows many more, like FORWARD and BACK, but as they are always present, being 'built in' (called *primitive procedures*), you are not told about them by POTS.

You can look at the instructions in all the procedures you have defined by means of the instruction:

POPS

This stands for Print Out Procedures and does exactly that, showing you the definition of each procedure, one after another. You can look at one procedure on its own either by using the editor, as explained above, or by means of the instruction PO (Print Out) followed by the name of the procedure to be printed, for instance:

PO "LEGS

These two are the commands to use if you want a printed copy of your procedures or a procedure. First make sure your printer is correctly connected and then type:

PRINTON

which will instruct LOGO to copy all the text that appears on the screen from now on to the printer. So you can now type:

PO "LEGS

for a copy on the printer (this is called *hardcopy*). Use the instruction:

PRINTOFF

to stop producing a printed copy.

To erase a procedure from your workspace, simply type:

ERASE "HEAD

or the name of whatever procedure you wish to remove. Try this and then use POTS to verify that the procedure has disappeared.

Finally, should you wish to clear all your workspace to start again, the instruction:

ERPS

(standing for ERase ProcedureS) will remove all procedures from memory.

All the commands in this chapter which have been able to take a procedure name preceded by a double quotes mark , ", as an input – namely, EDIT, PO

and ERASE – can also have a list of procedures as their input. In this case, the list is enclosed in square brackets exactly as in the case of SAVE and the instruction applies to all the procedures in the list, in the order that they are found in it. So

```
EDIT [LEGS ARMS]
```

will give you two procedures to edit at once (useful if you want to move a line from one to another);

```
PO [HEAD BODY]
```

will print out both, with the definition of HEAD being first; and

```
ERASE [MAN TRIANGLE]
```

will remove both those procedures from the workspace.

Printing text

If you want to print a message on the screen (or on the printer by using PRINTON as well), LOGO provides the instruction PRINT. If you wish to print a single word it may follow the instruction with a double quotes mark in front of it:

```
PRINT "HELLO
```

This time it does matter if you use capital letters or not after the quotes but only because LOGO will print exactly what you type in. For a sentence (or list of words) enclose them in square brackets:

```
PRINT [WELCOME TO LOGO]
```

If you want to print a blank line use PRINT [] or PRINT".

We will illustrate this instruction with a simple procedure. If you have not studied LOGO 1, it may pose a problem as to what is happening. If you have, it may serve to illustrate and explain an important idea:

```
TO FINEGAN
  PRINT []
  PRINT [THERE WAS AN OLD MAN CALLED MICHAEL
         FINEGAN]
  PRINT [HE GREW WHISKERS ON HIS CHINEGAN]
  PRINT [THE WIND CAME UP AND BLEW THEM INEGAN]
  PRINT [POOR OLD MICHAEL FINEGAN]
  PRINT [BEGIN AGAIN]
  FINEGAN
END
```

Run this procedure by typing:

```
FINEGAN
```

Stop it by pressing CAPS and BREAK/SPACE at the same time.

3

Words and Variables

In LOGO, as in most other computer languages, we can use words to represent other information such as numbers. The reason we want to be able to do this is that this is how we normally convey general instructions. For instance, in a procedure to handle the calculation of VAT we might want to talk about the price of an article. Let us suppose that the price is £20. Then the LOGO instruction:

```
MAKE "PRICE 20
```

makes the word PRICE stand for the number 20. The double quotes mark indicates that PRICE is a word and not the name of a procedure; that is, not an instruction to do something.

Now try:

```
PRINT "PRICE
```

LOGO, not unnaturally, prints the word PRICE. Again, the quotes mark shows that PRICE is just a word. However, there is a special symbol to indicate that we want the thing which PRICE represents, it is a colon, :. Try the instruction:

```
PRINT :PRICE
```

This time you should get the number 20 printed out on your screen. So " means the word; : means the thing it stands for. In the same way, we can translate the instruction 'VAT is 15% of the price' straight into LOGO as:

```
MAKE "VAT :PRICE * 15 / 100
```

The MAKE command expects two inputs; the first should be a word which is going to stand for something else, often a number, the second, as you might expect, is the thing to be represented by the word. Remember to put quotes, ", before the word which follows MAKE. There are occasions when these quotes are not needed but they are rather special.

Notice that the word PRICE, in the calculation which follows, is preceded by a colon, :. This is because it is the *thing* (the number 20) which we want to multiply by 15 and divide by 100, not the *word* PRICE. As you can see, * and / are the symbols for multiply and divide, respectively. Try this instruction and then type in:

```
PRINT :VAT
```

– the colon, remember, because we want the value, not the word.

In computer terminology, the words PRICE and VAT are called *variables* because they represent unknown, and possibly changing, things. To see a variable used to represent a varying length, try this procedure:

```
TO TRISPI
    SHOWTURTLE
    MAKE "LENGTH 10
    REPEAT 21 [FORWARD :LENGTH RIGHT 120
                MAKE "LENGTH :LENGTH + 5]
END
```

(Remember to enter the REPEAT instruction as a single line.) Look carefully at the last instruction in the REPEAT brackets. It tells LOGO to make the word LENGTH represent a new value which is found by taking the number which LENGTH now represents (:LENGTH) and adding 5 to it. Try using the procedure to see what happens.

When handling variables, LOGO distinguishes between names in upper and lower case; that is

```
MAKE "VAT 7
MAKE "vat 4
```

refer to two different things;

```
PRINT :VAT
```

gives 7; and

```
PRINT :vat
```

gives 4. This can cause a problem if you type in lower case and forget the quotes or colon. If, as part of a procedure, you type in:

```
print vat
```

and later edit the procedure, you will find that LOGO has changed both words to capitals because it thought that both were the names of procedures:

```
PRINT VAT
```

It will not be sufficient to put a colon before VAT; you will have to change it back to lower-case letters as well!

Arithmetic

Corresponding to the mathematical signs which we have already met, *, / and +, are named procedures which do the same or similar things. SUM adds up its inputs. Normally it expects there to be two of them but it can have more; in this case, you must put round brackets around both SUM and its inputs:

```
PRINT SUM 2 3
5
PRINT (SUM 1 2 3 4 5)
15
```

PRODUCT behaves in the same way but multiplies all its inputs together. Again, it normally has two inputs but may have more if brackets are used:

```
PRINT PRODUCT 7 3
21
PRINT (PRODUCT 2 3 5 7)
210
```

A warning: a number of the built-in procedures can take more than two inputs by using round brackets in this way. This seems to cause a problem if used inside square brackets (for a REPEAT instruction, for instance) and it should be avoided.

DIV has two inputs only and gives the result of dividing the first by the second:

```
PRINT DIV 10 5
2
```

A warning: LOGO cannot divide by zero any more than we can; try:

```
PRINT DIV 5 0
```

to see what happens. Trying to divide by zero is a common cause of failure of programs written by beginning programmers, usually because what has been written is something like:

```
PRINT DIV :TOTAL :NUMBER
```

and the programmer has forgotten to check in case :NUMBER is zero.

For subtraction, you can only use the sign, –.

```
PRINT 12 – 4
8
```

The minus sign is used with two meanings. Between two numbers, it means subtract, but if used in front of a number it shows that the number is negative (so –4, for instance, represents the temperature which is four degrees below freezing point on the centigrade scale). LOGO will try to make sense of what you type in, and is quite successful at doing so. It will, for instance accept:

```
PRINT 12–4
```

or

```
PRINT 12 – 4
```

if you want to subtract 4 from 12. To subtract negative numbers you can use:

```
PRINT 12 – – 4
```

or

```
PRINT 12 --4
```

Nevertheless there are occasions when LOGO can be confused about the meaning of the minus sign. It is best, therefore, to get into the habit of always putting a space before a negative number and leaving no space between the minus sign and the following digit.

If you are not used to the idea of subtracting negative numbers, try a few more examples to get the hang of it.

You can combine the 'four rules' to get more complicated arithmetic expressions, for example:

```
PRINT 3 *4 + 2
14
```

It is important to know in what order LOGO will do the various operations. This is not just from left to right; for instance:

```
PRINT 2 + 3 * 4
```

will also give the answer 14. The rule is that arithmetic is done in the following order:

- Things in brackets.
- Division.
- Multiplication.
- Subtraction.
- Addition.

This means that you can use brackets, (), to alter the normal order of operations:

```
PRINT (2 + 3) * 4
```

gives 20 by calculating 2 + 3 first then multiplying by 4. When LOGO works out what is inside brackets it uses the same rules over again. If you are unsure of the order in which arithmetic will be done, always use brackets to show the order that you want, or do the calculations in short, simple steps. Two more examples:

```
MAKE "CENTIGRADE (:FAHRENHEIT -32) * 5 / 9
```

does the subtraction first, and

```
MAKE "FAHRENHEIT :CENTIGRADE * 9 / 5 + 32
```

divides by 5, multiplies by 9 and then adds 32.

It makes no difference to the value of the answer that division is done before multiplication, but it may help LOGO to prevent numbers from getting too large for it to cope with.

One of the most helpful things about variables is the way we can use them to define our own procedures with inputs. For instance, we might want a procedure SQUARE which would draw squares of different sizes according to an input. So SQUARE 50 would produce a square of side 50 units while SQUARE 100 would give one with a side of 100 units. We begin by telling

LOGO:

```
TO SQUARE :SIZE
```

meaning that SQUARE will have one input which we will call SIZE during the definition of the procedure. Notice the colon, telling LOGO that it is the value, the thing which SIZE represents, which matters, not the word itself. If you are using the editor, type EDIT SQUARE and then add :SIZE when in the editor.

```
TO SQUARE :SIZE
    REPEAT 4 [FORWARD :SIZE  RIGHT  90]
END
```

Now try using this to draw squares of different sizes.

Exercise:

Write a procedure with two inputs, :SIDE and :NUM, to draw a figure which has :NUM sides, each of length :SIDE. Call it POLYGON. You will need to change the procedure for SQUARE, given above, in two ways. The number of times the repetition is done – obvious, I hope – and the angle turned through each time. This is a bit harder, but notice that 90 times 4 is 360.

Before we get even more ambitious, here are some more useful turtle commands. Normally, when the turtle goes off the screen, it reappears immediately on the opposite side as if the two were connected together in some way – what is called the *wrap-round* effect. For some applications, you might want the turtle to stop when it reaches the edge of the screen. Use the command:

```
FENCE
```

to instruct that this is to happen. Alternatively, you might want the turtle to leave the screen, still obeying the same rules, and possibly come back later, as if the screen were a window looking down on to a much larger table that the turtle was moving on. You can use:

```
WINDOW
```

to tell LOGO that this is to be the case. With WINDOW in operation, the turtle will travel up to 32 767 turtle steps from the centre of the screen. To get back from either of these ways of using the screen to the original wrap-round, use the instruction:

```
WRAP
```

A number of instructions allow you to control the colours which appear on your screen. Firstly, we can move the turtle without drawing a line at all. The instruction to command the turtle not to draw is:

```
PENUP
```

and to start drawing again:

```
PENDOWN
```

If you remember that the turtle robot, on which the ideas are based, carries a pen on its underside and draws on sheets of paper on the floor, these commands will make more sense. The screen turtle can, however, do things

which are impossible for a floor turtle.

PENERASE

instructs the turtle to rub out any lines it comes across while moving, and

PENREVERSE

reverses whatever it finds, drawing lines where there are none and removing lines where there are. You can switch either of these effects off by using PENUP or PENDOWN.

The Spectrum uses eight colours which for convenience are numbered:

0 Black
1 Blue
2 Red
3 Magenta
4 Green
5 Cyan
6 Yellow
7 White

You can set the colour which the turtle is to use for drawing by instructing SETPC (SET Pen-Colour) followed by the number of the colour that you want:

SETPC 6

for instance. You will see that it changes the colour of the turtle too. Any line drawn in a new colour will, unfortunately, change any lines which fall within the same character space (the position occupied by a printed character on the screen). To allow anything otherwise would mean that a lot more memory space in the computer would have to be allocated to the screen (the colour of each point would have to be recorded rather than the colour of each character space), and this would reduce the amount of memory available for your procedures. LOGO also provides a procedure which will output the colour being used by the pen as a number. It is PENCOLOUR. You might use this with a MAKE command to record the colour being used at the start of a procedure:

MAKE "STARTCOLOUR PENCOLOUR

and, having drawn a shape in some other colour, return to the original at the end of the procedure with:

SETPC :STARTCOLOUR

Two more commands, SETBG (SET BackGround) and SETBORDER, control the colour of the background and the screen border respectively. Each must have a number as its input to specify the colour required. In addition, the operation BACKGROUND works like PENCOLOUR to output the number of the background colour.

Here are some procedures which seem to need the use of WINDOW to give their proper effect; you might like to incorporate some colour changes too. Firstly spirals, another essential ingredient of books about LOGO: let us

start by drawing a 'square' one. We do the instructions:

FORWARD :SIZE RIGHT 90

just as for a SQUARE, but then, to get the spiralling effect, increase the size of :SIZE. Unlike SQUARE, we want to go on repeating this for ever (or until you get tired of watching). How is this done? Remember FINEGAN?

```
TO SQUARESPI :SIZE
   FORWARD :SIZE
   RIGHT 90
   MAKE "SIZE :SIZE + 5
   SQUARESPI :SIZE
END
```

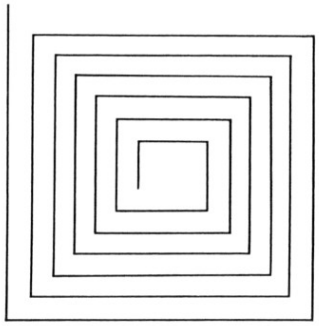

Figure 3 *SQUARESPI 10*

Remember that you can interrupt this by pressing CAPS and BREAK/SPACE keys at the same time. This method, in which a procedure uses itself as one of its own instructions – we say that it *calls* itself – is a useful way of getting things to repeat for ever. It is called *recursion*.

Spirals do not have to be square ones. You can modify the above procedure to give many different kinds of spiral by changing the angle which is turned through on each step. Perhaps you might like to have some control over the amount by which the distance moved increases as well; this is useful in case the turtle goes off the edge of the screen too soon.

Here is a spiral procedure with three inputs. SIZE will be the distance moved at first, ANGLE the angle turned through and INC the amount that the

distance is increased by.

```
TO SPIRAL :SIZE :ANGLE :INC
   FORWARD :SIZE
   RIGHT :ANGLE
   MAKE "SIZE :SIZE + :INC
   SPIRAL :SIZE :ANGLE :INC
END
```

Try this with some values. The most interesting spirals seem to be obtained with angles which are almost, but not quite, the angle of a polygon. You might like to try:

SPIRAL 10 121 5
or SPIRAL 10 119 5
or SPIRAL 10 135 5

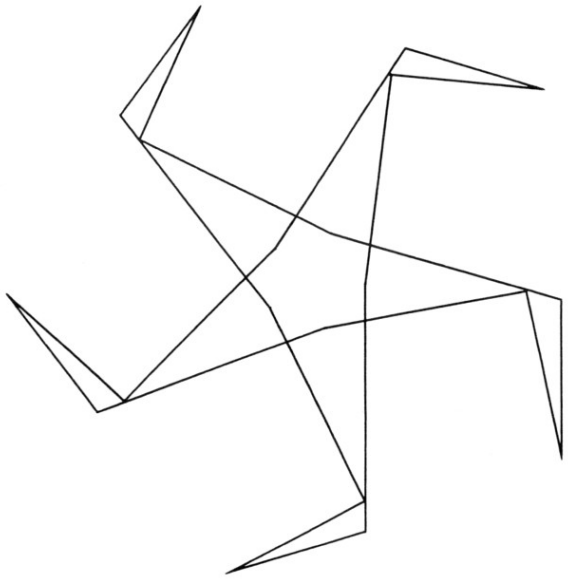

Figure 4 *ASPI 30 9 90*

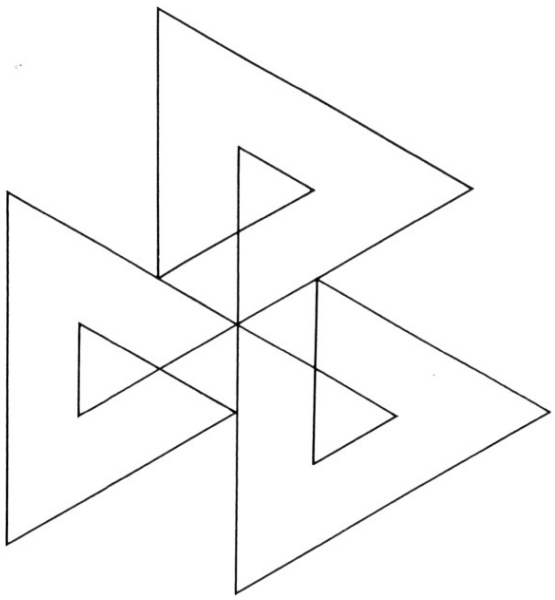

Figure 5 *RESPI 10 120 5*

Exercise:
To see what happens when the distance moved on each leg of the 'spiral' stays the same and the angle increases, write a procedure called ASPI with three inputs as above but with the increase added on to the size of the angle. Try it with:

ASPI 10 0 5
ASPI 10 40 30
ASPI 10 2 20

Equally interesting are the patterns formed by drawing part of a spiral, then beginning it again from the point that the turtle has now reached. In this case, instead of changing the value of SIZE (which we will need as the same starting

value each time), we go FORWARD by an amount which is :SIZE multiplied by a number called COUNT, which runs from 1 up to a maximum value:

```
TO RESPI :SIZE :ANGLE :MAX
   MAKE "COUNT 1
   REPEAT :MAX [FORWARD :SIZE * :COUNT
                RIGHT :ANGLE
                MAKE "COUNT :COUNT + 1]
   RESPI :SIZE :ANGLE :MAX
END
```

Words and things

We will now look more thoroughly at the reasons for all those colons and quotes marks. The ideas in the following section are not in themselves difficult, but they are more easily understood by experience than by explanation. If at first some of it seems a little complicated, don't worry – you can always look back at it when you have written a few more procedures.

LOGO handles four different kinds of LOGO *objects*:

- Words.
- Numbers.
- Procedures.
- Lists.

Lists will be dealt with in the next chapter.

Numbers are officially classed as words but since they are subject to slightly different rules it might be best to think of them as a different kind of object. Numbers are made up of digits (0 to 9) with possibly a minus sign at the start and a decimal point. Above 10 000 000, LOGO prints out numbers in exponent notation. Using, this, 2.3E+9 means 2 300 000 000; that is, the decimal point should be nine places to the right of where it is shown. For small numbers a negative sign is used: 1.4E-6 means 0.000 001 4 with the decimal point six places to the left. You can use this system to type in numbers if you wish. If a calculation will produce a number greater than 1E+38 the message:

```
Number too big
```

is produced. If a number is calculated to be smaller than 1E-38 it is considered to be 0.

LOGO normally assumes that any word it comes across is an instruction, the name of a procedure which it is to obey, and will try to find that procedure in memory before either obeying it or giving an error message if it cannot be found. So that LOGO will know the difference, a word which is just a word is preceded by a double quotes mark, ". So for instance:

```
"FRED
```

is a word and, if you type it in on its own, LOGO will be puzzled about what you want done with "FRED. Be careful not to leave a space between the quotes and the start of the word. This is because LOGO recognizes the existence of an empty word, without any letters in it. It is shown by means of the quotes mark followed by a space. This curious object is not quite as pointless as might at first seem. When we handle words by taking letters from them one by one, it will prove very useful as an indication of when to stop.

A word can also be the name of some other object. We have seen words used to name procedures and to name numbers. This last, remember, is done by the MAKE command:

```
MAKE "VAT 15 * 40 / 100
```

gives "VAT the value 6 (that is, 15 * 40 / 100). We say that 6 is the THING of which "VAT is the name. Try the above MAKE command and follow it by:

```
PRINT THING "VAT
```

This should give the result: 6. Because we so often need to refer to the thing which a word names, rather than the word itself, we can abbreviate THING to a colon placed immediately before the word, and omit the quotes mark:

```
PRINT :VAT
```

(Actually the colon is not a perfect abbreviation for THING for a reason which will be explained later.)

The MAKE command is always:

```
MAKE name thing
```

and because its first input must be a name, it is usually preceded by quotes. The second input may be a word, number or list.

```
MAKE "PRICE 27
PRINT :PRICE
27
```

27 is the THING corresponding to PRICE.

```
MAKE "MARY "CONTRARY
PRINT "MARY
MARY
```

The quotes mark means that the word is to be printed.

```
PRINT :MARY
CONTRARY
```

Here the colon means the THING named by MARY should be printed.

```
MAKE "JEAN :MARY
PRINT :JEAN
CONTRARY
```

The MAKE instruction gives the name JEAN to the THING that MARY names, so the word CONTRARY now has two names!

It may seem unnecessary to use a word to name another word, but this is not always so. You may want someone to type in his or her name during a procedure. You cannot know in advance what word will be put in so you must call it, for instance, ANSWER:

```
TO GREET
   PRINT [WHAT IS YOUR NAME]
   MAKE "ANSWER READLIST
   PRINT "HELLO
   PRINT :ANSWER
END
```

To handle the input here we have used a procedure, READLIST, which is covered more fully later. For the moment it is sufficient to say that it will get a word, or list of words, typed at the keyboard and pass it on, as its output to the MAKE command.

Just as the word 'John' names the man at the keyboard, so the word "ANSWER names the word, 'John', while it is in the computer memory. Now try these:

```
MAKE "GIRL "MARY
PRINT "GIRL
GIRL
PRINT :GIRL
MARY
PRINT THING :GIRL
CONTRARY
```

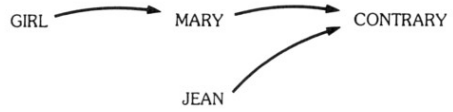

Figure 6 *Words and the things that they name*

Remember that :GIRL gives the result MARY, so in THING :GIRL you are really asking for THING "MARY. This is the way in which THING is different from the colon, you can have THING :GIRL or even THING THING "GIRL (try PRINTing these and see) but not ::GIRL.

Why should we need more than one level of name? Let us suppose that we have a procedure to handle costs of goods, and we want a procedure to print out the name of a quantity (VAT, PRICE etc.) and its value in pounds.

```
TO NAMEPRINT :OBJECT
   PRINT :OBJECT
   PRINT THING :OBJECT
END
```



```
MAKE "VAT 21
NAMEPRINT "VAT
VAT
21
```

The word OBJECT appears as a THING, preceded by a colon, throughout the procedure, because it really is the *thing* it represents (VAT in the example) that we want to deal with, not the word OBJECT. When we give the command NAMEPRINT "VAT we pass to the procedure the word "VAT (not its value but the word itself, which is why the quotes mark is used) as the thing which OBJECT is to name. In the next line the *thing* of OBJECT is printed out (that is, the word VAT). In the third line of the procedure the THING of *this* is printed, its value, 21.

Using the operation SENTENCE (dealt with more fully later) we can tidy up the output:

```
TO NAMEPRINT :OBJECT
   PRINT (SENTENCE "THE :OBJECT "IS THING :OBJECT)
END
MAKE "PRICE 140
NAMEPRINT "PRICE
THE PRICE IS 140
```

On occasions we may want to use a similar technique with the MAKE command, assigning a value to a variable indirectly. It is in these cases that the first input to MAKE does not have a quotes mark. For instance, if firstly you

```
MAKE "QUANTITY "PRICE
```

and then instruct

```
MAKE :QUANTITY 27
PRINT :PRICE
```

you will get:

```
27
```

Here the second MAKE operation assigns the value 27 to the THING of QUANTITY, i.e. to PRICE.

As we have already said, LOGO assumes that a word which is not a number and does not begin with " or : is an instruction to obey a procedure. So:

```
JUMP
```

will either do the procedure JUMP or, if you have not defined one, give an error message to say so. Should you wish to do something else with the procedure, the name needs to be introduced by quotes:

```
ED "JUMP
```

for instance, to edit it. Similar rules apply to PO, ER and SAVE.

Kinds of procedure

It is useful to distinguish two kinds of procedure: *commands*, which instruct the computer to perform some action, FORWARD 50, PRINT "HELLO, PENUP etc.; and *operations*, which output a result for another procedure to use, e.g. SUM or DIV.

In writing LOGO each procedure takes its inputs from the right, if it expects any, and each operation outputs its results to the left. For this reason, every LOGO line must start with a command; which is why when demonstrating the arithmetic operations the PRINT command was always included. The only exceptions to the 'input from the right, output to the left' rule are the arithmetic signs, +, −, /, >,< and =, which are placed between their two inputs (we will look at the last three in this list in a later chapter).

Procedures which are built into LOGO and not defined by you are called *primitives*.

Occasionally LOGO lines can look quite complicated:

MAKE "P LIST FIRST THING :A SENTENCE "PR WORD CHAR 34 :A

for instance. There is, however, a simple way to understand them. Firstly, pick out the names of all the procedures involved – they are the words which are not numbers and do not begin with quotes or a colon. Find out how many inputs each expects and write this under the name. You will be unfamiliar with most of those used in this example but they are all primitives explained in *LOGO 2*.

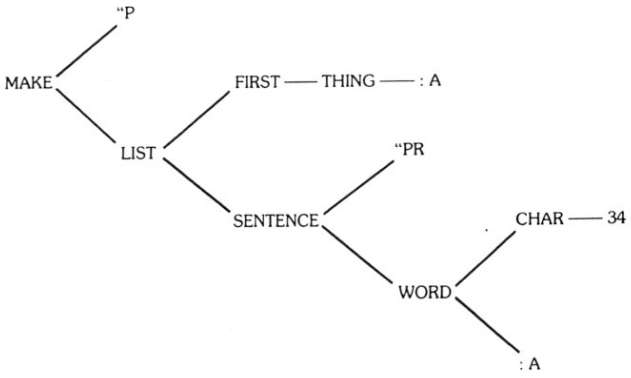

Figure 7 *Analysing a LOGO line*

MAKE "P LIST FIRST THING :A SENTENCE "PR WORD CHAR 34 :A
2 2 1 1 2 2 1

On a piece of paper write down the first word of the line, which must be the name of a procedure. From it draw lines, one for each of its inputs. Write the next word at the end of the first of these lines. If it is not a procedure go on to write the next word at the end of the next line. Otherwise draw lines for the inputs from it in the same way as before. Continue with this until you reach the end of the LOGO line. The result for this example is given in figure 7; it shows clearly where each procedure expects its inputs to come from. If this method does not work – if there are extra words at the end when all the input lines have been completed or there is a line without an input – then there is a fault in the instruction.

4

Lists and Words

We are all familiar with lists; for instance, a shopping list:

sausages peas butter stamps

As an object which LOGO is able to handle, a list is no different from what we are used to. LOGO likes to have the items of a list enclosed in square brackets and separated from each other by spaces. The order of the items within the list is important; that is

[JANUARY FEBRUARY MARCH]

is considered to be a different list from

[JANUARY MARCH FEBRUARY]

The items in a list are either words or other lists. We will deal with the latter possibility in a later chapter, confining our attention for the moment to lists of words. Normally it is not necessary to precede the words in a list by the quotes mark; LOGO will assume that

[JANUARY FEBRUARY MARCH]

is the same as

["JANUARY "FEBRUARY "MARCH]

Just as with numbers, we can give a name to a list, and this is done in the same way, by means of the MAKE command; for instance:

MAKE "SHOPPING [SAUSAGES PEAS BUTTER STAMPS]

which makes the word "SHOPPING represent the whole list which follows. The advantage of this is that we can handle the list as a whole by referring to it

by name; for instance:

PRINT :SHOPPING

will produce the output:

SAUSAGES PEAS BUTTER STAMPS

Notice that in printing out a list, LOGO misses off the outer square brackets. As with numbers, the THING of the word is what it represents, in this case, the words, taken in order, which make up our shopping list. We can therefore give the list another name if we wish:

MAKE "BOUGHT :SHOPPING
PRINT :BOUGHT
SAUSAGES PEAS BUTTER STAMPS

Selecting items

There would not be much point in using lists if all we could do was to handle them as whole things. We need some way of getting at individual items in a list. LOGO provides several ways in which we can do this. The simplest, and most standard, is the operation FIRST, which not surprisingly gives the first item on the list.

PRINT FIRST [JANUARY FEBRUARY MARCH]
JANUARY

FIRST can handle named lists also:

PRINT FIRST :SHOPPING
SAUSAGES

Now we need a way of looking at everything else in the list. This is done with BUTFIRST. It outputs a list which consists of all the items except the first.

PRINT BUTFIRST [JANUARY FEBRUARY MARCH]

gives the output:

FEBRUARY MARCH

and:

PRINT BUTFIRST :SHOPPING
PEAS BUTTER STAMPS

There is an important difference between FIRST and BUTFIRST; FIRST gives a word, BUTFIRST a list.

For convenience, LOGO also provides us with a means of getting at the items in a list from the other end. LAST outputs the last item in the list:

PRINT LAST :SHOPPING
STAMPS

It will not be a surprise to find that BUTLAST gives all items except the last one:

PRINT BUTLAST [MONDAY TUESDAY WEDNESDAY THURSDAY FRIDAY]
MONDAY TUESDAY WEDNESDAY THURSDAY

In order to pick out items between the first and the last, we use the ITEM operation. This has two inputs of which the first is the number specifying the item that is wanted and the second is the list that it is to be selected from. It outputs the corresponding word.

```
PRINT ITEM 2 [APRIL MAY JUNE JULY]
MAY
PRINT ITEM 3 :SHOPPING
BUTTER.
```

It will also be useful to know how many items there are in a list. This is done with COUNT. Its input is a list and its output the number of items in that list. You can test this with such examples as:

```
PRINT COUNT [JANUARY FEBRUARY MARCH]
3
PRINT COUNT :SHOPPING
4
```

Many standard LOGO procedures expect lists as their inputs. A number of them we have already seen, for instance the REPEAT instruction; it expects two inputs of which the first is a number and the second a list of instructions. It is one example when the quotes mark might actually be needed inside a list; for instance:

```
REPEAT 4 [MAKE "COUNTER :COUNTER + 1]
```

Here, although REPEAT takes a list as its input, the MAKE command, which is inside that list, needs to have its first input specified as a name. (Should you wish to try this instruction, be sure to give COUNTER some value first.) Some of the built-in procedures, primitives as they are correctly called, will handle either lists or words. An example of this is the PRINT command which can take as its input either a single word or a single list. This is why we have found that a sentence of several words to be printed out must be enclosed in square brackets while one word need only be preceded by its quotes mark.

We can also write our own procedures to handle lists. For instance, here is a procedure to take a list and print it out with the items on separate lines. It does this by repeating, for the number of items there are in the list, an instruction to print an item from the list. Picking out which item is required, we use the ITEM operation, controlled by a counter which is increased by one at every repetition.

```
TO LBL :ALIST
   MAKE "COUNTER 1
   REPEAT COUNT :ALIST [PRINT ITEM :COUNTER :ALIST
                        MAKE "COUNTER :COUNTER + 1]
END
LBL [LBL STANDS FOR LINE BY LINE]
```

will give the result:

```
LBL
STANDS
FOR
LINE
BY
LINE
```


Exercise:
Write a procedure to print a list, line by line, in reverse order.

With these tools we can use lists as stores for information. Let us take a simple example. If we want to store expenditures for the months of a year so that we can find their total and average, it is possible to put them into a list in which the first item is the expenditure for January, the second for February and so on. The procedure given below reads through the list, picking out each item in turn, in the same way as LBL. Instead of printing them out, though, it adds each to a running total which has to be set initially to zero. The average can then be found by dividing by twelve.

```
TO AVERAGE :ALIST
   MAKE "TOTAL 0
   MAKE "COUNTER 1
   REPEAT 12 [MAKE "TOTAL :TOTAL + ITEM :COUNTER :ALIST
              MAKE "COUNTER :COUNTER + 1]
   MAKE "AVERAGE :TOTAL / 12
   PRINT "TOTAL
   PRINT :TOTAL
   PRINT "AVERAGE
   PRINT :AVERAGE
END
```

Try this procedure with a suitable list of twelve numbers.

Exercise:
Modify the procedure AVERAGE so that it will work with a list of any length not just twelve. Write a procedure which will request a number from one to twelve and print out the expenditure for that month. (Use FIRST READLIST to provide the number.)

Typing in lists and words

A list can be obtained from the keyboard by the operation READLIST. It is necessary to pass its output list on to some other procedure, usually the MAKE command.

```
MAKE "ANSWER READLIST
```

causes the computer to pause until you type something in at the keyboard. It indicates this by printing a question mark. The input list then consists of the words that you type before pressing ENTER. Try the above instruction and

verify that what you have typed in is now called ANSWER by typing:

```
PRINT :ANSWER
```

LOGO does not provide a corresponding operation READWORD, but you can produce your own easily enough. The standard technique is to use FIRST READLIST, which will, of course, pass on the first word typed in and cause the rest to be ignored. You cannot use

```
MAKE "START FIRST READLIST
MAKE "REST BUTFIRST READLIST
```

to provide the two parts of an input list. This is because the two uses of the READLIST procedure will each cause the computer to pause for input, so you will get two parts of two separate lists. If you want to split the input up you must use

```
MAKE "ANSWER READLIST
MAKE "START FIRST :ANSWER
MAKE "REST BUTFIRST :ANSWER
```

The empty list

An unusual but occasionally useful object is a list without any items in it. It is indicated by [] and called the *empty list*. We will need to use it as a starting point for building up lists and, later, as an end point to be tested for when breaking them down one item at a time.

Joining and building lists

The procedure SENTENCE, which we have already used in conjunction with the PRINT command, takes as its inputs two or more words or lists and outputs a single list. So, for instance:

```
MAKE "PROVERB SENTENCE "MANY "HANDS
PRINT :PROVERB
```

produces the result:

```
MANY HANDS
```

Or, using lists as its inputs:

```
MAKE "PROVERB SENTENCE [MANY HANDS]
                       [MAKE LIGHT WORK]
PRINT :PROVERB
```

gives:

```
MANY HANDS MAKE LIGHT WORK
```

Or:

```
MAKE "NEWSAYING SENTENCE :PROVERB [OR SPOIL BROTH]
PRINT :NEWSAYING
```

which gives:

```
MANY HANDS MAKE LIGHT WORK OR SPOIL BROTH
```

If SENTENCE has more than two inputs then it and its inputs are enclosed in round brackets:

```
PRINT (SENTENCE [A ROLLING] [STONE GATHERS] [NO MOSS])
```

Or:

```
MAKE "OLDSAYING (SENTENCE [THE SAYING] :PROVERB [IS
                TRUE])
PRINT "OLDSAYING
```

which gives:

```
THE SAYING MANY HANDS MAKE LIGHT WORK IS TRUE
```

The PRINT procedure always requires a single word or list as its input. So you could not use

```
PRINT "MANY "HANDS
```

Instead you would need to make them into a single list by using square brackets:

```
PRINT [MANY HANDS]
```

but you would need the SENTENCE operation for anything more complicated; for instance:

```
PRINT SENTENCE [THE SHOPPING LIST IS] :SHOPPING
```

prints:

```
THE SHOPPING LIST IS SAUSAGES PEAS BUTTER STAMPS
```

We will now use SENTENCE to build up a list. The problem is to construct the list of monthly expenditures which we used as an example above. The procedure needs to print out the name of a month, take in from the keyboard the expenditure and add it to the expenditure list.

We begin by constructing a list of the names of the months, making the expenditure list an empty list, and then repeating twelve times a procedure which will input the value for each month.

```
TO BUILDLIST
  MAKE "YEAR [JANUARY FEBRUARY MARCH APRIL MAY JUNE
              JULY AUGUST SEPTEMBER OCTOBER NOVEM-
              BER DECEMBER]
  MAKE "EXPEND []
  MAKE "COUNTER 1
  REPEAT 12 [INMONTH]
END
```

The second procedure prints the name of the month, gets a value – by using the FIRST READLIST technique which we have seen before – and appends it to the EXPEND list using SENTENCE:

```
TO INMONTH
  MAKE "MONTH ITEM :COUNTER :YEAR
  PRINT :MONTH
  MAKE "EXPEND SENTENCE :EXPEND FIRST READLIST
  MAKE "COUNTER :COUNTER + 1
END
```

Actually there is a more sophisticated way of doing this which does not make use of the SENTENCE operation. Sophisticated methods are not necessarily improvements, but this one does give us one advantage. This method uses the ideas explained at the end of the previous chapter; the words in the YEAR list are considered to be the names of variables which are the monthly expenditure, that is "JANUARY names the expenditure for that month. The INMONTH procedure then becomes:

```
TO INMONTH
    MAKE "MONTH  ITEM :COUNTER :YEAR
    PRINT :MONTH
    MAKE :MONTH  FIRST READLIST
    MAKE "COUNTER  :COUNTER + 1
END
```

The second MAKE instruction uses the THING of MONTH as its first input. The first time round, this will be JANUARY so that name will represent whatever value is typed in. The advantage this gives over the building-up process is that indivdual items can be changed quite easily. (It is always good practice in writing programs to allow for the possibility that a user will make a mistake.) If we tackle the problem in this way, there is no need for the list EXPEND.

Exercise:
Write a CHANGE procedure which will get the name of a month and new expenditure from the keyboard and change the value in the list.

If you want to convert this to a collection of procedures for handling your accounts, say, you will need some way of saving information on cassette. LOGO provides only one way of doing this. The command SAVEALL followed by a file-name; for instance:

```
SAVEALL "MONEY
```

saves everything in the workspace in that file – that is, all variables and procedures. This has the advantage that LOADing the file will give not only the information you want but also the procedures to handle it.

Handling words

Many of the procedures used to handle lists and words in lists can also be used for words and characters within them. For instance, if we give FIRST a word as its input rather than a list:

```
PRINT FIRST "ABC
```

we get the first letter of the word:

```
A
```

In the same way:

```
PRINT COUNT "FRIDAY
```

gives 6.

Of all the procedures we have considered in this chapter, the two which will not apply to words are ITEM and SENTENCE. We will need to deal with the lack of an equivalent procedure to ITEM at a later stage, making use of FIRST, LAST, BUTFIRST, BUTLAST and recursion.

In order to construct words we can use the procedure WORD, which works in much the same way as SENTENCE, taking two or more words as its inputs and combining them into a single word.

```
PRINT WORD "BLACK "POOL
BLACKPOOL
```

Like SENTENCE it needs round brackets if it has more than two inputs.

```
PRINT (WORD "BIRM "ING "HAM)
BIRMINGHAM
```

Here is a simple example, using some of the given procedures, to ask for a word from the user and print out a new word which is the reverse of the one typed in. In order to get a single word we again use the FIRST READLIST technique. Then each character is taken from the end of the input word and, using the WORD operation, put on to the end of the new word. To make this possible we first set up the new word to be empty.

```
TO REVERSE
    PRINT [GIVE ME A WORD]
    MAKE "INWORD FIRST READLIST
    MAKE "NEW WORD  "
    REPEAT COUNT :INWORD [MAKE "NEWWORD
                          WORD :NEWWORD LAST :INWORD
                          MAKE "INWORD BUTLAST :INWORD]
    PRINT :NEWWORD
END
```

Those used to programming in BASIC will be familiar with the idea of procedures which split off left and right portions of words. There are no corresponding primitives in LOGO; instead, we will write a procedure which will take as its inputs a word and a number, split off that number of characters from the front of the word and form two new words, to be called LEFT and RIGHT. It works in a similar way to the REVERSE procedure above, starting by setting RIGHT to the whole word input and LEFT to the empty word. The required number of characters are then transferred one by one from RIGHT to LEFT.

```
TO SPLIT :AWORD :NUM
    MAKE "RIGHT :AWORD
    MAKE "LEFT  "
    REPEAT :NUM [MAKE "LEFT  WORD :LEFT  FIRST :RIGHT
                 MAKE "RIGHT  BUTFIRST :RIGHT]
END
```

Try:

```
SPLIT "DEFINED 4
PRINT :RIGHT
PRINT :LEFT
```

LOGO should reply:

```
NED
DEFI
```

to the two PRINT instructions.

5

Repeating Instructions

Conditions

By *condition*, we mean an operation which gives the result TRUE or FALSE in answer to a question like 'Is one number equal to a second?' In the study of logic and list-handling programming languages, such operations are correctly called *predicates*. This perhaps explains why the primitives which fall into this category all have names ending in P. Try typing:

```
PRINT EQUALP 5 5
```

which should give the result:

```
TRUE
```

Here the condition EQUALP tests if its two inputs are equal. If they are it outputs the result TRUE. Otherwise it outputs FALSE.

```
PRINT EQUALP 5 7
FALSE
```

Mathematical conditions

Like other mathematical operations, these are used as symbols placed *between* their two inputs (this is called *infix* notation). These symbols are > (greater than), < (less than) and = (equal to).

```
PRINT 1 > 2
FALSE
```

```
PRINT 5 < 8
TRUE
MAKE "FACT  5 = 5
PRINT :FACT
TRUE
```

= can be used to compare two objects of any kind, whether numbers, words or lists, and will always give the result TRUE if they are the same. It is equivalent to the operation EQUALP which we used above and which is used *before* its two inputs.

```
MAKE "SHOPPING [SAUSAGES PEAS BUTTER STAMPS]
PRINT :SHOPPING = [SAUSAGES PEAS BUTTER STAMPS]
TRUE
MAKE "GIRL "MARY
PRINT :GIRL = "MARY
TRUE
MAKE "NAME "MARY
PRINT :GIRL = :NAME
TRUE
PRINT (7 – 5) = 2
TRUE
PRINT EQUALP (7–5)  2
TRUE
```

Conditions are very often used in an IF command. IF expects at least two, and sometimes three, inputs. The first is a condition, the second a list of instructions which are to be obeyed if the condition is true; for instance:

```
MAKE "TODAY "FRIDAY
IF :TODAY = "FRIDAY [PRINT "TGIF]
TGIF
```

When the IF command has three inputs, the third is a list of instructions to be obeyed if the condition is *not* true.

```
MAKE "TODAY "MONDAY
IF :TODAY = "FRIDAY [PRINT "TGIF] [PRINT  [NOT FRIDAY YET]]
NOT FRIDAY YET
```

Like REPEAT instructions, IF commands can get rather long. I will use the same technique to make them readable; the line will be split and continued below with extra indentation. Usually it will be possible to put the first list of things to do on one line and the alternative list below it. When typing them in, however, press ENTER only when you have reached the end of the complete instruction.

We might use conditions to make communication with the user a bit more friendly. For example, the procedure KNOWNAME of Chapter 4 would be improved if, when referring to the user's first names, the plural was used only if there was more than one name. To do this, we use a variable called ISARE

which is the list [YOUR FIRST NAME IS] if there is only one and [YOUR FIRST NAMES ARE] if there is more than one. This is assigned using an IF statement and used as part of the print-out.

```
TO KNOWNAME
  MAKE "FULLNAME READLIST
  IF COUNT :FULLNAME < 3 [MAKE "ISARE [YOUR FIRST NAME
          IS]]
          [MAKE "ISARE [YOUR FIRST NAMES ARE]]
  PRINT SENTENCE [YOUR SURNAME IS] LAST :FULLNAME
  PRINT SENTENCE :ISARE BUTLAST :FULLNAME
END
```

We will now look at the various primitive conditions and consider the circumstances in which they can be used.

MEMBERP

This is used to check if its first input – which may be a number, word or list – is a member of the second, which must be a list.

```
TO CHECKSHOP :STUFF
  MAKE "SHOPPING [SAUSAGES PEAS BUTTER STAMPS]
  IF MEMBERP :STUFF :SHOPPING [PRINT [GOT IT]]
                              [PRINT [OH DEAR FORGOT-
                              TEN IT]]
END
CHECKSHOP "PEAS
GOT IT
CHECKSHOP "BACON
OH DEAR FORGOTTEN IT
```

Because it is often important in LOGO to know what kind of object we are dealing with, we have five conditional procedures which check the *type* of an object.

NUMBERP

This procedure gives the result TRUE if its input is a number and false otherwise. You might use this to check if the user's input is a number before trying to handle it as one. It is of little use asking 'HOW OLD ARE YOU?' and then trying to add the input to a total age if the user answers NOT VERY.

```
TO CHECKNUM
  MAKE "ANSWER FIRST READLIST
  IF NUMBERP :ANSWER [PRINT [THANK YOU]]
                     [PRINT [I WANTED A NUMBER]]
END
```

Notice that we must use FIRST READLIST for the input here. If READLIST

were used on its own then the result would be a list. Even if it were a list containing a single number, 5, it would be recognized as a list and not a number.

WORDP

This operation gives the result TRUE if its input is a word. Remember that numbers are considered to be words too, so they will also produce a TRUE result from this procedure.

```
PRINT WORDP 3
TRUE
PRINT WORDP "WISE
TRUE
PRINT WORDP [MEN]
FALSE
```

Again, the last object is a list containing one word, and is not itself a word.

LISTP

This gives the result TRUE if its input is a list. So

```
PRINT LISTP :SHOPPING
```

gives:

```
TRUE
```

We will need to write a number of procedures to handle lists in various ways. It will be a good idea to check first if the input to such a procedure is a list.

EMPTYP

This checks if its input is empty. It will check for the empty word, ", or the empty list, [], and give the result TRUE for each. This check is of fundamental importance when breaking down lists or words or examining them one item or character at a time, and we will need to make frequent use of a statement like:

```
IF EMPTYP :ALIST [STOP]
```

in order to end such a process.

NAMEP

This gives the result TRUE if its input is the name of something else; that is, if it has a THING attached to it.

```
MAKE "MONTH "APRIL
PRINT NAMEP "MONTH
TRUE
PRINT NAMEP "FRED
FALSE
```

We mean, of course, that the procedure gives the output TRUE if the word is the name of a LOGO object, not if it is a name like FRED.

Recursion

We have already used procedures like:

```
TO TRI
   FORWARD 50
   RIGHT 120
   TRI
END
```

What happens when the procedure is run? Stop it using CAPS BREAK/SPACE. The important thing about this procedure is that it repeats, not by using a repeat instruction, but by starting itself again, TRI. So TRI obeys the first two instructions, FORWARD 50, RIGHT 120 and then does TRI, which obeys the instructions FORWARD 50, RIGHT 120 . . . and . . .

Recursive procedures can have inputs like other procedures. In the simplest case the inputs are the same each time the procedure is called. We can change TRI to give any polygon:

```
TO POLY :SIDE :ANGLE
   FORWARD :SIDE
   RIGHT :ANGLE
   POLY :SIDE :ANGLE
END
```

This procedure is different from the one suggested in Chapter 2 using the REPEAT command. It can draw some polygons which that one cannot. Try:

```
POLY 50 144
```

When a procedure refers to itself, this is called *recursion*. It is the main way of getting repetition done in LOGO. Seymour Papert, one of LOGO's originators, asks us to consider the plight of a man who always keeps his promises and can be tricked into saying 'I promise to repeat the last thing I said'. Actually, repetitions by means of recursion will not go on for ever, as LOGO keeps a note of every use of a procedure. This requires memory space; when memory is used up the recursion will stop.

In the TRI and POLY examples the use of recursion was very simple, with the consequence that the procedure did not know when to stop. Recursion is not solely used to repeat things, however. There may be additional gains. Suppose that buying a tube of toothpaste from your local supermarket you find that the packets carry a voucher '20p off your next tube'; when do you buy your next tube? The answer is 'immediately'; after all, it does have 20p off (from the voucher on your first tube) and it has another voucher.

We can use the same sort of principle in LOGO:

```
TO ADDON :NUMBER
PRINT :NUMBER
ADDON :NUMBER + 1
END
ADDON 1
1
2
3
4
etc.
```

Here the first use of ADDON prints 1, its input, and passes 2 to the next use of ADDON; this second use sees its input, NUMBER, as 2, so prints 2 and passes on 3 to the next use of ADDON and so on.

Now let us suppose that you find toothpaste tubes whose packets each carry a voucher for '20p off each of your next two packets'. Eager to save money, you buy another two packets, each of which carries a voucher for two more. . . . Your subsequent saving of cash and gains of toothpaste might lead you to appreciate the power of recursion until, like LOGO, you run out of storage space.

As a further example, suppose that we want to add up all the numbers from 1 to N, with the number N starting at 1 and increasing. We might decide to stop when the total becomes greater than one thousand. Here we introduce the STOP instruction which causes the current procedure to end, just like END itself, although STOP can be put anywhere in the procedure. Remember that it only stops the procedure it is in. If that procedure has been called by another, then that previous procedure will continue.

```
TO ADDUP :NUMBER
    IF :TOTAL > 1000 [STOP]
    MAKE "TOTAL  :TOTAL + :NUMBER
    PRINT :TOTAL
    ADDUP :NUMBER + 1
END
MAKE "TOTAL 0
ADDUP 1
 1
 3
 6
10
15
etc.
```

Notice that we have to make the initial value of TOTAL zero outside the procedure as we do not want it returned to zero every time the procedure is obeyed. ADDUP works in much the same way as ADDON with the exception that its second statement checks the current value of TOTAL and stops the procedure if it is greater than one thousand.

When we are doing things with lists, recursion is often the best method to use. For instance, the line-by-line print-out of a list, which we introduced in Chapter 4, can be done more elegantly by a recursive method. We might decide to handle it this way:

```
TO LBL :ALIST
    PRINT FIRST :ALIST
    LBL BUTFIRST :ALIST
END
```

Quite simply, the procedure prints the item at the start of the list and then does the same thing with the remainder. How is it to know when to stop? When the list is empty.

```
TO LBL :ALIST
    IF EMPTYP :ALIST [STOP]
    PRINT FIRST :ALIST
    LBL BUTFIRST :ALIST
END
```

Try this with a suitable list, then modify the procedure so that it prints a list in reverse.

It is true that some uses of recursion can become quite complex but simple recursion really is simple. Apart from the kind of recursive procedure that goes on for ever (or until space in memory runs out) there are two possibilities. In the examples given so far the procedure begins with a test. In 'structured' programming languages like Pascal (and at least one recent version of BASIC) this is called a WHILE loop. In LOGO it looks like this:

```
TO WHILE input
    IF some condition [STOP]
      rest of the procedure
    WHILE input
END
```

In this case the remaining lines of the procedure are not obeyed when the condition is TRUE. In particular they are not obeyed if the condition is true to start with.

The alternative would be put to the test at the end of the set of instructions to be repeated. This corresponds to what in other languages is an UNTIL loop and has the following structure:

```
TO UNTIL input
    lines of procedure
    IF some condition [UNTIL input]
END
```

Here, the lines of the procedure are always obeyed before the condition is met and so they will be obeyed at least once.

Very often in using recursion to repeat, we will find the need to create a start-up procedure to set up initial values, make the first call to the recursive procedure and handle any final output. In the following example, to find the sum of a series of numbers typed in at the keyboard, DOTOTAL first sets a running total to zero and calls the recursive procedure TOTALUP. When all is finished, DOTOTAL then prints out the result.

```
TO DOTOTAL
   MAKE "TOTAL  0
   TOTALUP
   PRINT SENTENCE [THE TOTAL IS] :TOTAL
END
```

TOTALUP illustrates the use of NUMBERP to test if a number has been typed in. It enables the user to put in a series of numbers and finish them with the word END or some other non-numeric input. The procedure is arranged in a similar way to the UNTIL pattern shown above, the only difference being that an extra instruction (MAKE "TOTAL :TOTAL + :NUM) is inserted in the list of things to do if the condition is TRUE. It would be possible to put this at the start of the procedure but only if NUM was also given the value zero by DOTOTAL which would produce the somewhat artificial situation of having a zero value of NUM added on to the zero value of TOTAL the very first time that the procedure was called from DOTOTAL.

```
TO TOTALUP
   PRINT [TYPE IN A NUMBER]
   MAKE "NUM FIRST READLIST
   IF NUMBERP :NUM [MAKE "TOTAL :TOTAL + NUM
                    TOTALUP]
END
```

When the word END is typed the current call to TOTALUP will end. This will cause the whole process to end since it will return to the instruction after TOTALUP in the previous call, an END, and so on. Eventually the very first call of TOTALUP will finish and return to the following line in DOTOTAL, the instruction to print out the result.

Exercise:
Modify TOTALUP so that it only ends if the word END is typed and ignores all other non-numeric input. Do this test for END separately and use the STOP instruction if it is found; then use the NUMBERP condition to control whether the input is added to the total. You will need to call TOTALUP again regardless of whether the input is a number.

A random sentence generator

We will use the ideas of this chapter to write a procedure which will produce random sentences. You can look upon this as an amusing idea or as the first step towards computer-written poetry.

Firstly we need to look at how you can instruct LOGO to produce random numbers. The operation RANDOM does this and will output numbers chosen at random in much the same way as rolling a pair of dice. (Actually the numbers are calculated in some way but we need not bother about how. You are unlikely to notice any variation from true randomness.) RANDOM 5, for instance will output one of the numbers 0, 1, 2, 3 or 4. That is, it gives one of five numbers which run from 0 to 1 less than the input to the RANDOM procedure. In the same way, RANDOM 6 would give one of the numbers 0, 1, 2, 3, 4 or 5. Try:

```
REPEAT 50 [PRINT RANDOM 6]
```

and see what happens.

If you wanted the random number to be chosen from 1, 2, 3, 4 or 5 then you would have to add 1 to RANDOM 5. Type this in as 1 + RANDOM 5, because RANDOM 5 + 1 does the addition first and would actually give you the same as RANDOM 6, i.e. from 0 to 5.

We will use this procedure to pick out a random item from a list. Firstly we make ITEMNO a randomly chosen number from 1 to the COUNT of the list and then use this to select the item.

```
TO CHOOSE :ALIST
   MAKE "ITEMNO  1 + RANDOM COUNT :ALIST
   MAKE "CHOICE  ITEM :ITEMNO :ALIST
END
```

Try this with, say, a list "PRESENT which is:

```
[DOLL TRAINSET COMPUTER] and
   CHOOSE :PRESENT
```

To see the effect of the RANDOM operation you really need to CHOOSE a number of times.

In order to produce sentences, we will need a pattern to work from; for instance,

THE SMALL DOG QUICKLY BITES THE CARELESS MAN

For the benefit of those who have not studied English grammar, we will explain some technical terms as we look at the structure of that sentence. It consists of:

- An ARTICLE, 'THE', an alternative would be A.
- An ADJECTIVE or describing word, 'SMALL', like any of [BIG BROWN SPOTTED CARELESS].
- A NOUN, 'DOG', the name of a thing, such as [MAN TREE TABLE KENNEL].
- AN ADVERB, 'QUICKLY' which describes how something is done, such as [VICIOUSLY SLOWLY CAREFULLY HAPPILY].
- A VERB, 'BITES' which says what is done, like [LIKES CHASES

CARRIES HIDES].

- Another article, 'THE'.
- Another adjective, 'CARELESS'.
- Another noun, 'MAN'.

So, using abbreviations, ART, ADJ and ADVB for article, adjective and adverb, the pattern is:

[ART ADJ NOUN ADVB VERB ART ADJ NOUN]

We will make a variable, "PATTERN, equal to this list and then use each of the names ART, ADJ, NOUN etc. to represent a list of the possible words that can be used in the corresponding place in the pattern. So, for instance, "ADJ will be the list [SMALL BIG BROWN SPOTTED CARELESS]. All this is done by the procedure RANDSENTENCE, which will then repeat, however many times you require, the procedure DORANDOM to construct and output each sentence. This second procedure sets up an empty list, "OUTLINE, to hold the finished sentence, makes the first call to the procedure FOLLOW, which will actually follow the pattern to produce the sentence, and then prints the results. Those two procedures look like this:

```
TO RANDSENTENCE
   MAKE "PATTERN [ART ADJ NOUN ADVB VERB ART ADJ
                  NOUN]
   MAKE "ART  [THE A]
   MAKE "ADJ  [SMALL BIG BROWN SPOTTED CARELESS]
   MAKE "NOUN  [MAN TREE TABLE KENNEL DOG]
   MAKE "ADVB  [QUICKLY VICIOUSLY SLOWLY CAREFULLY
                HAPPILY]
   MAKE "VERB  [BITES LIKES CHASES CARRIES HIDES]
   REPEAT 50 [DORANDOM]
END
TO DORANDOM
   MAKE "OUTLINE  []
   FOLLOW :PATTERN
   PRINT :OUTLINE
   PRINT"
END
```

The procedure FOLLOW first checks if its input list, which will be the pattern or what remains of it, is empty. If so it can stop immediately. Otherwise, it takes the first item of the pattern, which will be the name of a list of words, makes a random choice from this list and puts the chosen word on to the end of OUTLINE. It then calls itself to handle the rest of the pattern. The random selection routine is the same as CHOOSE, which we looked at above.

```
TO FOLLOW :APATTERN
   IF EMPTYP :APATTERN [STOP]
   MAKE "THISWORD FIRST :APATTERN
   CHOOSE  THING :THISWORD
   MAKE "OUTLINE  SENTENCE :OUTLINE :CHOICE
   FOLLOW BUTFIRST :APATTERN
END
```


Notice the use of THING. If the first word of the pattern is ART, then THISWORD names ART which names the list [THE A]. This is the list we want to choose from.

Exercise:

Try incorporating some fixed words into the pattern. Do this by using NAMEP to test if :THISWORD is the name of something; if so, it is the name of a list, and a word is to be chosen from the list. If not, the word itself can be used.

6

More Turtle Graphics

We can illustrate the ideas of the two previous chapters by modifying the RESPI procedure of Chapter 3 so that it turns right on some occasions and left on others. We do this by providing an extra input to the procedure which will be a list of numbers, the values of COUNT for which the turtle is to turn left. When COUNT is a member of this list a left turn is to be made, otherwise a right turn. This might be rather a lot to get into the REPEAT list, so this will be turned into a separate procedure:

```
TO RESPI2 :SIZE :ANGLE :MAX :ALIST
   MAKE "COUNT 1
   REPEAT :MAX [SUBSPI]
   RESPI2 :SIZE :ANGLE :MAX :ALIST
END
TO SUBSPI
   FORWARD :SIZE * :COUNT
   IF MEMBERP :COUNT :ALIST [LEFT :ANGLE] [RIGHT :ANGLE]
   MAKE "COUNT :COUNT + 1
END
```

Try this procedure with the following inputs:

```
RESPI2 5 120  6 [1 3]
RESPI2 5  90 11 [3 4 5]
```

Now that we have looked at how to stop recursive procedures, it might be a good idea to think how we can stop procedures like this one which draw *closed* diagrams, i.e. those which come back on themselves. Firstly we want to know if the turtle has got back to its starting point. There is a primitive procedure which gives the turtle's position at any time. POSITION outputs a list of two numbers. The first is the distance horizontally across the screen from the centre, positive if it is to the right and negative to the left. The second is the distance vertically up the screen from the centre, positive if it is above the centre and negative below. If you like mathematical jargon, they are called the *x-coordinate* and *y-coordinate* respectively. If, on the other hand, you have not met this sort of thing before and find it potentially confusing, it may help to remember that everything goes in alphabetical order; the first number is x, Horizontally, Across, and the second y, Vertically, Up.

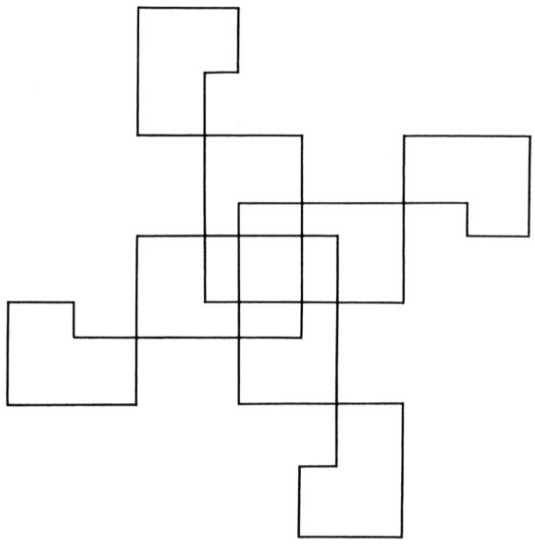

Figure 8 *RESPI2 5 90 7 [1 2 3 4]*

LOGO provides three instructions for moving the turtle to points specified in this way.

```
SETX 50
```

moves to a point which is fifty units across to the right of centre, the distance above or below the centre remaining fixed.

```
SETY -50
```

moves to a point where the distance down from the centre is fifty units but keeps the horizontal distance from the centre fixed. If at any time you want to know either the distance across or up from the centre on its own, you can use XCOR to output the former and YCOR the latter. (These are operations; *output* means provide a result for another procedure to use, not 'display on the screen'.)

```
SETPOS [-20 30]
```

changes both at once, putting the turtle in this case twenty units to the left of centre and thirty units up from it. For each of these instructions, if the pen is down, a line will be drawn from the starting point to the finishing point. Since, as was pointed out in Chapter 3, the minus sign has two meanings, LOGO has to adopt a simple rule for minus signs inside a list to decide which meaning should be used. If there is a blank space or nothing (in the list) in front of it and a number immediately afterwards, it is considered to be part of that number and means that the number is negative. In all other cases it counts as a separate word. That is, [- 10 10] has three words; so have [10 - 10] and [10-10] and none of these could be used as inputs to SETPOS. On the other hand [10 -10] and [-10 10] each have two words and can be inputs to SETPOS. The same care will need to be taken for other procedures which expect a list of numbers as input.

The other thing we would need to know if we wanted to check for a closed diagram is the direction in which the turtle is pointing, its *heading*. The operation to output this is, naturally enough, HEADING. It gives a number between 0 and 360 which works like a compass bearing. Imagining straight up as being North, this is a heading of 0, East is 90, South is 180 and West 270. So as you turn the turtle towards the right you increase its heading; left decreases it. Just as you can set the turtle's position, so you can set its heading. Use the command SETHEADING, followed by a number between 0 and 360, to do this.

Now to solve the problem of drawings which go over themselves. We need to record the turtle's position and heading when it starts. For convenience, we can put the three numbers into one list with heading last:

```
MAKE "STATE LPUT HEADING POSITION
```

We can then keep testing to see if the turtle ever gets back to this state, when it will be in the same place and going in the same direction as it was initially.

```
IF EQUALP :STATE LPUT HEADING POSITION [STOP]
```

Of course, you must be careful not to do this test before the turtle gets a chance to move, or the present position and heading will still be the same as at the start and it will be stopped before it gets anywhere.

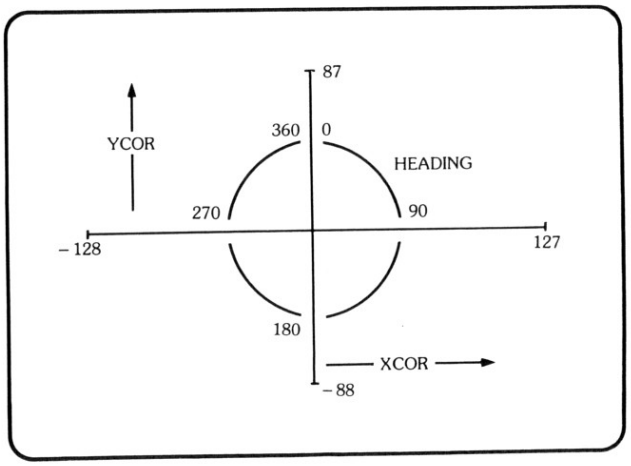

Figure 9 *Turtle coordinates and headings*

Exercise:

Try incorporating this test into some of the procedures of Chapter 3. You will find that you need to create special 'starting-up' procedures to hold the initial MAKE instruction.

We can use the procedures for setting x and y to draw the more normal kind of graphs. Suppose that we have a list of twelve monthly expenditures (or pocket money or team scores, say) and that this is to be passed as an input to a procedure which will draw a graph. For the time being we will assume that the numbers have maximum value of 160 and are not too small compared with this. Firstly, we need to draw a pair of axes. Starting from the bottom left-hand corner, at position [-125 -80], draw a horizontal line with twelve vertical marks for the months.

```
TO XAXIS
    PENUP
    SETPOS [-125 -80]
```

```
        PENDOWN
        SETHEADING 90
        FORWARD 5
        REPEAT 12 [FORWARD 20 SETY –82 SETY –80]
    END
```

The vertical strokes occur every twenty units, so twenty turtle steps across will correspond to one month in the list. In a similar way, a vertical line is drawn for the other axis. It should also be marked with a scale, but since that depends on the size of the quantities in the list, I will leave this to you to decide on and include.

```
    TO YAXIS
        PENUP
        STEPOS [–120 –85]
        PENDOWN
        SETY 80
    END
```

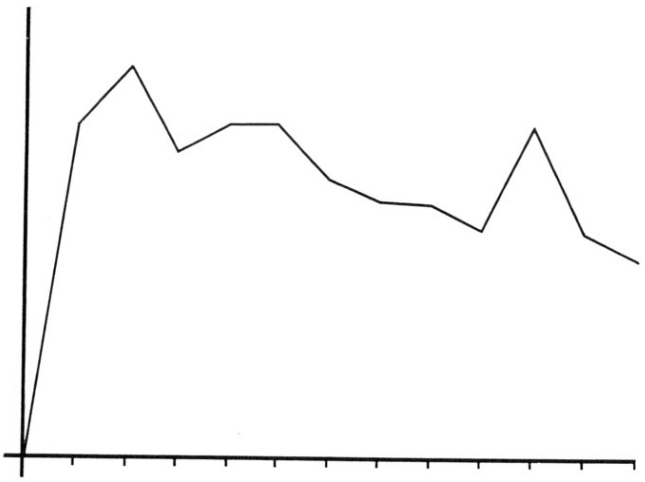

Figure 10 *Design for the procedure GRAPH*

The axes actually cross at the point [–120, 80] so this will be our starting point for the graph. Now we move through the list, using a variable, COUNT, to pick out our position in it, and for each item drawing a line to the point which represents it. This point is twenty units across the screen for each one of COUNT and its distance above the base line is the value of the item. Because the starting position of our graph is 120 units to the left and 80 units down from the centre of the screen we must subtract 120 and 80 respectively from the horizontal and vertical values when they are calculated.

```
    TO PLOT
        MAKE "COUNT :COUNT + 1
        MAKE "XCOR :COUNT * 20–120
        MAKE "YCOR ITEM :COUNT :VALUELIST–80
        SETPOS LIST :XCOR :YCOR
    END
```

COUNT will need to be started off at zero, so the first time this procedure is done it will be brought up to one, and VALUELIST must be provided as the input list of values to be plotted. We can now incorporate all this into the main procedure:

```
    TO GRAPH :VALUELIST
        XAXIS
        YAXIS
        SETPOS [–120 –80]
        PENDOWN
        MAKE "COUNT 0
        REPEAT 12 [PLOT]
        PENUP
        SETPOS [–120 –80]
    END
```

If the values in the list are likely to be greater than 160 or are very much less than this, so that either the turtle goes off the screen or the graph is too small to be seen, the values will have to be *scaled*. That is, they should be multiplied or divided by a suitable number to make them a reasonable size to graph.

Exercise:
Put an instruction to do this into PLOT.

So far we have used recursion with graphics, only as a convenient way of repeating things. As a change, here are some interesting shapes which need recursion.

Firstly the *snowflake curve* (mathematicians seem to call all drawings 'curves'). Draw an equilateral triangle. Now, in the middle of each side, draw another triangle one third the size (point outwards) and on each of the four lines so formed put a triangle one third their size and so on. . . . Now, to draw a triangle, we simply repeat FORWARD :LENGTH and RIGHT 120 three

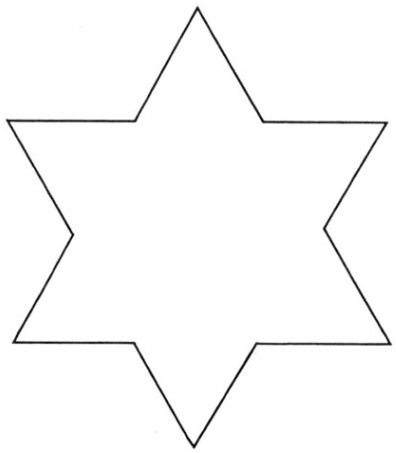

Figure 11 *The first stage of the Snowflake curve showing how 'snowlines' are constructed*

times. To draw the snowflake, we need to replace FORWARD by a procedure to draw a 'snowline' which has a triangle one third of its size at its middle.

```
TO SNOWFLAKE :LENGTH
    REPEAT 3 [SNOWLINE :LENGTH  RIGHT 120]
END
```

To draw the 'snowline', go along it for one third of its length, turn outwards (to the left if you are going right around the triangle) by 60°, draw a line of one third the length, turn back (right) by 120°, draw a line of one third the length, and finally turn left through another 60° to get back on to the right path and complete the line with a further one third:

```
TO SNOWLINE :LENGTH
    FORWARD :LENGTH / 3
    LEFT 60
    FORWARD :LENGTH / 3
    RIGHT 120
    FORWARD :LENGTH / 3
    LEFT 60
    FORWARD :LENGTH / 3
END
```


That gives the effect we want, but wait! Each of the four lines produced has to be broken up with a triangle one third its size; it should be a snowline, in fact. So each FORWARD in the procedure must be replaced with a SNOWLINE command. Eventually the steps will get too small to see. We can use a limit, and when the length of a line is less than this amount, simply go forward rather than try to introduce another even smaller triangle. If we make this limit an input to the original SNOWFLAKE procedure you will be able to start with it fairly large and, by gradually reducing it and re-drawing the diagram, see how the shape is built up.

```
TO SNOWFLAKE :LENGTH :LIMIT
    REPEAT 3 SNOWLINE :LENGTH  RIGHT 120
END
TO SNOWLINE :LENGTH
    IF :LENGTH < :LIMIT [FORWARD :LENGTH  STOP]
    SNOWLINE :LENGTH / 3
    LEFT 60
    SNOWLINE :LENGTH / 3
    RIGHT 120
    SNOWLINE :LENGTH / 3
    LEFT 60
    SNOWLINE :LENGTH / 3
END
```

A suitable demonstration of this curve is:

```
PENUP
SETY -30
PENDOWN
SNOWFLAKE 100 5
```

although you might like to start with higher values of the limit to see how it is built up.

Finally, a curve called *the dragon*. It consists of two procedures which call each other alternately. In between the two calls one of them does a right turn, the other a left turn. The distance to move remains the same throughout the recursion, so we need another method of indicating when the process is to stop. Instead, we can say that when the procedures have got down to a certain level of recursion that will be enough. To do this we introduce a counter, which we will call LEVL, that will actually count in the opposite direction to the levels of procedure calls, starting off at five, for instance, and gradually reducing until zero is reached. The two procedures are, then:

```
TO LEFTDRAGON :SIZE :LEVL
    IF :LEVL = 0 [FORWARD :SIZE  STOP]
```

```
    LEFTDRAGON :SIZE :LEVL – 1
    LEFT 90
    RIGHTDRAGON :SIZE :LEVL – 1
END
TO RIGHTDRAGON :SIZE :LEVL
    IF :LEVL = [FORWARD :SIZE  STOP]
    LEFTDRAGON :SIZE :LEVL – 1
    RIGHT 90
    RIGHTDRAGON :SIZE :LEVL – 1
END
```

It does not matter if you begin with the left or right version. It is recommended that you keep the values of both SIZE and LEVL small to start with. If you increase LEVL you will get a more complex diagram. The following sequence of instructions show the curve well, but may take some time to draw,

```
CLEARSCREEN
PENUP
SETH 90
SETX 100
PENDOWN
LEFTDRAGON 2 20
```

Suggestions for further investigation

You might like to investigate what happens if the triangles on the snowflake curve point inwards rather than outwards. Is it possible to do the same thing with other shapes, say a square, which the snowflake does with a triangle? Is it possible to write a procedure to 'snowflake' any polygon? There are a number of ways you can modify the dragon procedures too, but I will leave you to decide them for yourself.

7

More List Processing

We have already seen some procedures which output results, i.e. operations. You can create your own operations in the usual way using the command OUTPUT to specify the value that is to be returned as the result. For instance, if we want to supply a named procedure that will fill a gap in LOGO – that is, one to give the difference between its two inputs, corresponding to '–' in the same way that SUM corresponds to '+' – then we could use:

```
TO DIFFERENCE :NUM1 :NUM2
    OUTPUT :NUM1 – :NUM2
END
```

Then

```
PRINT DIFFERENCE 7 5
```

will give the result:

```
2
```

When LOGO meets the OUTPUT command, it passes on the value which is found on the right of the word OUTPUT and stops the procedure. So DIFFERENCE 7 5 gives the value 7 – 5. You can have more than one OUTPUT in a procedure if you wish, but their use needs to be controlled by IF commands because, since the procedure ends when OUTPUT is met, only one of them will ever be obeyed; you can, after all, have only one output from any operation. For instance, you might want to adapt the above procedure so that it always gives a positive number as a result, regardless of

which of its inputs is larger. To do this it must compare the two and always subtract the smaller from the larger.

```
TO DIFFERENCE :NUM1 :NUM2
   IF :NUM1 > :NUM2 [OUTPUT :NUM1 – NUM2]
   OUTPUT :NUM2 – NUM1
END
PRINT DIFFERENCE 10 3
7
PRINT DIFFERENCE 2 6
4
```

We could have written:

```
IF :NUM1 > :NUM2 [OUTPUT :NUM1 – :NUM2] [OUTPUT :NUM2 –
                                        :NUM1]
```

but this is not necessary. If the first OUTPUT command is obeyed, then the procedure will come to an end and the second will never be encountered.

It is also possible to construct an operation to output a word or a list as its result in exactly the same way. For instance, we may want to find if a word begins with a vowel or not so that we know whether to use A or AN in front of it. Since this means looking at the first character in the word, it is a good idea to check if the work is empty first. Here is an operation to output AN if the word begins with a vowel, A if it does not and the empty word if the input itself is empty.

```
TO ARTICLE :AWORD
   IF EMPTYP :AWORD [OUTPUT"]
   IF MEMBERP FIRST :AWORD [A E I O U] [OUTPUT "AN]
   OUTPUT  "A
END
```

Exercise:
Change the procedure ARTICLE so that it deals with words written with either capital or lower-case letters.

In some situations you will need to be careful about how you use operations. Firstly, some operations may have side effects; that is, they may also behave like commands. For instance, we might write an operation to ask the user for his or her name and give the first name as its output.

```
TO GETNAME
   PRINT [WHAT IS YOUR NAME]
   OUTPUT FIRST READLIST
END
```

So:

```
MAKE "YOU GETNAME
```

will give the message:

```
WHAT IS YOUR NAME
```


and wait for an answer from the keyboard before passing its first word on as output from the operation. This seems simple enough, but what happens if you try the following?

```
PRINT GETNAME
```

Remember that there is a PRINT instruction in the procedure too. LOGO works out the results of procedures from the right, and so first of all obeys GETNAME and passes its output on to PRINT. In the instruction

```
PRINT SENTENCE "HELLO GETNAME
```

the same thing happens, GETNAME and "HELLO provide inputs to SENTENCE, which then passes its output on to PRINT. Try:

```
PRINT (SENTENCE GETNAME "ALIAS GETNAME)
```

Does it give the result you would expect?

A second thing to remember is that every time an operation is used the result will be obtained afresh. This causes no problem for

```
PRINT DIFFERENCE 7 5
2
PRINT DIFFERENCE 7 5
2
```

because we expect to get the same result every time. (It takes time to perform the calculation repeatedly, though.) We have already seen that the RANDOM operation gives a different result each time it is used (that, after all, is the whole point of using it), and the operation GETNAME, if used repeatedly, will always print the message and wait for input from the keyboard. If you wanted the first name to be preserved you would have to name it and refer to it by name in all subsequent instructions:

```
MAKE "YOU GETNAME
PRINT SENTENCE "HELLO :YOU
PRINT SENTENCE [I AM GLAD TO MEET YOU] :YOU
```

Levels of procedure

When a procedure refers to another one, we say that we have gone to a second *level* of procedure. If this then uses another, we are on a third level and so on. For instance, here are three procedures which do not do very much:

```
TO PROC1
   PRINT [LEVEL 1]
   PROC2
END
TO PROC2
   PRINT [LEVEL 2]
   PROC3
END
```

```
TO PROC3
    PRINT [LEVEL 3]
END
```

PROC1 is the first level. When it calls PROC2, we move to the second level. When PROC3 is called we are at level 3 and then have 3 END, STOP or OUTPUT commands to obey before we can get back to level zero, the top level, which is the LOGO command level with the prompt '?'. There is a command TOPLEVEL which will do this all in one go from any level of procedure. It is not recommended that you make much use of this instruction except in emergencies. The level does not depend on what procedure is used, but on how many procedure calls you have come down through to get there. Imagine each procedure name, including the built-in primitives, as being a step down, and each END, STOP or OUTPUT command as being a step up. So if from LOGO command level you use

```
PROC3
```

you will go down to level 1, even though the procedure is constructed to print out LEVEL 3. A recursive procedure can go through a great many levels.

```
TO DOLEVEL :N
    PRINT SENTENCE "LEVEL :N
    IF :N < 10 [DOLEVEL :N + 1]
END
```

The command:

```
DOLEVEL 1
```

then gives the output:

```
LEVEL 1
LEVEL 2
LEVEL 3
LEVEL 4
LEVEL 5
LEVEL 6
LEVEL 7
LEVEL 8
LEVEL 9
```

When :N finally becomes equal to 10, we have to obey the end statement at level 9, then at level 8 and so on to get back to the top level. You can confirm this by changing DOLEVEL to:

```
TO DOLEVEL :N
    PRINT SENTENCE "LEVEL :N
    IF :N < 10 [DOLEVEL :N + 1]
    PRINT "ENDING
END
DOLEVEL 1
LEVEL 1
```



```
LEVEL 2
LEVEL 3
LEVEL 4
LEVEL 5
LEVEL 6
LEVEL 7
LEVEL 8
LEVEL 9
ENDING
ENDING
ENDING
ENDING
ENDING
ENDING
ENDING
ENDING
ENDING
```

Local and global variables

If a word is used as an input to a procedure when the procedure is defined, then when that procedure is used, the word must take whatever value is given it as input. As a simple example:

```
TO SAYNUM :N
    PRINT SENTENCE [N IS] :N
END
```

then

```
SAYNUM 5
```

gives the not unexpected result:

```
N IS 5
```

But try this:

```
MAKE "N 7
SAYNUM 5
N IS 5
```

then enter:

```
PRINT :N
```

This time the result is perhaps surprising, 7.

LOGO has remembered what value N had outside the procedure SAYNUM and has restored it once the procedure is finished! We say that N is *local* to the procedure; that is, SAYNUM has its own private value of N restricted to its own use. Words which do not occur in the definition of a procedure are called *global* (i.e. their values are the same everywhere). This has an important consequence if we use several different levels of procedure. We will illustrate it by changing the three procedures, PROC1, PROC2,

```
PROC3:
    TO PROC1 :N
       PRINT SENTENCE "PROC1 :NUM
       PROC2 5
       PRINT SENTENCE [PROC1 AGAIN] :NUM
    END
    TO PROC2 :NUM
       PRINT SENTENCE "PROC2 :NUM
       PROC3 7
       PRINT SENTENCE [PROC2 AGAIN] :NUM
    END
    TO PROC3 :NUM
       PRINT SENTENCE "PROC3 :NUM
    END
```

Here procedures 1 and 2 print out the value of their input before and after calling another procedure, so

```
    PROC1 3
```

produces the display:

```
    PROC1 2
    PROC2 5
    PROC3 7
    PROC2 AGAIN 5
    PROC1 AGAIN 3
```

So LOGO remembers which value of NUM belongs to which level of procedure. It is probably easiest to think of local variables as being *different* variables inside the procedure even if they have the same name as any which occur outside it. (In fact, LOGO just makes a temporary change to the value and remembers the old value. This is one reason why memory is taken up every time you call a procedure.)

If a local variable is changed inside a procedure it has no effect outside it. If a global variable is changed, the new value is kept outside too, e.g.

```
    TO TESTIT :NUM
       MAKE "NUM  :NUM + 1
       PRINT :NUM
       MAKE "OTHER  :OTHER + 1
    END
    MAKE "NUM 5
    MAKE "OTHER 7
    TESTIT 5
    6
    PRINT :NUM
    5
    PRINT :OTHER
    8
```


This has an important consequence in recursion. You can rely on LOGO when it finishes a procedure to remember exactly where it was and what the values of any local variables were.

Outputs from recursive procedures

We can use an operation recursively by using the procedure name in its own OUTPUT command; but if you want the procedure to end and actually give a result, you will have to have at least one other OUTPUT which does not use the procedure recursively. To make this clear, we will change the procedure GETNAME used earlier in the chapter so that it requests the name of the user, outputs the first name, if possible, but otherwise issues a suitable message and calls itself recursively to try again. Remember that this will result in the request message being printed out once more too. We do not use FIRST READLIST because, not unreasonably, FIRST does not like the empty list as an input, so FIRST can only be used after we have checked for this.

```
    TO GETNAME
       PRINT [PLEASE TELL ME YOUR NAME]
       MAKE "YOU READLIST
       IF EMPTYP :YOU [PRINT [YOU MUST HAVE ONE]
                       OUTPUT GETNAME]
       MAKE "YOU FIRST :YOU
       IF NUMBERP :YOU [PRINT [ARE YOU A CONVICT ?]
                        OUTPUT GETNAME]
       OUTPUT :YOU
    END
```

We can illustrate the effects of all this by designing a procedure whose inputs will be a number and a word, and whose output will be a word consisting of that number of characters taken from the start of the input word. So

```
    START 5 "RECURSIVE
```

is to produce

```
    RECUR
```

as its output.

If :N is the number and :INWORD the input word, we want to get the first :N characters. Now, we have a way of stripping off the first character of a word. Then all we need to do is apply START recursively to give the first :N – 1 characters from the remainder of INWORD, i.e.

```
    START :N –1 BUTFIRST :INWORD
```

then put the first character on to the front of this with

```
    WORD FIRST :INWORD START :N – 1 BUTFIRST :INWORD
```

and OUTPUT it.

So far so good. As always, the question is, when does the recursion stop? Since N is gradually being reduced and always says how many characters are

wanted, we stop when :N is zero and output the empty word at that point to provide a result for START to the next level up.

```
TO START :N :INWORD
   IF :N = [0 OUTPUT"]
   OUTPUT WORD FIRST :INWORD START :N – 1 BUTFIRST
                  :INWORD
END
PRINT START 5 "RECURSIVE
RECUR
```

Follow the procedure from level to level using the following table:

Level	:N	:INWORD	FIRST :INWORD	BUTFIRST :INWORD	OUTPUT
1	5	RECURSIVE	R	ECURSIVE	
2	4	ECURSIVE	E	CURSIVE	
3	3	CURSIVE	C	URSIVE	
4	2	URSIVE	U	RSIVE	
5	1	RSIVE	R	SIVE	
6	0				"
5	1	RSIVE	R		R = WORD "R "
4	2	URSIVE	U		UR = WORD "U "R
3	3	CURSIVE	C		CUR = WORD "C "UR
2	4	ECURSIVE	E		ECUR = WORD "E "CUR
1	5	RECURSIVE	R		RECUR = WORD "R "ECUR

Exercise:
The start procedure is incomplete since it lacks checks to see if N is not less than 1 or greater than the length of the input word. Decide on appropriate output in these cases and supply the instructions.

Lists within lists

The items in a LOGO list may themselves be lists. We have already seen this in the random sentence generator where NOUN was an item in the PATTERN list and was itself the name of another list. There is nothing particularly difficult in this idea; if I make a list of all the things I have in my pocket, it might be:

[STRING POUND.NOTE SHOPPING.LIST]

Notice that the items on my shopping list are not items on the list of things in my pocket.

In the same way, the above list is quite acceptable to LOGO and has just three items in it even if one of them, SHOPPING.LIST, is itself the name of the list:

[SAUSAGES PEAS BUTTER STAMPS]

and SAUSAGES is *not* a member of the list of things in my pocket! LOGO allows us to write this list as:

[STRING POUND. [NOTE SAUSAGES PEAS BUTTER STAMPS]

and this list still has three items in it even though the third item is a list with four members.

So how many items are there in the following list?

[[SAUSAGES PEAS BUTTER STAMPS] [JANE DAVID ROSEMARY] [GARDENING IRONING] [MAY8 JUNE20 JULY1 SEPT10 NOV23]]

There are four, all of them lists. It may make it clearer to give them names:

[SHOPPING PEOPLE JOBS DATES]

Ignoring what significance these lists may have to me, suppose you were to compile four lists like these for yourself and a friend. Then we could have a list:

[MINE YOURS FRIENDS]

Mine is a list of lists: [SHOPPING PEOPLE JOBS DATES], and so are YOURS and FRIENDS. We could go on putting lists inside each other for a long time but it is probably best to leave the work to LOGO.

Procedures for constructing lists

We have already made use of SENTENCE to form a list from two or more inputs, which may be lists or words. It constructs a single list by taking off the outer square brackets and rewriting the whole list; that is,

(SENTENCE [THIS IS] [A SENTENCE] [EXAMPLE])

outputs [THIS IS A SENTENCE EXAMPLE].

We shall now look at three other operations which construct lists in different ways.

FPUT (First PUT) has two inputs. It expects the second one to be a list and outputs a new list with its first input as the first element and the second input as the rest of the list. So

FPUT "ONE [TWO THREE FOUR]

outputs [ONE TWO THREE FOUR]. Unlike SENTENCE it does not operate by removing the outer brackets of lists and then putting them all together. If the first input is a list, then it is treated as the single first item of the result. So

FPUT [ONE TWO] [THREE FOUR]

gives the output [[ONE TWO] THREE FOUR]. Imagine FPUT as taking its first input and pushing it into the first position of the second one.

LPUT (Last PUT) works in a similar way but makes its first input become the last item in the output list. Again it expects its second input to be a list.

LPUT [FINAL ITEM] [THIS IS THE]

gives the output [THIS IS THE [FINAL ITEM]]. Notice that it seems to expect its inputs in the wrong order.

The operation LIST takes two or more inputs and makes a list of them regardless of whether they are lists or words.

```
LIST "BANK [SAUSAGES PEAS BUTTER STAMPS]
```

has two inputs and forms a list with two items:

```
[BANK [SAUSAGES PEAS BUTTER STAMPS]]
```

If LIST has more than two inputs then it and its inputs must be enclosed in brackets:

```
(LIST "BANK [SAUSAGES PEAS BUTTER STAMPS] [JANE DAVID
      ROSEMARY]),
```

outputs the list:

```
[BANK [SAUSAGES PEAS BUTTER STAMPS] [JANE DAVID
      ROSEMARY]]
```

You can think of LIST as simply taking its inputs and putting an extra pair of square brackets around all of them. Used with more than two inputs, LIST cannot always be relied on to perform properly if it is used as an input to itself. As we have already noted, it will also cause problems if included within the square brackets of a REPEAT or IF list. Complicated structures may, therefore, need to be built up in stages.

Sorting numbers

We will now look at how lists can be used to represent more complicated data structure and how this can be used for practical purposes. Firstly we consider the problem of sorting into order a series of numbers, typed in at the keyboard. Sorting is an important application of computer power and many methods have been develped to do it. This is just one of them.

We are going to store the numbers in a structure called a *tree*. Figure 12 shows such a tree. If it is not obvious why it has this name, turn the diagram upside down. Each circle holding a number is called a *node*. At each node the tree splits into, at most, two branches. To see how it is arranged, we will consider putting another number, 6 for example, into the tree. We start at the top of the tree and, at each node we come to, compare the number we are adding with the number at the node. If it is less, we go down the left branch, otherwise the right. So imagine the 6 being put in at the top where, being less than 7, it goes down the left-hand branch to the 3. Here it is greater and so goes down the right-hand branch to the 5. Again, being greater than this, it goes right. This time, however, there is no right-hand branch so we make one. Simply draw in another arrow going down to the right and put the 6 at the end of it.

So each node has two branches; the left one holds numbers less than the number at the node, the right, numbers greater than it. A simple decision – bigger than the number at the node or not? – sends us down one branch or another.

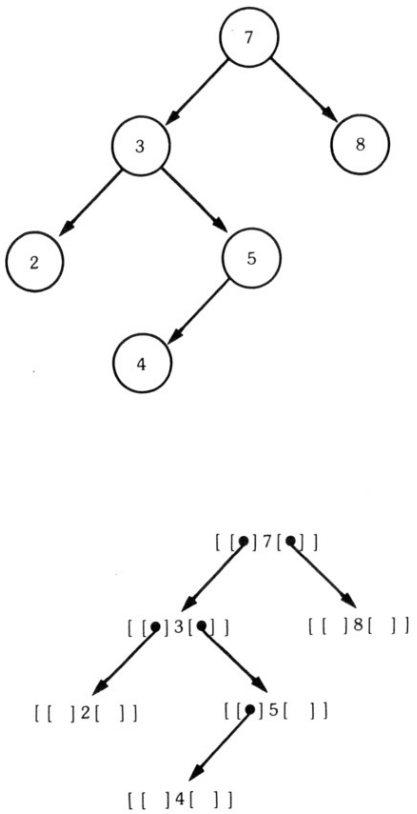

Figure 12 *A number tree and its implementation in lists*

We can build up a tree with lists quite easily in the following way. Each node is represented by a list with three members. The middle member is the number at the node and the other two are lists; the first item is the left-hand branch of the tree from there on and the last item is the right-hand branch. Each of these two is then built up in the same way. If there is no branch then the list is empty. So the empty lists serve as markers to show that we have reached the end, at least as far as that branch is concerned. You may not be surprised to learn that such end points are called *leaves*, while the original node from which the tree grew is called its *root*.

The list structure is also shown in figure 12. Where a list is not empty, it is shown by a spot with an arrow pointing to its contents; so although the lists are actually inside one another, this 'exploded' view shows the similarity to the tree. Look again at how the number 6 would be put into the tree. First look at the main, outermost, list which is at the top of the diagram; it has three members, the middle of which is 7. The new number 6 is less than this so we go down the left-hand branch by considering the first item. This is a list whose middle element is 3; 6 is greater than this so we look at the list following the 3. This has 5 as its middle element; 6 is greater than this so again we look at the list after it. This time the list is empty – we have come to an end point and found somewhere to put the 6. It goes into this empty list and, to mark the new end points, empty lists are put on either side of it: [[] 6[]].

So there are just two rules for putting a number into the tree:

- If a number is put into a list which already has a number in the middle position, it is compared with that number and, if less, is put into the list at the start; otherwise it is put into the list after it.
- If a number is put into an empty list, it becomes the middle element, and two empty lists are put on either side of it.

We begin with a single empty list. If, for example, the numbers are put in in the order 7, 3, 5, 8 ... then the list is built up as in the following table: at each stage, only new things are shown in order to make the structure clear without a clutter of brackets:

Stage

```
1 [                                            ]
2   [                        ] 7 [            ]
3     [ ] 3 [              ]
4              [ ] 5 [ ]
5                                  [ ] 8 [ ]
```

You can see that the numbers are already falling into the right order.

Now we will consider how to do this in LOGO. Firstly, we need a procedure which will give the user some instructions, set up an initially empty list, handle the input of the numbers and print out the list in order:


```
TO SORT
   PRINT [INPUT SOME NUMBERS ONE BY ONE]
   PRINT [WHEN YOU HAVE FINISHED TYPE END]
   MAKE "NUMLIST []
   INPUT
   PRINTINORDER :NUMLIST
END
```

Now the INPUT procedure must get the first word typed in from the keyboard, in the usual way, and check whether it is a number. We can use the occurrence of something which is not a number to signal that we have finished typing numbers in. If it is a number, however, it must be tried in the list to see where it fits, and a new list be formed with the number in its place.

```
TO INPUT
   MAKE "NUM  FIRST READLIST
   IF NOT NUMBERP :NUM [STOP]
   MAKE "NUMLIST  RENEW :NUMLIST
   INPUT
END
```

This makes use of the operation NOT for the first time. This takes a condition as its input and changes the result from TRUE to FALSE and vice versa. So it is used in the obvious way to give the result TRUE if :NUM is NOT a number.

We now need to write the RENEW procedure which will try the input number in any of our lists and put it in its rightful place. This procedure needs to construct lists with three elements, usually as the result of an IF command. As pointed out in an earlier chapter, LIST cannot be used with round brackets inside a list, so we must begin by writing a procedure which will take three inputs and make a list of them.

```
TO LIST3 :A :B :C
   OUTPUT (LIST :A :B :C)
END
```

Now, when the input number is tried with a list (which we will call ALIST inside the RENEW procedure), there are three possibilities. The list may be empty, in which case we replace it by a new list of three items, the outer two being empty lists and the middle one the number:

```
LIST3 [] :NUM []
```

If the number is greater than the middle item, then our new list to replace it must have the first and second items unchanged and the third replaced by a new list containing the incoming number:

```
LIST3 FIRST :ALIST  ITEM 2 :ALIST  RENEW LAST :ALIST
```

If the number is less than the middle item of the list, then it is its first list which must be changed to incorporate it:

```
LIST3 RENEW FIRST :ALIST  ITEM 2 :ALIST  LAST :ALIST
```

So the RENEW operation is:

```
TO RENEW :ALIST
    IF EMPTYP :ALIST [OUTPUT LIST3 [] :NUM []]
    IF :NUM > ITEM 2 :ALIST
              [OUTPUT LIST3 FIRST :ALIST ITEM 2 :ALIST
                            RENEW LAST :ALIST]
              [OUTPUT LIST3 RENEW FIRST :ALIST ITEM 2 :ALIST
                            LAST :ALIST]
END
```

As we have already seen this will result in the numbers being in the right order; the problem is the large number of square brackets that seem to be in the way. After all, our main list is left with just three items in it no matter how many numbers have been typed in. Fortunately the power of recursion helps. To print in order a list consisting of a list, a number and a list, all we need to do is print the first list in order, print the number and then print the second list in order. If the list is empty then we need print nothing at all:

```
TO PRINTINORDER :ALIST
    IF EMPTYP :ALIST [STOP]
    PRINTINORDER FIRST :ALIST
    PRINT ITEM 2 :ALIST
    PRINTINORDER LAST :ALIST
END
```

When you have typed in the procedures, give the command:

```
SORT
```

Type a few numbers, one on each line. End them with END or some other convenient input which is not a number. After the numbers have been printed out in order you might also like to try:

```
PRINT :NUMLIST
```

which will show how the tree is stored in a list. If you want to see the list built up while the SORT procedure is running, put the above instruction into INPUT, just before it calls itself.

A learning game

We will now make use of the tree structure to write a program to play ANIMAL. In this well-known computer game, the program, by asking questions which are answered 'yes' or 'no', endeavours to guess what animal you are thinking about. The program's most interesting feature is its ability to learn. Initially, it 'knows' about only two animals; as you play the game, it learns the names of more and what questions to ask to distinguish between them.

The information is stored in a tree. Each node is a list consisting of three items. Normally the middle one is a question to be put to the user. If the answer is 'yes' then the program uses the first item, also itself a list, as the next

node to consider, if 'no' then the program moves to the last item. Eventually we reach an end point. This is picked out by the fact that the first and last items in it are empty lists. The middle item in this case is not a question but the name of an animal. Imagine working down the number tree of figure 12 in this way. Is it bigger than 7? No. Is it bigger than 3? Yes. Is it bigger than 5? And so on. When the animal is suggested to the person playing the game, there are two possibilities. If the program has guessed right, it obviously has nothing to learn, but if the user was thinking of a different animal the program will require a question to give the difference between the one found and the right answer. This question now becomes the middle item in a new node in this position with the two animal names in lists before and after it. Again, empty lists are put on either side of them to show that end points have been reached.

The basic procedure must provide a suitable introduction to the user and then set up the root list before calling a procedure which will handle the actual working of the game.

```
TO ANIMAL
    PRINT "
    PRINT [WELCOME TO THE ANIMAL GAME]
    PRINT [ANSWER THE QUESTIONS YES OR NO]
    MAKE "NODE [[[][A DUCK][]] [DOES IT SWIM] [[][A PIG][]]]
    GO
END
```

The working procedure GO then invites the user to think of an animal, tries the list in order to guess it and provides for the whole process to be repeated with another animal.

```
TO GO
    PRINT [THINK OF AN ANIMAL]
    PRINT "
    MAKE "NODE TRY :NODE
    PRINT [ANOTHER GO ?]
    MAKE "YN FIRST READLIST
    IF :YN = "YES [GO]
END
```

The procedure GO 'tries' the current node. Eventually this may result in the list being changed, so the TRY procedure is an operation which outputs the new value of the node. There are two possibilities to consider. The middle item of the node may either be a question to ask or an answer to guess. The program distinguishes between them by looking at the first item to see if it is empty, as explained above.

```
TO TRY :ANODE
    IF EMPTYP FIRST :ANODE [OUTPUT GUESSANSWER]
                           [OUTPUT PUTQUESTION]
END
```

Now, the procedure GUESSANSWER must suggest the answer which has been arrived at and, if correct, make no change, i.e. output the same node. If wrong, however, a new node must be constructed.

```
TO GUESSANSWER
    PRINT "
    PRINT SENTENCE [IS IT] ITEM 2 :ANODE
    MAKE "YN FIRST READLIST
    IF :YN = "YES [PRINT [I THOUGHT SO] OUTPUT :ANODE]
                  [OUTPUT NEWNODE]
END
```

The NEWNODE procedure requests the correct answer and constructs a new node for the two animals involved using a question provided by the user. The new animal must be put into a list with empty lists on either side. The old animal is already arranged like this in the current node. The procedure LIST3, explained above, is again needed.

```
TO NEWNODE
    PRINT "
    PRINT [I GIVE UP, WHAT IS IT THEN ?]
    MAKE "ANIMAL1  READLIST
    MAKE "NEWPART (LIST [] :ANIMAL1 [])
    MAKE "ANIMAL2  ITEM 2 :ANODE
    PRINT (SENTENCE [GIVE ME A QUESTION TO GIVE THE
          DIFFERENCE BETWEEN] :ANIMAL1 "AND :ANIMAL2)
    MAKE "QUESTION  READLIST
    PRINT (SENTENCE [AND FOR] :ANIMAL1 [WOULD THE
          ANSWER BE YES ?])
    MAKE "YN FIRST READLIST
    IF :YN = "YES [OUTPUT LIST3 :NEWPART :QUESTION
                  :ANODE]
                  [OUTPUT LIST3 :ANODE :QUESTION
                  :NEWPART]
END
```

This is the most complicated procedure involved in the program, but the key to understanding it lies in following the two alternative OUTPUT instructions which appear in the last line.

There remains the provision of the procedure PUTQUESTION. This must ask the question that forms the central item of the node under consideration and, as the answer is 'yes' or 'no', operate with the procedure TRY on either the first or last item.

```
TO PUTQUESTION
    PRINT "
    PRINT ITEM 2 :ANODE
    MAKE "YN FIRST READLIST
    IF :YN = "YES [OUTPUT LIST3 TRY FIRST :ANODE
                                ITEM 2 :ANODE
                                LAST :ANODE]
                  [OUTPUT LIST3 FIRST :ANODE
                                ITEM 2 :ANODE
                                TRY LAST :ANODE]
END
```

Again, the important parts to understand are the two OUTPUT commands; the three items making up the output list in each case have been written on separate lines to make easy to see which is which.

Now you can type the command:

```
ANIMAL
```

and, if all is well, see how it is possible for a program to 'learn' from its user.

8

Some Mathematical Operations

LOGO recognizes two different kinds of numbers: *integers*, i.e. whole numbers like 1, 245 and –23; and *decimals*, 0.67, 28.91 etc. Usually you will not have to remember that there is a difference. Possibly the only time the distinction will be important is when you want to use a number which may be a decimal as an input to a procedure which expects a whole number. Even then, LOGO will usually take care of you.

The operation INT (INTeger) takes any number as its input and outputs a whole number by removing any decimal parts.

```
PRINT INT 3.14
3
PRINT INT 6.999
6
PRINT INT –2.1
–2
```

Notice that, even though 6.999 is very near to 7, the answer is still 6. This is a process called *truncating*. It always removes the decimals even though the input might be very near to a whole number. In some cases this can be useful. For instance, if you were sharing something out, say dividing a number of objects into groups of 6, you might want to know the number of whole groups and the remainder. The number of whole groups is INT (:NUM / 6). The remainder will be dealt with later. You might also need to test a number to see

if it is a whole number. If it is then INT makes no difference to it.

```
TO TESTINT :NUM
   IF :NUM = INT :NUM [PRINT "YES] [PRINT "NO]
END
```

We can extend this example to ask if one number is divisible by a second (i.e. does the second go into it exactly?).

```
TO TESTDIV :NUM1 :NUM2
   IF (:NUM1 / NUM2) = INT (:NUM1 / :NUM2)
                       [PRINT "YES]
                       [PRINT "NO]
END
```

If, on the other hand, you want the nearest whole number, you can use the operation ROUND. Again, it takes any number as input and outputs a whole number, the nearest one to it. In the case of an input like 2.5 it will take the higher of the two whole numbers on either side of it.

```
PRINT ROUND 3.14
3
PRINT ROUND 6.99
7
PRINT ROUND –2.9
–3
```

This operation, which derives its name from the term *rounding*, used to describe this process, is used if you want to give an answer to a certain degree of accuracy. With a little effort it can be used to give answers to a certain number of decimal places too. For instance, to give 3.14159 to two decimal places, multiply it by 100 (314.159), ROUND it (314) and divide by 100 again (3.14). We can convert this to a recursive procedure by, at each stage, multiplying by 10, finding the rounded number for one fewer decimal places and then multiplying by 10. If the number of decimal places is 0, the number is simply rounded.

```
TO CORRECT :NUM :DECPLS
   IF :DECPLS = 0 [OUTPUT ROUND :NUM]
   OUTPUT (CORRECT :NUM * 10 :DECPLS – 1) / 10
END
```

Another operation which always outputs a whole number is REMAINDER, which has two inputs. It outputs the remainder on dividing the first by the second.

```
PRINT REMAINDER 12 5
2
```

This gives an alternative way to test if one number is divisible by another. If it is, the remainder is 0.

TRUE and FALSE

We can write procedures which test conditions and give the result TRUE or FALSE just like the procedures EQUALP, GREATERP etc. This can be done in one of two ways. You can output the right word, TRUE or FALSE, e.g.

```
TO DIVISIBLE :A :B
   IF 0 = REMAINDER :A :B [OUTPUT "TRUE] [OUTPUT "FALSE]
END
```

or by getting the truth value as a result of a test and outputting the result immediately:

```
TO DIVISIBLE :A :B
   OUTPUT  0 = REMAINDER :A :B
END
```

Here, 0 = REMAINDER :A :B gives the result TRUE or FALSE for DIVISIBLE to output. So it is now possible to use this as an input to IF:

```
IF DIVISIBLE 20 5 [PRINT "YES] [PRINT "NO]
```

You have, in fact, defined your own predicate procedure. You might like to call it DIVISIBLEP to emphasize your mastery of this type of operation.

We might find that something was to be true or false depending on one or more other conditions. LOGO provides three operations to act on predicates. We have already met NOT which changes the output of a condition. AND gives the result TRUE only if all its inputs are true. It normally expects two inputs but can take more if surrounded by round brackets.

```
PRINT AND 1 < 2 3 < 7
TRUE
```

OR, on the other hand, gives the result TRUE if any one of its inputs is true. That includes the possibility that more than one may be true. Like AND it can take more than two inputs in the usual way. We can combine these two together, provided we are careful. For instance, you can come to my birthday party if your name is FRED or if you are female and beautiful. So my testing procedure for issuing invitations takes as input a list of three characteristics:

```
TO CANCOME :ALIST
   MAKE "NAME  FIRST :ALIST
   MAKE "SEX  ITEM 2 :ALIST
   MAKE "FORM  LAST :ALIST
   OUTPUT OR :NAME = "FRED AND :SEX = "FEMALE
                               :FORM = "BEAUTIFUL
END
```

This gives me the results:

```
PRINT CANCOME [FRED MALE UGLY]
TRUE
PRINT CANCOME [JOE MALE UGLY]
FALSE
```



```
PRINT CANCOME [VERA FEMALE UGLY]
FALSE
PRINT CANCOME [VANESSA FEMALE BEAUTIFUL]
TRUE
```

Powers and square roots

We can find the square of a number by multiplying it by itself. Other powers can be found by simple recursion.

```
TO POWER :A :B
   IF :B < 2 [OUTPUT :A]
   OUTPUT (POWER  :A  :B – 1) * A
END
```

LOGO provides a special operation to give the square root of a number, SQRT.

```
PRINT SQRT 25
5
```

We can use the theorem of Pythagoras to give the distance between two points whose positions (x-coordinate and y-coordinate) are known. You will find in mathematical text books that if the positions are [X1 Y1] and [X2 Y2] then the distance between them is the square root of $(X2 - X1)^2 + (Y2 - Y1)^2$:

```
TO DISTANCE :POS1 :POS2
   MAKE "XDIF  FIRST  :POS1 – FIRST :POS2
   MAKE "YDIF  LAST :POS1 – LAST :POS2
   OUTPUT SQRT(XDIF * XDIF + YDIF * YDIF)
END
```

Finding prime numbers

A prime number is one which is divisible by itself and 1 only. Mathematicians have been fascinated by the problem of producing prime numbers for thousands of years. It seems that there is no formula to give them all and many suggested formulae to generate some of them have, after producing a list of prime numbers, started to produce non-primes as well.

Although there may not be a formula for primes, there is a method. We can simply take each number in turn and test it to see whether it is divisible by any of the numbers smaller than itself. This is simplified by the fact that we need only test for divisibility by numbers up to the square root of the one we are testing. If it can be divided by something bigger than its square root then the result of the division would be a number smaller than the square root which we have already tested.

```
TO TESTPRIME :NUM :N
   IF :N > SQRT :NUM [OUTPUT "TRUE]
```

```
    IF DIVISIBLE :NUM :N [OUTPUT "FALSE]
    OUTPUT TESTPRIME :NUM :N + 1
END
```

We need to start testing the number with 2, the smallest prime number.

```
TO PRIMESFROM :NUM
    IF TESTPRIME :NUM 2 [PRINT :NUM]
    PRIMESFROM :NUM + 1
END
```

The command

```
PRIMESFROM 2
```

will then print out the primes from 2 onwards.

Trigonometric operations

You should skip this paragraph if you are already familiar with trigonometric operations, or read on if you need further explanation. If the turtle has a HEADING of :ANGLE then for every turtle step it goes forward it will go different amounts across and up the screen depending on :ANGLE. The distance move horizontally is called the *sine* of the angle and the distance moved upwards its *cosine*. (That is, the x-coordinate changes by the sine and the y-coordinate by the cosine.) If the distance moved is one unit then obviously these two will be less than 1. You might like to think of ten turtle steps being taken forward, in which case the horizontal movement is 10 * SINE :ANGLE and the vertical movement 10 * COSINE :ANGLE. Mathematicians also use two other operations: *tangent*, which is found by dividing the sine by the cosine, and *cotangent*, which is found by dividing the cosine by the sine.

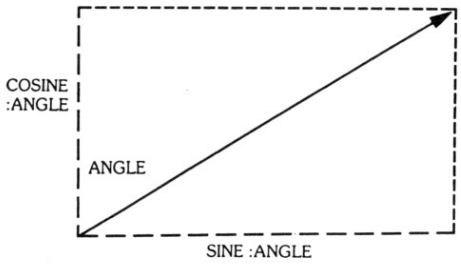

Figure 13 *The SINE and COSINE of an angle*

LOGO provides four trigonometric operations: SINE, COSINE, TANGENT and COTANGENT, with the abbreviations SIN, COS, TAN and COT. There is a big advantage over most other computer languages in that LOGO expects the inputs to these operations to be angles in degrees. So if you have used other languages which expected angles measured in radians you can heave a sigh of relief. If, on the other hand, you have never heard of radians you can heave a sigh of relief too because you will not need to.

In addition to the four operations given above there are another four, ARCSIN, ARCCOS, ARCTAN and ARCCOT, which perform the opposite function, taking a single number as an input and outputting the angle, in degrees, which has that sine, cosine, tangent or cotangent. So

```
PRINT ARCCOS 0.5
```

will give you the angle whose cosine is 0.5, i.e. 60. (A reminder for those new to these operations: this means simply that on a heading of 60 the turtle will travel half as far up the screen as the distance it goes forward.)

These operations are of great importance in the study of electronics, sound waves and all sorts of vibrations. They are also capable of generating interesting and beautiful curved figures. Perhaps the reason will be clear from the following pair of procedures which draw a graph of cosine of angles from 0 to 360. X and Y are calculated from the angle and the cosine respectively, scaling them so they fit on the screen. You will get a 'turtle out of bounds' message when the edge of the screen is reached.

```
TO SHOWCOS
    PENUP
    SETPOS [120 0]
    PENDOWN
    SETPOS [-120 0]
    DOCOS 0
END
TO DOCOS :ANGLE
    MAKE "X 2 * :ANGLE / 3 - 120
    MAKE "Y 80 * COS :ANGLE
    SETPOS SE :X :Y
    DOCOS :ANGLE + 15
END
```

This has an oscillating movement up the screen only. What happens if we allow oscillation across too? The resulting shapes are named, after the man who discovered them, Lissajou's figures:

```
TO LISSAJOU :A :B :C :D :INC
    SHOWTURTLE
    CLEARSCREEN
    PENUP
    LISS 0
```

```
END
TO LISS :ANGLE
    MAKE "X  120 * COS(:A * :ANGLE + :B)
    MAKE "Y  80 * COS(:C * :ANGLE + :D)
    SETPOS SE :X :Y
    PENDOWN
    LISS :ANGLE + :INC
END
```

The pen is put down after the first use of SETPOS (and kept down afterwards, of course) so that the turtle will move to its starting position without drawing a line. The higher the value of INC the more angular the drawing will become due to the turtle 'cutting the corners'. Try:

```
LISSAJOU 11 20 21 40 1
```

Incorporating random changes of pen-colour into the procedure can produce interesting effects.

Here are some more turtle procedures to try:

```
TO CYCLO :K :ANGLE :SIZE
    FORWARD :SIZE * SIN(:K * HEADING)
    RIGHT :ANGLE
    CYCLO :K ANGLE :SIZE
END
```

Try:

```
CYCLO 1 10 10
CYCLO 4 5 20
```

And:

```
TO LOBE :A :B :INC :ANGLE
    FORWARD 1
    RIGHT :A + :B * SIN :ANGLE
    LOBE :A :B :INC :ANGLE + :INC
END
```

Try:

```
LOBE 2 3 9 0
LOBE 10 8 6 0
LOBE 2 8 6 0
LOBE 2 9 3 0
```

Some adjustment of the size of the turtle's starting position may be necessary. Experiment!

9

More Input and Output

So far we have considered only one method of getting information into the computer – READLIST – and, apart from turtle drawings, one method of getting information out – PRINT. In this chapter we will look at methods which give you, the programmer, more control over input and output, though possibly at the expense of having to write more complicated programs.

READCHAR reads a single character from the keyboard, waiting until the user presses a key and then passing the character for that key on as its output. You do not have to press ENTER after the key; the character is not printed on the screen and there is no prompt printed to show that LOGO is ready for input. Try:

```
MAKE "CHARACTER READCHAR
```

After pressing ENTER, press another character; the ? prompt should reappear now as you are back to LOGO command mode. You will have to type in:

```
PRINT :CHARACTER
```

to see what it was that you pressed! This operation can be used whenever you want to control a program with single keys but without anything appearing on the screen. Here, for instance, is a procedure which enables you to steer the turtle around the screen:

```
TO DRAW
    MAKE "CHARACTER READCHAR
    IF :CHARACTER = "F [FORWARD 5]
```

```
    IF :CHARACTER = "B [BACK 5]
    IF :CHARACTER = "R [RIGHT 10]
    IF :CHARACTER = "L [LEFT 10]
    IF :CHARACTER = "S [STOP]
    DRAW
END
```

READCHAR will often be useful if the codes for the various characters are also used. The computer stores and recognizes characters by using the numbers from 1 to 126 according to a system called the American Standard Code for Information Interchange, shortened to ASCII. Codes 1 to 31 are used by the computer for controlling input or output (for instance, the 'cursor movement' keys, CAPS 5 etc.) and cannot be printed on the screen. In addition, for the Spectrum, character 127 is a 'copyright' character consisting of a small 'c' in a circle, and the coding is extended beyond that with codes 128 to 143 representing a blank and the fifteen 'graphics' characters which the Spectrum provides. Codes 144 to 164 are definable characters. See your Spectrum manual or *LOGO 2* for details of the coding.

You can get LOGO to tell you what codes the Spectrum understands by means of the following procedure:

```
TO SHOWASCII
    MAKE "CHARACTER READCHAR
    PRINT ASCII :CHARACTER
    SHOWASCII
END
```

Type SHOWASCII and then press some keys to see their codes displayed. You should find that the digits 0 to 9 give the codes 48 to 57. You can use this to modify the drawing procedure by inserting the following:

```
MAKE "CODE ASCII :CHARACTER
IF AND :CODE > 47 :CODE < [56 SETPC :CODE - 48]
```

which will enable you to change pen-colour by pressing one of the number keys.

The operation CHAR does the opposite of ASCII, turning a CODE number between 32 and 143 into the corresponding character.

```
PRINT CHAR 65
```

should give

```
A
```

We can use this in combination with ASCII to change what is typed in. We will also need a command which works like PRINT but does not take a new line every time it is used. The command is TYPE. Try the following procedure, which replaces the character typed in with the one with the next higher code.

```
TO CONFUSE
    MAKE "CODE  ASCII READCHAR
    IF :CODE = 13 [STOP]
    TYPE CHAR (:CODE + 1)
    CONFUSE
END
```


The procedure ends when ENTER (ASCII code 13) is pressed.

If you do not want to wait for a key to be pressed, you can use the test KEYP, which gives the result TRUE or FALSE accordingly as a key is pressed or not. Here is another drawing program but one in which the turtle keeps moving.

```
TO DRAW2
    IF KEYP [DOIT]
    FORWARD 5
    DRAW2
END
TO DOIT
    MAKE "CHARACTER READCHAR
    IF :CHARACTER = "M [RIGHT 90]
    IF :CHARACTER = "Z [LEFT 90]
END
```

The keys are chosen so that they can be operated by fingers of the hand appropriate to the direction of turn.

The characters with codes between 144 and 164 inclusive are user definable; that is, they can be changed by you to any shape you wish. Full details about how to do this will be found in the Spectrum manual, but it requires that certain locations in the Spectrum memory should have their contents changed. This is done in LOGO using the command .DEPOSIT which is explained in *LOGO 2*. Warning: this command changes memory directly, i.e. without LOGO keeping a check on what it is doing; if used without care, it can destroy procedures that you have stored. The following pair of procedures re-define a character whose ASCII code is between 144 and 164 according to a list of values which will be put into the appropriate positions in memory.

```
TO DEFCHAR :CH :ALIST
    IF OR NOT NUMBERP :CH NOT LISTP :ALIST [STOP]
    IF OR :CH < 144 :CH > 164 [STOP]
    DODEF :ALIST 0
END
TO DODEF :ALIST :COUNT
    IF EMPTYP :ALIST [STOP]
    .DEPOSIT 64216 + :CH * 8 + :COUNT  FIRST :ALIST
    DODEF BUTFIRST :ALIST :COUNT + 1
END
```

You should consult the manual to see how to decide the numbers in the list, and test the procedures on their own first to make sure that everything has

been typed in correctly.

```
DEFCHAR 144 [16 16 16 16 146 84 56 16]
```

is a suitable test and should make CHAR 144 a downward pointing arrow.

Text output is controlled by means of a number of instructions which we will examine one by one.

SETCURSOR takes as its input a list of two numbers and puts the cursor at the position that they indicate. Subsequent printing starts there. The first number in the list indicates how far across the screen, in whole character positions, the printing is to start, with 0 on the extreme left and 30 on the extreme right; the second number gives the number of lines down the screen with 0 at the top and 21 at the bottom.

SETTC (SET Text Colour) also takes a list of two numbers as its input. The first is the new background colour for printing text and the second the colour of the text itself. The usual numbers are used to give the colours. The normal situation is [7 0]. You can test this command with a procedure such as:

```
TO TESTCOL
    TEXTSCREEN
    SETCUR [12 10]
    SETTC [1 6]
    PRINT [HELLO THERE]
    MAKE "G READCHAR
    SETTC [7 0]
END
```

The final lines wait for a key to be pressed before restoring the normal colouring.

The brightness of the 'paper' on which the text is shown can be controlled by the command BRIGHT which has one input. If the input is 1, the background for printing is bright, if 0 then dull. 0 is the usual state. This instruction can only be used inside a procedure but it may help to make things more readable.

INVERSE prints with the text in the background colour and the background in the text colour. FLASH swaps continually between inverse and normal printing and back again. Finally, NORMAL restores the ordinary setting for printing text, cancelling the effects of INVERSE, FLASH and BRIGHT.

Here then is a group of procedures for those who want to see their names in lights:

```
TO FANCYNAME
    PRINT [WHAT IS YOUR NAME?]
    MAKE "NAME READLIST
    TEXTSCREEN
    BRIGHT 1
    SHOWNAME
    SETTC [7 0]
    SETCUR [0 21]
END
TO SHOWNAME
    IF KEYP [STOP]
    MAKE "X  RANDOM 31
    MAKE "Y  RANDOM 21
    MAKE "BG  RANDOM 8
    MAKE "FG  RNDCOL
    SETTC  SENTENCE :BG :FG
    SETCUR  SENTENCE :X :Y
    TYPE :NAME
    SHOWNAME
END
TO RNDCOL
    MAKE "FG RANDOM 8
    IF :FG = :BG [OUTPUT RNDCOL] [OUTPUT :FG]
END
```

The procedure RNDCOL not only generates a random number for the text but also checks to see if it is the same as the background; if it is, it repeats.

Now a group of procedures which demonstrate all these techniques. They handle match results for a small football or other sports league. The main procedure sets up the lists required and prints suitable headings in colour. It then calls a procedure, HEADCOL, to print the headings to the columns of the table, and a second procedure, DOTEAM, to handle each team's results.

```
TO LEAGUE
    MAKE "TEAMS [WANDERERS ROVERS RANGERS ALLSTARS]
    MAKE "COLS [P W D L PTS]
    TEXTSCREEN
    SETCURSOR [9 0]
    SETTC [5 0]
    TYPE [LEAGUE TABLE]
    SETCURSOR [0 2]
    SETTC [7 3]
    TYPE "TEAM
    HEADCOL 1
    SETTC [7 0]
    DOTEAM 1
    PRINT "
END
```

Notice that ALLSTARS has to be one word if we want it to be considered by LOGO as a single item. You could separate the two parts by an underline sign (SYS 0) or a full stop if you wished.

The procedure HEADCOL selects an item from the list of column headings and, by the use of SETCURSOR, prints it in an appropriate place. It then repeats until all the headings have been printed.

```
TO HEADCOL :COL
   IF :COL > 5  STOP
   SETCURSOR SENTENCE [3 * :COL + 11] 2
   TYPE ITEM :COL :COLS
   HEADCOL :COL + 1
END
```

DOTEAM selects a suitable line for printing which depends on the team's number in the list. It puts the team name at the start of the line and then gets the number of matches played, won and drawn as typed in at the keyboard by the user. It assumes the existence of another procedure called GETNUM to do this, which takes as its first two inputs the position at which the number is to be shown on the screen. The third input is initially the empty word as it will be used as the word to hold the number typed in. The number of matches lost and the points total can then be worked out. I have assumed that there will be two points for a win and one for a draw. You can change the line in the procedure to suit your own needs.

```
TO DOTEAM :TEAMNO
   IF :TEAMNO > COUNT :TEAMS [STOP]
   MAKE "LINE  2 * : TEAMNO + 2
   SETCURSOR SENTENCE 0 :LINE
   TYPE ITEM :TEAMNO :TEAMS
   MAKE "PLAYED GETNUM 14 :LINE"
   MAKE "WON GETNUM 17 :LINE"
   MAKE "DRAWN GETNUM 20 :LINE"
   MAKE "LOST :PLAYED – :WON – :DRAWN
   SETCURSOR SENTENCE 23 :LINE
   TYPE :LOST
   MAKE "POINTS  2 * :WON + :DRAWN
   SETCURSOR SENTENCE 26 :LINE
   TYPE :POINTS
   DOTEAM :TEAMNO + 1
END
```

Before looking at the procedure GETNUM we will consider another procedure which it will use. GETDIG gets a digit typed at the keyboard and supplies both the digit and its ASCII code. If ENTER (code 13) or DELETE (code 12) are pressed these codes are also passed on. Otherwise, if the key pressed is not a digit, it is ignored and the procedure called again. (Remember that the digits 0 to 9 have codes 48 to 57.) This is an alternative solution to the problem of what to do if the user does not type a number when one is expected – don't let it happen!


```
TO GETDIG
   MAKE "DIG  READCHAR
   MAKE "CODE  ASCII :DIG
   IF OR :CODE = 13 :CODE = 12 [STOP]
   IF OR :CODE < 48 :CODE > 57 [GETDIG]
END
```

Now the procedure GETNUM: it begins, curiously, by putting the cursor in the right place and printing out its third input. This is because we are going to use the procedure recursively. Eventually this word will hold the number typed in; it is printed out at this stage so that we can see how far we have got in typing it. The instructions

```
FLASH
TYPE "?
NORMAL
```

provide a flashing question mark as a cursor when used within a procedure. Then GETDIG provides the digit or ENTER or DELETE which the user types. In case ENTER or DELETE is pressed we want to get rid of that flashing question mark – otherwise it will stay in that position on the screen while the user is typing somewhere else. We can remove it by putting a blank space (character code 32) in its place. But where exactly is it? The best way is to go back to the start of the number (we know where that is), print the number again and put the blank at its end where the question mark was.

```
SETCURSOR SENTENCE :X :Y
TYPE :NO
TYPE CHAR 32
```

where :X and :Y are the position of the start of the number.

Now there are three possibilities for the result of GETDIG: the key pressed may have been a digit (with a code between 48 and 57), or ENTER or DELETE. If a digit, we just want to put it on to the end of the number we have already got. In this example, I have decided to restrict the number to two digits, so if this is the third digit it is ignored:

```
IF AND :CODE > 13 (COUNT :NO)<2 MAKE "NO WORD :NO :DIG
```

If the ENTER key was pressed and if the number is not empty then we can output the number and press on. If the DELETE key was pressed then we want to remove the last character in the number using BUTLAST. In this case we also want to be sure that the number is not empty before we start. If you look at the procedure below, you will see a rather unusual line below this:

```
IF EMPTYP :NO[MAKE "NO"]
```

which does not seem to be necessary. Unfortunately, when BUTLAST operates on a word with one character in it, it changes it not to the empty word but to an empty list, which WORD – which must be used to put another digit in place – does not like as input. This instruction is to turn the empty list back to a word as it ought to be.

```
TO GETNUM :X :Y :NO
  SETCURSOR SENTENCE :X :Y
  TYPE :NO
  FLASH
  TYPE "?
  NORMAL
  GETDIG
  SETCURSOR SENTENCE :X :Y
  TYPE :NO
  TYPE CHAR 32
  IF AND :CODE = 13 (NOT EMPTYP :NO) [OUTPUT :NO]
  IF AND :CODE = 12 (NOT EMPTYP :NO) [MAKE "NO BUTLAST
                                          :NO]
  IF EMPTYP :NO [MAKE "NO"]
  IF AND :CODE > 13 (COUNT :NO) < 2 [MAKE "NO WORD :NO
                                          :DIG]
  OUTPUT GETNUM :X :Y :NO
END
```

The above has a number of deficiencies; for instance, you can type in more wins than matches played, giving negative losses, and it does not pick out an overall winner, I hope, however, it shows how any sort of table can be built up.

Exercise:
Change DOTEAM to keep a record of the best team so far. Start with a variable, BESTSCORE, as zero; each time a team is found whose score is better change BESTSCORE, appropriately and store the team's name as BESTTEAM. Then arrange for a suitable announcement to be printed out at the end.

Sound

The Spectrum can also output musical notes. This is controlled by the instruction SOUND whose input is a list of two numbers. The first, the duration of the sound in seconds, must be between 0 and 10.5; the second, its pitch in semitones above middle C (the central note on a piano keyboard) must be between –60 and +69. The following procedure turns your keyboard into an electronic organ, though not one that is easy to play.

```
TO KEYBOARD
  MAKE "KEY (ASCII READCHAR) –65
  IF AND :KEY > –59 :KEY < 70 [SOUND SENTENCE 0.5 :KEY]
  KEYBOARD
END
```

You can play a list of notes, all of the same duration, using this procedure:


```
TO PLAY :ALIST
  IF EMPTYP :ALIST [STOP]
  SOUND SENTENCE 1 FIRST :ALIST
  PLAY BUTFIRST :ALIST
END
```

A semitone is the difference in sound between adjacent frets on a guitar or similar instrument, or between adjacent keys (black or white) on a piano. If you do not know how semitones relate to a musical scale, try the following, which plays the scale of C:

```
PLAY [0 2 4 5 7 9 11 12]
```

Unfortunately, the input expected by SOUND is not very like ordinary musical notation. It is possible, however, to write a procedure to do the conversion for us. The following procedure expects a 'rate' to be given first which will control the speed at which the music is played. It will be the length of time, in seconds, for a note whose length is entered as 1, so the lower this number, the faster the tune will be played. The procedure then gets a list of notes, putting them into a list previously set up, called SLIST. Finally the music is played.

```
TO MUSIC
  MAKE "NOTELIST [C C# D D# E F F# G G# A A# B C']
  MAKE "SLIST []
  PRINT [WHAT RATE?]
  MAKE "RATE  FIRST READLIST
  PRINT [TYPE IN NOTES ONE PER LINE, DURATION FIRST]
  GETMUS
  PLAY :SLIST
END
```

The procedure GETMUS inputs a list of two items from the keyboard. The first is the length of the note. I suggest that you call the shortest note in the piece 1, and give all the others as multiples of it. How fast they are actually played will be controlled by RATE, and the duration is calculated, in GETMUS, by multiplying the length of the note typed in by RATE. The pitch can be typed in as one of the notes given in NOTELIST above. Use SYS 3 to get the sharp sign and SYS 7 to get the dash which indicates the C above middle C. You could change the note names to Do, Re, Mi, etc. if you prefer but you would need some way of writing the 'intermediate' notes. GETMUS requires an additional procedure, FIND, which finds an object in a list. The first two inputs to FIND are the names of the object and list, the third is the number from which it is to start counting. In this case we want this to be 0 because we want our first note to be middle C.

```
TO GETMUS
  MAKE "NOTE READLIST
  IF EMPTYP :NOTE [STOP]
```

```
    MAKE "DUR :RATE * FIRST :NOTE
    MAKE "PITCH  FIND LAST :NOTE :NOTELIST 0
    MAKE "SLIST  LPUT LIST :DUR :PITCH :SLIST
    GETMUS
END
```

The last-but-one instruction makes duration and pitch into a list of two items ready for SOUND to use and puts this on to the end of SLIST. The procedure is constructed to stop when an empty list is typed in.

```
TO FIND :OBJECT :ALIST :N
    IF EMPTYP :ALIST [OUTPUT 0]
    IF :OBJECT = FIRST :ALIST [OUTPUT :N]
    OUTPUT FIND :OBJECT BUTFIRST :ALIST :N + 1
END
```

Finally, to play the music, the procedure PLAY. This differs from the procedure called PLAY given earlier, but only in that it expects each item in its input list to be a list of two numbers which it can pass on to SOUND immediately.

```
TO PLAY :ALIST
    IF EMPTYP :ALIST [STOP]
    SOUND FIRST :ALIST
    PLAY BUTFIRST :ALIST
END
```

After typing in the procedures, try the following. Your input is introduced by a question mark at the start of the line.

```
WHAT RATE?
?0.3
TYPE IN NOTES ONE PER LINE, DURATION FIRST
?2 G
?4 C'
?2 B
?2 A
?4 G
?2 A
?1 B
?1 C'
?2 E
?2 E
?2 F
?2 D
?6 C
```

Press ENTER when the next prompt appears. There will be a short delay before the music is played. Once entered you can play it again by the instruction:


```
PLAY :SLIST
```

Exercise:

You can have more than one tune in memory at a time by giving each a name. Adapt MUSIC so that it starts by asking for a name. MAKE "SLIST this name and change all the subsequent uses of SLIST so that its THING is used instead of its name. You could then replay a tune called JINGLE by means of PLAY :JINGLE.

10

When Things Go Wrong

Firstly some words intended to be comforting to all those who have just found an error in procedure: the most important thing to learn about computer programming is that going wrong seems to be the natural condition of programs. It has been said that the inexperienced programmer is surprised when his or her programs do not work first time; but the experienced programmer is surprised when they do. Programs can go wrong in two ways. You may make a mistake in writing LOGO instructions, which will result in an error message being displayed; or you may write a procedure which LOGO understands and can obey but which does not do what you intended it to. The methods of discovering and eliminating the 'bugs', as program errors are usually called, are very much the same in each case. However, the former are usually easier to handle, since you will be given an error message – in LOGO usually more helpful than those of many other computer languages – which will tell you exactly what LOGO found difficult and what procedure it was obeying at the time. Despite the helpful nature of LOGO messages in general, remember that they have been written by a programmer who has had to guess, in advance what you were likely to have done to cause a problem. For this reason they are not always completely accurate.

A look at some of the more usual error messages and their likely causes might help.

You don't say what to do with blah

A message which is easy to explain. Remember that an operation outputs a result and must always have the name of a command, which will accept that result as input, on its left. In my example, an operation has produced the word 'blah' as output and either there is no command to handle it or the command which is there does not expect that input. As an example of the latter case, suppose that the word is produced by an operation, SAYBLAH; then

PRINT "RESULT SAYBLAH

will produce the above error message as PRINT expects one input, "RESULT, and there is no indication of what to do with the output of SAYBLAH. Obviously, SENTENCE should be used in this case.

FIRSTNAME does not output to SECONDNAME

This happens when you have an instruction, like

SECONDNAME FIRSTNAME

and SECONDNAME expects an input when FIRSTNAME is a command not an operation. It frequently occurs if you type:

PO FIRSTNAME

forgetting that there should be quotes before the name of the procedure to be printed.

Not enough inputs to BLAH

If you examine your definition of 'BLAH', or that given in *LOGO 2* if it is a primitive, you will find that it expects more inputs than it has been given. This may not be obvious if the line is complicated but, for this message and the two previous ones, the method of analysing LOGO instructions given at the end of Chapter 3 will usually find the error.

Too many inside parentheses

will occur if you have too many round brackets inside a list. For practical purposes 'too many' seems to mean any at all. See my comments about the use of brackets around primitives in Chapter 3.

FIRSTNAME doesn't like SECONDNAME as input

(Sometimes there may just be a blank space between 'like' and 'input'; in this case the second thing is either a blank or the empty word.) There are two possibilities if you get this message. Either the input may be of the wrong type – it may, for instance, be a number where a list is expected:

SETCUR 10 15

or it may not be in the right range of values:

SETCUR [10 45]

You should check the manual carefully to decide what the procedure expects as input.

Not enough space to proceed

refers to space in memory, not space on the screen. It may be possible to do something about this using the command RECYCLE. This performs what in computer jargon is called a *garbage collection*; it finds all the memory

locations which have been set aside for something and which are no longer needed for their original purpose, thus freeing extra memory. This is unlikely to be of much help, however, as garbage collection does take place automatically from time to time. A more useful solution would be to remove all the procedures that you do not need. You may find RECYCLE of more use if this automatic collection holds up a procedure when you want it to work without interruption. Inserting RECYCLE at a point when you can afford to wait means that garbage is collected at a time to suit you.

We will now consider the sorts of techniques which can be used to locate and remove bugs in procedures. As an example we will look at the development of a procedure that we have already seen. In Chapter 9 the procedure GETNUM was used to input a two-digit number with a flashing prompt at a point on the screen given by x- and y-coordinates. It uses a procedure, GETDIG, given in the last chapter:

```
TO GETDIG
   MAKE "DIG READCHAR
   MAKE "CODE ASCII :DIG
   IF OR :CODE = 13 :CODE = 12 [STOP]
   IF OR :CODE < 48 :CODE > 57 [GETDIG]
END
```

Firstly, GETDIG was tested by running it several times, pressing keys and printing the resulting values of DIG and CODE. This was satisfactory. Then GETNUM was entered as follows:

```
TO GETNUM :X :Y :NO
   SETCUR SE :X :Y
   TYPE :NO
   FLASH
   TYPE "?
   NORMAL
   GETDIG
   IF AND :CODE = 13 NOT EMPTYP :NO [OUTPUT :NO]
   IF AND :CODE = 12 NOT EMPTYP :NO [MAKE "NO BUTLAST
                                          :NO]
   IF AND :CODE > 13 COUNT :NO < 2 [MAKE "NO WORD :NO
                                          :DIG]
   OUTPUT GETNUM :X:Y :NO
END
```

You might find it interesting to type in this procedure and follow the various stages of debugging.

The procedure was tested by entering:

```
PRINT GETNUM 10 10 "
```

Since GETNUM is an operation, a command like PRINT was necessary to handle its output, but this was also a useful check on whether the resulting number was correct. A flashing question mark was shown at the position [10 10]. So far so good. When the key 3 was pressed, however, the message

```
5 is not TRUE or FALSE in GETNUM
```

appeared. Now, the only places where TRUE or FALSE were expected are the three IF commands. But which one is giving trouble? The instructions

```
PRINT "I1
PRINT "I2
PRINT "I3
```

were edited into GETNUM, one after each of the IF commands, and the procedure run again. Again 3 was pressed. The output was:

```
I1
I2
5 is not TRUE or FALSE in GETNUM
```

So the procedure passes the first two IFs and finds the third one troublesome. The three extra PRINT instructions were removed and, just before the third IF, were put two more commands:

```
PRINT :CODE
PRINT COUNT :NO
```

to test the two things involved in that IF. The procedure was run once more with the same input. This time it gave:

```
51
0
5 is not TRUE or FALSE in GETNUM
```

Well, 51 is the code for the character 3 and 0 is the number of characters expected in NO at that point. So where is the number 5 coming from? The two print statements were changed to ones which printed out the results of the conditions tested:

```
PRINT :CODE > 13
PRINT COUNT :NO < 2
```

This time the output was:

```
TRUE
5
```

and, inevitably:

```
5 is not TRUE or FALSE in GETNUM
```

Well, the first condition seems all right but what has happened to the second? It was time to test this on its own. The following instructions were typed in in command mode:

```
MAKE "NO "
PRINT COUNT :NO < 2
```

The second gave the message:

```
< does not like  as input
```

Now, the only way the < could be getting nothing as input is if it were taking :NO and not COUNT :NO. So the problem must be the order in which the

operations are done.

The solution, therefore, is to use brackets to make LOGO first of all find COUNT :NO and then check if this is less than 2. So the IF instruction becomes:

```
IF AND :CODE > 13 (COUNT :NO) < 2 [MAKE "NO WORD
                                   :NO :DIG]
```

Running the procedure and pressing the 3 key gives a figure 3 followed by the flashing question mark, exactly what was intended. It seems that LOGO behaved a little differently inside the procedure than in command mode; for somehow, in the procedure, it decided that :NO < 2 was FALSE. The COUNT of this was 5, which of course could not form an input to AND.

Is the procedure working satisfactorily? After pressing 3, the 2 key was pressed. The display showed 32 followed by the flashing question mark. After ENTER was pressed the number 32 was displayed as the output from the PRINT command. A few more tests revealed that keys other than digits were ignored, and so were any extra digits after two had been typed in, exactly as was required.

The next thing to test was the delete key. The procedure was run and the digit 2 typed. The display showed:

```
2?
```

The delete key was pressed; the display showed:

```
??
```

An examination of the procedure suggested that the first of these two question marks was the one which should be there, while the second was the one which had been there previously and should be removed. The technique decided upon, as already explained, was to remove the prompt, as soon as a digit was accepted, by printing a space on top of it. This was retried. As before, 2 was first pressed, giving:

```
2?
```

Then delete:

```
?
```

Good! Now another digit, 3. The message

```
Word doesn't like [] as input in GETNUM
```

appeared. Now, what could be causing that? This time there was only one use of WORD so there was no difficulty in locating the troublesome line. But the message quite clearly stated that the input was an empty list, not a word. This had only occurred after using delete so it must be the statement that handled delete which was the source of the problem.

It was easy to experiment with that sort of statement:

```
MAKE "NO 2
MAKE "NO BUTLAST :NO
```

At this stage

```
PRINT :NO
```


gave a blank line as you would expect. However:

```
MAKE "NO WORD :NO 3
```

repeated the error message from the procedure. Now, in addition to PRINT and TYPE, which we have already met, LOGO provides a third output instruction which shows the outermost brackets on any list – a useful check. Using it

```
SHOW :NO
```

gave []

It seemed that BUTLAST was creating a problem. The instruction

```
SHOW BUTLAST 5
```

gave immediately:

```
[]
```

which exposed the unusual behaviour of this operation. It is possible that future versions of Spectrum LOGO will be changed to make BUTLAST output an empty word under these circumstances, as BUTFIRST does, in which case you will not need the solution, explained in Chapter 9, of testing to see if NO is empty and ensuring, in this case, that it is the empty word.

This example illustrates some important techniques which can be used to get things to work properly. Firstly, test procedures separately. Although LOGO will helpfully tell you the name of the procedure in which it found an error, the problem may be that the output from another procedure is not what it ought to be. It is always a good idea to try each procedure alone before incorporating it into a more ambitious scheme. Sometimes you may need to write a special testing procedure to do this. For instance, if you have a procedure to produce, at random, a list which will be the name of a playing card such as

```
[QUEEN OF HEARTS]
```

then you will need a testing procedure like

```
TO TESTCARD
   PRINT CARD
   TESTCARD
END
```

in order to assure yourself that the cards that it produces really do seem to be at random. If you have thoroughly tested this alone, then when you run into trouble with a procedure using it, which might be called DEALHAND, you can at least start by saying 'now I know that CARD works properly'. It has to be admitted, though, that even then there may be occasions when you are wrong.

The second technique is *tracing*. This means getting some sort of output at intermediate stages so that you can see where LOGO has got to. For instance, in the procedure to choose the name of a playing card, you might decide not to allow one that has been chosen already. Certainly one way to do this would be to keep a list, called CLIST for example, initially empty, of all

the cards chosen and check for membership of this. So with a deliberate error, the end of CARD could be:

```
IF MEMBERP :CNAME :CLIST [MAKE "CLIST SENTENCE :CLIST
                              :CNAME OUTPUT :CNAME]
                         [OUTPUT CARD]
END
```

With this instruction in place, the procedure would *hang*, the term used to describe a situation when the computer apparently does nothing, giving no output and accepting no input. Suitable messages printed out at points throughout the procedure, like the instructions

```
PRINT "I1
```

etc. that were used above, would reveal that LOGO was doing something, and might, if inspected carefully, reveal that this final instruction was the cause of the trouble. (If you have not spotted why, remember that, when first used, CLIST will be empty. Whatever card is selected will not be a member and CARD will be called recursively and repeatedly. The two instruction lists should be in reverse order.)

An alternative form of tracing would be to include a PRINT command at the start of every procedure giving its name. This would be useful if the interconnections of the procedures were complicated so that it was difficult to follow how the action of the program moved from one procedure to another.

Either of these two forms of tracing might be usefully combined with *single stepping*. If results are likely to appear on the screen so quickly that you will not have time to follow what is happening, you can slow the action down by introducing the use of READCHAR to make the computer wait until you are ready to go on. So you might begin the procedure CARD with

```
TO CARD
   PRINT [CARD STARTED]
   MAKE "G READCHAR
```

These two instructions can be taken out later when you are sure the procedure is working properly. Of course, you would need to be sure that "G was not being used as the name of something else, whose value you would destroy with the MAKE command.

A third technique used in my example above, is that of printing out values of variables at intermediate places. This is of particular importance when the method of deciding the values is particularly complicated or where something seems to be going wrong which should not be possible if the variables have the values they ought to have. This method ought to be used in testing procedures even before you have reason to think they are going wrong. So, if the procedure CARD is going to output for use in a procedure DEALHAND, it might be a good idea to include a statement to show what card is actually produced each time it is used:

```
MAKE "CNAME CARD
PRINT :CNAME
```

so that the cards which are in the 'hand' at the end of dealing correspond to the ones actually produced by CARD. Again this could be combined with the use of READCHAR if necessary to give you time to note down the results. Instructions like this can also be removed before the procedure is said to be finished. As happened in my example, the SHOW command is frequently of use for this purpose because it displays lists as lists, including their outermost brackets.

Finally, a group of three procedures which can be used to make procedure-tracing a little easier. To use them with a procedure called PROC (for instance), type in:

```
TRACE "PROC
```

Then, whenever PROC is called, each of its lines will be printed out before it is obeyed. The procedure will then wait until you press a key before continuing. To switch off this effect, type:

```
UNTRACE "PROC
```

These instructions can be used from within another procedure if required. The method needs a lot of memory space since it creates a new version of the procedure to be traced, renaming the old one by putting a full stop in front of its name. For this reason it will not function if you already have a procedure named in this way. It may also cause problems if you have used a full stop or ".P as the name of a variable. Three new primitives are used: COPYDEF, which makes a new copy of a procedure; TEXT, which converts a procedure into a list of instructions which LOGO can handle; and DEFINE, which converts such a list of instructions into a procedure. You should consult LOGO 2 for an explanation of these instructions and use what follows here as a demonstration of their use.

```
TO TRACE :APROC
   IF DEFINEDP WORD ". :APROC [PRINT SENTENCE :APROC
                                [ALREADY TRACED] STOP]
   IF NOT DEFINEDP :APROC [PRINT SENTENCE :APROC [NOT A
                           PROCEDURE] STOP]
   COPYDEF WORD ". :APROC :APROC
   MAKE :APROC TEXT :APROC
   MAKE ".P LIST FIRST THING :APROC SENTENCE "PRINT
                              WORD CHAR 34 :APROC
   TREST BUTFIRST THING :APROC
   DEFINE :APROC :.P
END
TO TREST :ALIST
   IF EMPTYP :ALIST [STOP]
   MAKE ".P LPUT LIST "PRINT FIRST :ALIST :.P
   MAKE ".P LPUT FIRST :ALIST :.P
```

```
    MAKE ".P LPUT MAKE ". READCHAR :.P
    TREST BUTFIRST :ALIST
END
TO UNTRACE :APROC
    IF NOT DEFINEDP WORD ". :APROC [STOP]
    COPYDEF :APROC WORD ". :APROC
    ERASE WORD ". :APROC
END
```

You could, if you wished, extend TRACE to provide more information. This would require some study of the TEXT operation in *LOGO 2* and might possibly use more space in memory than you could afford. Practical debugging is more likely, therefore, to be successful using the general methods outlined in this chapter. Happy bug hunting!

Index

Symbols

Definitions of symbols appear on the following pages: