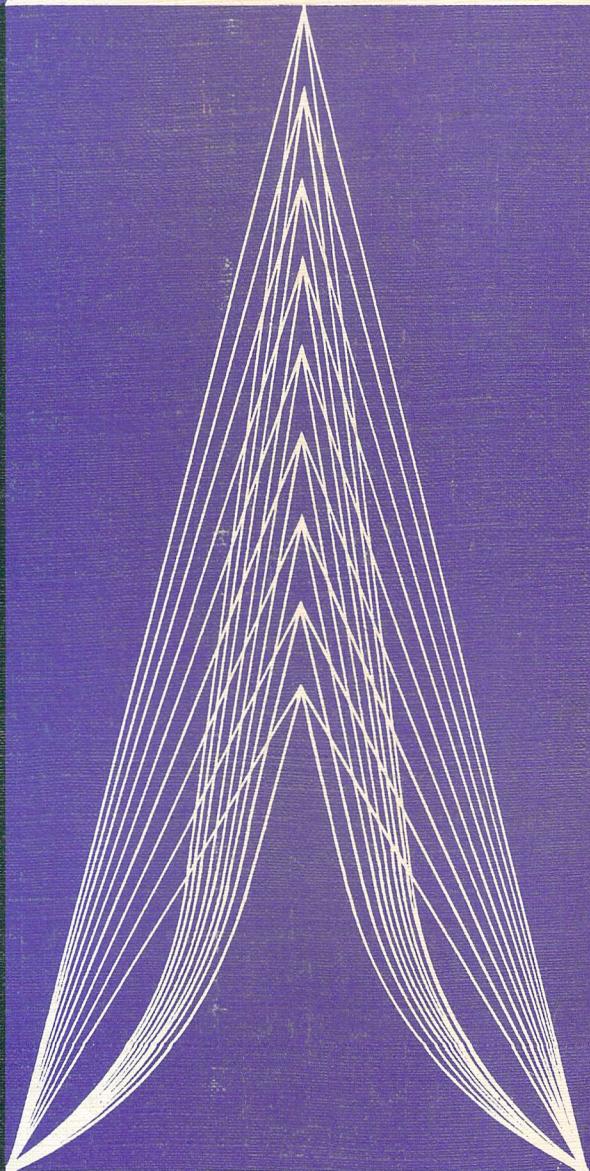


# Foundations in Computer Studies with COMAL

John Kelly

Second Edition

The Educational Company





# **Foundations in Computer Studies with COMAL**

**JOHN KELLY**

**THE EDUCATIONAL COMPANY**

First published 1983  
Second edition 1984

The Educational Company of Ireland Limited  
incorporating  
Longman, Browne & Nolan Limited  
Ballymount Road, Walkinstown, Dublin 12

© John Kelly 1984

Cover design, diagrams and cartoons: Terry Myler

### **Acknowledgements**

It is a pleasant duty to express my deep gratitude to the following for their help in the making of this book:

Mr. William McNamara, Principal of Wicklow Technical School for his unstinting generosity in giving me access to an Apple Computer System.

Dr. Frank Anderson, U.C.D., for the loan of some slides.

ICL for the use of some photographic material.

Lendac Ltd., for information on the ECONET microcomputer network and for the photograph of the network.

Professor C. J. van Rijsbergen, U.C.D., for some fresh material and new insights into the history of computing.

Finally, I was once again privileged to work with Miss Ursula Daly of the Educational Company of Ireland whose extraordinary editorial gifts, professionalism, perspicacity and pedagogic judgment wrought the usual magic transformation on the original raw product.

### **Dedication**

To my wife Lillian, and my children John, Stephen, Paul, Clare, Jo Ann and Barbara.

Printed in the Republic of Ireland by  
Cahill Printers Limited, Dublin.

# Contents

List of Programs	v
Preface	vii
1 What is a computer? Input – Process – Output. The Keyboard. PRINT. Algorithms and Programs. Constants & Variables. Important System Commands	1
2 Arithmetic on the computer. Real and Integer operators. Operator precedence. Structure Diagrams. Using disks	17
3 Selection. IF ... THEN ... ELSE. ENDIF. ELIF. CASE statement	36
4 Iteration I. REPEAT ... UNTIL	58
5 Iteration II. FOR ... TO (DOWN TO) ... STEP. NEXT. Nested multiplication. Nested Loops	69
6 Iteration III. WHILE ... DO. ENDWHILE. Differences between REPEAT and WHILE loops. Diagrams to represent sequence, selection, iteration	85
7 Hardware. Input devices. Output devices. Coding of Information. Computer memory. The Central Processing Unit	96
8 Software. Compilers. Interpreters. Assemblers. Operating systems. Editors. Utilities. Software packages	113
9 History of computing. Origins. Babbage. The Punched Card. The New Age. Three generations of computers	118
10 Lists. Arrays. Strings. DIM. Logical operators. EOD. Five variable types. Substrings	128
11 Two-dimensional Arrays. String Arrays	147
12 Functions. System or Built-in Functions. User-defined Functions. Local and Global Variables. Recursion	162

13	Procedures. Parameters. Actual and Formal Parameters. Value and Reference Parameters. Closed Procedures. Sorting. Binary Search. Recursive procedures. Bubble Sort	197
14	Files. OPEN. CLOSE. EOF. Random or Direct Files. Record numbers. Trapping errors	234
15	Problem Solving and Program Production. Some Hints on Program Construction and Debugging. Procedural approach to a problem	254
16	Graphics. Turtle Graphics. Colour. Recursive Patterns. Movement. Sprites	264
17	Computers at Work. Education. Commerce and Industry. Process Automation. Law. Medicine. Science and Technology. Arts and Humanities. Government and Public Services. Information Retrieval	301
18	Social Implications. The Computer and Privacy. Data Banks and Communications. Education. Work and Leisure. Working with Computers — Careers	318
	Appendix 1 COMAL pre-defined functions	331
	Appendix 2 Some COMAL features not covered in the main text	334
	Appendix 3 A Basic Toolkit of commands especially useful for the beginner	337
	Appendix 4 Dealing with Errors	339
	Appendix 5 Editing with Apple COMAL	345
	Appendix 6 Notes on the Commodore-64 and BBC versions of COMAL	346
	<b>Index</b>	<b>351</b>

# List of Programs

No.	Title	No.	Title
P1	Print 4 numbers, 5	P29	Output months in season (Version 2), 52
P2	Print 4 numbers on one line, 6	P30	Give currencies of EEC (Version 2), 53
P3	Print 4 numbers on one line, 6	P31	Identify symbol entered at keyboard, 54
P4	Print 4 numbers using TAB setting, 7	P32	Grade examination marks, 56
P5	Print two string constants, 7	P33	Calculate mean of 5 numbers, 58
P6	Input and print a number, 8	P34	Calculate mean of examination marks (Version 1), 61
P7	Input a number following message and print number, 9	P35	Calculate mean of examination marks (Version 2), 62
P8	Using a string variable to vary print output, 10	P36	Calculate $7!$ (Version 1), 63
P9	Using READ & DATA, 11	P37	Play a number-guessing game, 65
P10	Read in 2 numbers, add them and output answer, 18	P38	Convert singular nouns to plural form, 66
P11	As Program 10, but with message in INPUT statement, 20	P39	Print squares of numbers from 1 to 30 (Version 1), 69
P12	Read in 2 numbers, multiply them and output answer, 20	P40	Print squares of numbers from 1 to 30 (Version 2), 70
P13	Calculate value of $A + B - C + D - E$ , 21	P41	Calculate Compound Interest, 72
P14	Calculate the value of $A + B(C + DE/C)$ , 22	P42	Calculate $7!$ (Version 2), 73
P15	Add, subtract, multiply and divide 2 numbers, 23	P43	Evaluate $1 + 3 + 5 + 7 + \dots + 199$ , 73
P16	Add, subtract, multiply and divide 2 numbers, 28	P44	Tabulate values of $10x^3 - 13x^2 + 19x - 25$ (Version 1), 74
P17	Convert yards, feet, inches to metres, 30	P45	Tabulate values of $10x^3 - 13x^2 + 19x - 25$ (Version 2), 76
P18	Round a number off to 2 decimal places, 31	P46	Solve cubic equation $10x^3 - 13x^2 + 19x - 25 = 0$ , 77
P19	Convert a singular noun to plural form (Version 1), 32	P47	Draw a rectangle, 79
P20	Convert a singular noun to plural form (Version 2), 37	P48	Draw a triangle, 81
P21	Find the absolute difference of 2 numbers, 38	P49	Draw an upside-down triangle, 82
P22	Calculate $A/(B - C)$ , 40	P50	Find how long for investment to reach target (Version 1), 86
P23	Calculate income tax, 41	P51	Find how long for investment to reach target (Version 2), 87
P24	Convert a singular noun to plural form (Version 3), 43	P52	Find smallest power of 2 greater than given number (Version 1), 89
P25	Convert a singular noun to plural form (Version 4), 44	P53	Find smallest power of 2 greater than given number (Version 2), 90
P26	Output months in season (Version 1), 46	P54	Find if a number is prime, 92
P27	Give currencies of EEC (Version 1), 48	P55	Find mean absolute difference of a set of marks, 132
P28	Convert singular noun to plural form (Version 5), 50		

<b>No.</b>	<b>Title</b>
P56	Print every second element of an array, 135
P57	Print a list in reverse order, 135
P58	Find highest number in a list (Version 1), 137
P59	Count number of times the letter A appears in a string, 138
P60	Reverse a string, 139
P61	Find out if a string is a palindrome, 140
P62	Reverse christian name, surname (Version 1), 143
P63	Reverse christian name, surname (Version 2), 144
P64	Print an addition table (Version 1), 149
P65	Print an addition table (Version 2), 150
P66	Find largest element in an array, 152
P67	Swap two rows in an array, 153
P68	Calculate column totals, 156
P69	Deal with a list of stolen cars, 159
P70	Illustrate use of TAB function, 164
P71	Print a square, 164
P72	Draw a simple pattern, 165
P73	Draw a triangle, 166
P74	Draw a parabola, 167
P75	Draw a sine graph, 167
P76	Draw a cosine graph, 169
P77	Draw a graph of sine + cosine, 170
P78	Draw a graph of the exponential function, 172
P79	Draw a graph of a damped oscillation, 173
P80	Print some random numbers, 175
P81	Play a simple game of dice, 176
P82	Display an addition question, 179
P83	Give a set of addition questions, 182
P84	Function Example 1, 184
P85	Function Example 2, 185
P86	Function Example 3, 185
P87	Show two functions in one program, 185
P88	Show one function using another, 186
P89	Find area of a circle using a function, 187
P90	Find curved surface area and volume of a cylinder using functions, 187
P91	Solve a quadratic equation, 191
P92	Calculate powers of 2, 192
P93	Calculate Fibonacci numbers, 194
P94	Find a substring of a string, 200
P95	Factorise a number (Version 1), 204
P96	Factorise a number (Version 2), 205
P97	Test for a leap year, 209
P98	Factorise a number (Version 3), 211
P99	Illustrate value parameter, 212
P100	Illustrate reference parameter, 213
P101	Illustrate open procedure, 214
P102	Illustrate closed procedure, 214
P103	Find highest number in a list (Version 2), 215
P104	Find highest 3 numbers in a list, 217
P105	Sort a list of numbers into descending order, 218
P106	Illustrate binary search, 222
P107	Bubble Sort, 225
P108	Sort into alphabetical order (Version 1), 227
P109	Sort into alphabetical order (Version 2), 230
P110	Write a record to a file, 236
P111	Recover a record from a file, 237
P112	Set up a simple stock file, 238
P113	Use stock information, 241
P114	Set up stock file with reorder information, 242
P115	Stock management program, 244
P116	Set up a library file, 247
P117	Check library file, 248
P118	Find all books on chess in library file, 249
P119	Set up a telephone file, 250
P120	Find name of person with given telephone number, 251
P121	Draw a house, 260
P122	To draw a pentagon, 268
P123	To draw a dodecagon, 268
P124	To draw a circle (1), 269
P125	To draw a circle (2), 270
P126	To draw various closed figures (1), 271
P127	To draw various closed figures (2), 273
P128	To draw various closed figures (3), 274
P129	To draw some spirals, 275
P130	To rotate various closed figures, 276
P131	To draw various spirals, 278
P132	To stitch a pattern, 279
P133	To draw a face, 283
P134	To draw an imploding, exploding square, 285
P135	To draw a moving point, 286
P136	To draw a moving ball, using a sprite (1), 290
P137	To draw a moving ball, using a sprite (2), 291
P138	To draw a moving ball, using a sprite (3), 293
P139	To fill the screen with moving sprites, 296

# Preface

This book attempts to provide a comprehensive foundation course in Computer Studies and a sound introduction to the principles and practice of structured programming. It reflects the author's conviction that Computer Science is an exciting and enjoyable pursuit in its own right and requires no special expertise to profit from its study. It is designed to appeal to a wide variety of students and does not depend on any prior knowledge in any field.

The ability to program a digital computer has become a highly-prized one in modern society. There is no doubt that, properly taught, it has enormous educational value in motivating students, in developing clarity and comprehension, in enhancing problem solving, organisational and communication skills, etc. To reap this rich harvest it is now widely considered that students should be exposed to modern ideas and techniques in structured programming. COMAL provides an elegant example of a structured programming language and is eminently suitable for use in schools. COMAL was originally designed in 1974 by Borge R. Christensen and Benedict Lofstedt of Denmark, and has since undergone extensive revision and enhancement. Today it is being actively considered in several countries as a significant educational language.

This text provides ample material to enable the student

- (1) to appreciate structured problem solving
- (2) to develop skill in COMAL programming
- (3) to gain an insight into the historical development of the modern electronic computer
- (4) to study many diverse applications of computers
- (5) to appreciate and think about the social implications of computers
- (6) to develop an understanding of the combination of hardware and software which constitutes a complete computer system.

A total of 121 fully-tested COMAL programs is included. The examples have been carefully selected to illustrate fundamental programming points and many have been kept deliberately simple. They are not intended to work under all circumstances, because to achieve this would often require a degree of complexity which would obscure the pedagogical point at issue. Students will no doubt see an exciting challenge in changing the programs to do better. They should be encouraged to modify the programs, to clarify them, to expand them, to change their structure — in short, to use them as material for experimentation. Some program modifications are suggested in the exercises, but students and teachers will surely think of many more. In any event, extensive practice is required to develop programming skill.

Although the coverage of COMAL is complete in terms of structure and range, not every single aspect has been covered, as this was not required by the strategy of the text and would have made the book too long. The most significant omissions are the GOTO and GOSUB statements. These were deliberately avoided as they are widely considered to interfere with the development of good programming style and are somewhat anomalous in the overall design of the language. For those, however, who

are curious about such things, most of the additional features of COMAL are discussed briefly in Appendix 2.

The use of the more modern structure diagrams in place of flowcharts was prompted by their greater accord with structured solution development. It is hoped that they will be of great assistance to students in their own problem solving.

One of the principal aims of a schools Computer Studies course is to equip students to understand the central role of the computer in 20th century civilisation and to be in a position to take responsible decisions concerning its use. Consequently no opportunity should be lost to debate and explore the social applications and implications of computers.

### **Preface to the Second Edition**

In introducing the second edition of 'Foundations in Computer Studies with COMAL' I am delighted to express my thanks to the many who commented favourably on the first edition and to those who pointed out some minor flaws and made suggestions for improvement. I hope I will be forgiven if I single out Mike Norris of the Computer Centre in UCD for special thanks for the benefit of his extraordinary knowledge and skill and his generous advice.

This second edition includes corrections to all the errors detected in the first edition. However, the major improvements are

- (1) a completely new chapter on Graphics
- (2) three new appendices.

At the time the original book was being written, COMAL had no graphics commands. A measure of the growing popularity of this elegant language is its increasing availability and use. It is now implemented on Digital's VAX and PDP11 range, and on the BBC and Commodore-64 micros, as well as on other machines such as the Apple, PET, etc. The Commodore-64 version seems to have the most extensive range of Graphics instructions and this was the version used to prepare the new chapter. In this connection I must express my profound gratitude to Tomorrow's World, Dundrum, Dublin, who allowed me extensive use of one of their Commodore-64 systems in the preparation of this work. The programs in Chapter 16 were developed on this system. The listings of these programs show one or two variations from the Apple version used in the rest of the book. These are all relatively simple and easily understood. That is the beauty of a standardised language like COMAL.

The graphics chapter comes somewhat late in the text. This should not be taken as an indication that graphics are difficult. In fact the contrary is the case, and readers are encouraged to explore this exciting world as early and as often as they wish.

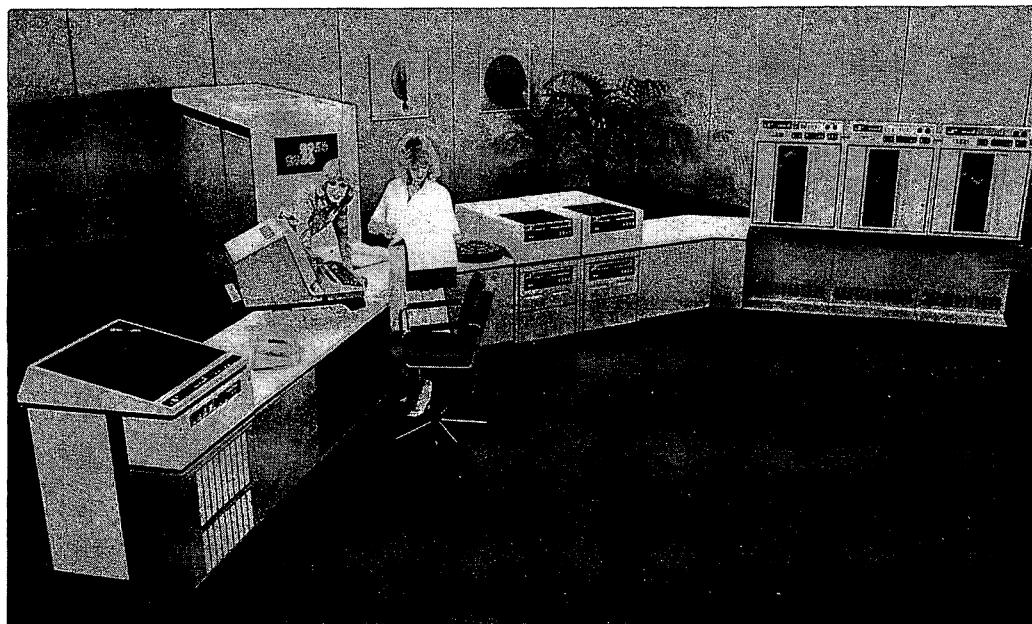
The other additional material in this edition is placed in three appendices. One of these deals with common errors and how to detect and deal with them and is in response to a valuable suggestion from Fr. T. Byrne of Blackrock College, Dublin. Another appendix discusses editing with Apple COMAL. The third draws attention to some features of the Commodore-64 and BBC versions of COMAL.

*John J. Kelly*

*Department of Computer Science  
University College, Dublin*

*July 1984*

# **1 Introduction: Print: Some System Commands**



## **What is a Computer?**

Consider these tasks:

- (a) Find the average age of the marathon winners in the last ten Olympic Games.
- (b) Count the number of vowels in a page of this book.
- (c) Set up a file containing the names and addresses of all the students in the school.
- (d) Sort the names listed in (c) into alphabetical order.
- (e) Using information on rates of pay, hours worked, tax and social welfare deductions, etc., produce a weekly payroll for a firm.
- (f) Draw a design for a house from an architect's specification.

Computers can be used to perform all these tasks and more. In fact there are very few problems to which computers have not been applied at one time or another, from performing routine business calculations to assisting man's flight to the moon, from trying to understand human speech to monitoring patients who are critically ill.

What, then, is a computer?

A computer is an information-processing system. Some people prefer to use the word **data** instead of information, in which case they would refer to the computer as a data-processing system. Both numeric and non-numeric data can be processed. For example, in calculating the average age of marathon winners, the computer would use the winners' ages as data. In sorting students' names into alphabetical order, the data would be the students' names.

The computer cannot, of course, process information entirely on its own. People must supply the data, give exact instructions as to how it is to be processed, specify what results are required and indicate in what form they are to be supplied. The activity of giving instructions to a computer is called **programming**. Our first exciting task is to learn how to write programs, i.e. how to give the computer exact instructions to enable it to perform whatever task we require.

## Talking to a Computer



There are three clear stages in the processing of information:

INPUT - PROCESS - OUTPUT

The data to be processed must be given to the machine (INPUT); the data must then be manipulated or transformed (PROCESS) and the results must be returned to the programmer or user (OUTPUT). Let us take two very simple problems to illustrate the point:

1. Find the sum of 93 and 58.

INPUT : the values 93, 58

PROCESS : add

OUTPUT : the result 151

2. Find the longest word in this list: John, Paul, Steve, Barbara, Joann, Clare.

INPUT : the list of words John ... Clare

PROCESS : find the length of each word (i.e. the number of letters in it) and compare lengths

OUTPUT : longest word, BARBARA

Generally speaking we cannot simply say to a computer 'find the sum of 93 and 58', or 'find the longest word in the following list'. There are several reasons for this:

- (1) Most computers are not designed to respond to the human voice. So we must type in our requests on a keyboard which is similar to a typewriter-keyboard.
- (2) 'Ordinary' English cannot be used in communicating with the computer. Languages such as Irish, or English, or French, which have been developed by human beings, are too rich and complex to allow computers to process them. Often, too, sentences in these languages are ambiguous, e.g. I saw the boy with the telescope. Computers cannot tolerate ambiguity. They must be given clear unambiguous instructions, in a language suitable for machine processing. One such language is COMAL. This is the language we are about to study.
- (3) A computer cannot be asked simply to 'solve' a problem. Detailed instructions on *how* to solve the problem must be given. Such a list of instructions is called an **algorithm**. An algorithm is similar to a recipe, or a knitting pattern. It spells out clearly the steps to be taken to produce the desired result.

When an algorithm is written in a suitable language, such as COMAL, it is called a **program**. Thus a program is a list of precise instructions designed to solve a problem.

Corresponding to the three essential phases of problem solving - INPUT, PROCESS, OUTPUT - we have input, processing and output machinery or hardware. This will be studied later on.

## The Keyboard

It is important to become familiar with the keyboard on your computer, since it is the primary means through which you will communicate with the machine.



Notice that all the letters of the English alphabet and all the digits 0 to 9 are represented. In addition there are keys for the symbols +, -, <, (,), : and many others.

Some of the keys involve two symbols or characters, e.g. the key . If you press this key with your finger the computer will accept the digit 9. In order to communicate the bracket to the machine you must press another key – the key with the word SHIFT on it – at the same time as you press the key. In general, in order to communicate the symbol on the upper part of a two-symbol key you must press the SHIFT key along with the appropriate two-symbol key. There is one exception – the key . Experiment with the keyboard to see if you can ring the bell!

The best way to get to know the keyboard is to practice with it. The characters you type will appear on the screen – the Visual Display Unit (VDU) – attached to your computer. For the moment don't worry if the computer responds in a strange way at times. You will learn the meaning of these responses later. Be reassured that you cannot damage the machine by typing away at the keyboard.

Besides the keys we have mentioned above there is one other important key to be noticed at this stage. This is the key marked RETURN. We will see the significance of it very shortly.

## The PRINT statement

Let us begin our study of programming by examining how to get output from the computer. The **PRINT** instruction is used to cause the computer to give out information, e.g. PRINT 47 will cause the number 47 to be displayed on the screen.

Try the PRINT instruction on your computer. Enter the command PRINT 47 and then press the RETURN key.

**N.B.** Each line of instruction must be terminated by pressing the RETURN key. The machine will not accept the instruction until you do that – try it and see! Sometimes instructions are so long that they run to more than one line on the screen. In this case RETURN is not pressed until the end of the instruction is reached.

**The rule is: At the end of each instruction  
press RETURN.**

The simple instruction PRINT 47 does not constitute a program. It makes use of the 'immediate execution mode' of COMAL. This means that the instruction is executed (i.e. carried out) immediately after the RETURN key is pressed. You should explore the action of the PRINT statement in immediate execution mode yourself, by typing in some examples, e.g.

```
PRINT 17
PRINT "A"
PRINT "JOE"
PRINT 89.732
```

Now type PRINT A and press RETURN. You will probably find that the computer will respond by telling you that there is an error in the command you have just given it. Don't worry about this for the moment. To recover from the situation and get yourself back into a position to carry on, press the key marked ESC (the 'escape' key). You may then continue. The ESC key is very important as it often allows you to recover from puzzling situations.

## Little Programs

Let us now construct a program using the PRINT statement. Remember that a program is a list of instructions. Our first program will simply print several numbers in succession.

```
0010 PRINT 12
0020 PRINT 41.6
0030 PRINT 18.47
0040 PRINT 0
0050 END
```

Note carefully that each line of the program begins with a line number. The program is executed in the order of the line numbers. Thus line 10 is executed before line 20, line 20 before line 30, etc. It is not necessary to have a gap of 10 between the line numbers. We could have simply numbered the lines 1, 2, 3, 4, 5. However, it is a good idea to leave a gap in case you want to insert extra lines into your program at some stage. We will look at an example of this later.

The END statement signals the end of the program. Normally it is the last statement in every program.

If you have taken the trouble to type in the above program (not forgetting RETURN at the end of each line) you will find that when you are finished nothing happens - the machine does not seem to want to carry out your commands.

The question now is - how can you get your instructions executed?

The answer is - simply type RUN. As soon as you type RUN and press RETURN, the program will be executed and you will see this output:

```
RUN
12
41.6
18.47
0
```

RUN is called 'a system command'. It causes the COMAL system in the computer to carry out the instructions in a COMAL program.

Note that each number in the result above is printed on a separate line. If you wish to have the values printed all on one line, just place a semicolon at the end of each PRINT instruction line, thus:

```
0010 PRINT 12;  
0020 PRINT 41.6;  
0030 PRINT 18.47;  
0040 PRINT 0  
0050 END
```

RUN

```
12 41.6 18.47 0
```

However, the same effect can be achieved by this more compact program:

```
0010 PRINT 12;41.6;18.47;0  
0020 END
```

RUN

```
12 41.6 18.47 0
```

Note that there is just one space between consecutive numbers.

## Print Zones

The spacing of the numbers output by the computer may be changed by using the TAB command. For example the statement

```
TAB := 6
```

will cause each zone to be set to six spaces wide. We say that the 'field width' or 'zone width' in the output is six. Essentially this means that six spaces are reserved for printing each value.

Note carefully the symbol := which we read 'is assigned the value'. We will meet this symbol over and over again throughout this book. The full statement TAB := 6 reads as follows - 'TAB is assigned the value six'.

In order to make use of the wider field width set by TAB, we must use a comma instead of a semi-colon between the items in the PRINT instruction, thus:

```
PRINT 12, 41.6, 18.47, 0
```

Try setting TAB to different values and then printing that same list of numbers.

What happens if the number you want to have printed is too big to fit into the field width which you have set? Not to worry! The number is printed anyway and the next value is moved out one or two zones as necessary. The best way to see what happens is to experiment. One thing you will notice is that numbers are printed to about seven significant digits.

If you do not set the TAB value yourself, the computer takes it to be zero. This means that numbers separated by commas are printed right next to each other. Confusing, isn't it? So we will normally set a TAB value in our programs whenever we are going to use a comma in our PRINT statements.

```
0010 TAB:=6  
0020 PRINT 12,41.6,18.47,0  
0030 END
```

RUN

```
12    41.6  18.47 0
```

## STRINGS

Numbers are not the only items of data which we may have printed by the PRINT instruction. Consider the following:

```
PRINT "HELLO"
```

Here we request the sequence of letters H, E, L, L, O to be printed. Such a sequence of letters is called **a string**. Note that the string is enclosed in quote marks. A string may contain digits and other symbols besides letters, e.g.

```
PRINT "WATCH OUT!! "
```

In order to get a string printed it is essential to place it in quote marks. Try typing PRINT HELLO and see what happens. We will soon learn how to use statements like this and discover what meaning they have.

Just as we can write number-printing programs, we can also write string-printing programs, e.g.

```
0010 PRINT "COME IN NO. 4"  
0020 PRINT "YOUR TIME IS UP"  
0030 END
```

RUN

```
COME IN NO. 4  
YOUR TIME IS UP
```

## The NEW command

When we type in the instructions in our programs they are stored in a special part of the computer called its **memory**. We will study this important component later. When we are finished with one program and want to move on to another it is important to clear the first program out of the section of memory in which it is held, in order to make room for the new program. This section of memory is called our work-space.

To clear our workspace we use the command **NEW**.

Before we start working on any new program we should simply type **NEW** (followed by **RETURN**, of course). Otherwise our new instructions will become confused with the instructions already in our work-space.

## Constants and Variables

The programs we have written so far are very restricted. They simply cause **constants**, either numeric constants, e.g. 71.8, or string constants, e.g. "HELLO", to be printed. Every time we run a program the same things are printed. We cannot change what is printed from one run to the next.

Let us now consider how we *can* vary the information printed. The following program will serve as an example

```
0010 INPUT NUMBER  
0020 PRINT NUMBER  
0030 END
```

```
? 23  
23
```



We have a new instruction here - **INPUT**. It is not hard to guess that this command has to do with giving information or data to the computer. In fact when this instruction is executed the computer will print a question mark (?) on the screen and wait for us to respond. We must respond by typing a number. The computer will then immediately print the same number on the screen. The following runs show what happens:

```
RUN  
? 56.853  
56.853  
  
RUN  
? -34.76  
-34.76
```

Thus we may change the number entered from one run of the program to another. The number we type in is stored in the computer's memory in a location which we have named NUMBER. NUMBER is known as a **variable name** or **identifier**.

Let us write a slightly friendlier version of our last program.

```
0010 PRINT "TYPE IN A NUMBER"  
0020 INPUT NUMBER  
0030 PRINT "YOUR NUMBER IS ",NUMBER  
0040 END  
  
RUN  
  
TYPE IN A NUMBER  
?  
47.85  
YOUR NUMBER IS 47.85
```

Why is this a friendlier program? Because now when we run the program the first thing that happens is that the message "TYPE IN A NUMBER" appears on the screen, followed by the question mark which indicates that the computer is waiting for input. This is better than having a question mark on its own staring stonily at us from the screen.

Remember however that the PRINT statement (line 10) has no effect whatever on the INPUT. It merely prints something on the screen. It is the INPUT statement which alerts the computer to expect input. Even if line 10 was: PRINT "DO NOT ENTER A NUMBER", the computer would still wait for input as long as the INPUT statement (line 20) was present. Of course our message would be a bit silly, but the computer wouldn't mind.

Note very carefully that the word NUMBER on line 20 is *not* enclosed in quote marks. It is acting as the name of a number, i.e. as an identifier and must never be enclosed in quotes. Now we can see the difference between

#### PRINT NUMBER and PRINT "NUMBER"

In the first case we are asking for a *value* which is referred to by the name NUMBER to be printed, whereas in the second case we require that the actual *word* NUMBER be printed. In the first case if NUMBER has not been given any value the computer will warn us. The second case always works since the computer is being asked to print literally the sequence N, U, M, B, E, R.

Program 7 uses a mixture of constants and variables (actually just one variable - NUMBER). The constants are the strings "TYPE IN A NUMBER" on line 10 and "YOUR NUMBER IS" on line 30. Note that we can mix string constants and variable names in the one print statement. We will often use this facility.

## Easy does it

To make it easy for you to enter your programs, COMAL provides an AUTOMATIC line-numbering feature. If you type

### **AUTO**

COMAL will automatically put 10 on the screen and wait for you to enter an instruction. When you have entered the instruction and pressed RETURN, COMAL will place 20 on the screen and wait for another instruction. The system will continue to precede each instruction with line numbers 30, 40, 50, etc. until you indicate that you want to finish by typing ESC. Note that the AUTO feature will not stop when you enter the instruction END in the program - you must press ESC!

If you do not want to start your lines at number 10 you can indicate as much in the AUTO command

e.g. AUTO 50 will commence line numbers at 50.

Similarly you may indicate that you would like a spacing other than 10 between line numbers

e.g. AUTO 100, 4 starts at line 100 and continues in steps of 4, i.e. 104, 108, 112, etc.

## String Variables

Just as we can use names for numbers, so too can we use string variables or identifiers. Consider the following example

```
0010 DIM NAME$ OF 20
0020 PRINT "TYPE IN YOUR NAME"
0030 INPUT NAME$
0040 PRINT "HELLO ",NAME$
0050 PRINT "YOU ARE WELCOME TO THE EXCITING WORLD OF COMPUTERS"
0060 END
```

RUN

```
TYPE IN YOUR NAME
? BILLY BONES
HELLO BILLY BONES
YOU ARE WELCOME TO THE EXCITING WORLD OF COMPUTERS
```

RUN

```
TYPE IN YOUR NAME
? CHARLIE CHAMPION
HELLO CHARLIE CHAMPION
YOU ARE WELCOME TO THE EXCITING WORLD OF COMPUTERS
```

Here we use the identifier NAME to stand for the string representing the name of the person when it is input. Thus in the first run above NAME will get the value "BILLY BONES" and in the second run "CHARLIE CHAMPION".

The fundamental difference between a number identifier and a string identifier is that every string identifier *must* end with the dollar (\$) sign.

Now what about line 10 - do I hear you ask? Well, we have seen that a string is made up of a sequence of letters, digits or other characters. Line 10 tells the COMAL system how many symbols at most the string variable NAME\$ will contain, i.e. it tells COMAL the dimension of NAME\$ (DIM is short for dimension).

*Every string variable used in a COMAL program  
must first be dimensioned.*

Note that the variable NAME\$ need not ever actually contain 20 characters. As long as it does not try to contain more everything is OK. It may contain less if you like.

## **Note on Identifiers**

We have now met two types of identifier:

- (1) Identifiers for numbers, e.g. NUMBER, TOTAL.
- (2) Identifiers for strings, e.g. NAME\$, PART\$.

Identifiers are simply names for numbers or strings. We may choose any name we like so long as

- (1) it begins with a letter (not a digit or any other symbol)
- (2) it contains only letters, digits and in some systems the underscore character (\_)
- (3) it is not more than 80 characters long
- (4) it is not a special (reserved) COMAL word such as RUN, INPUT, etc.

When using an identifier we should choose names that convey some meaning to us, e.g. SUM, TOTAL, ADDRESS\$, etc. Although we *may* use up to 80 characters it is not advisable to use very long names as they may appear clumsy. Remember the important thing is to be as clear and readable as possible.

## **READ ... DATA**

As we have seen, the INPUT statement allows us to enter data at the keyboard for processing by a program. The program halts while it is waiting for us to respond with the numbers or strings it needs. There is however another way to give a program the data it needs. This is by including the data in the actual program itself. A simple example will illustrate this:

```
0010 READ A, B, C, D
0020 PRINT A;B;C;D
0030 DATA 12, 14.6, 178, 92.66
0040 END
```

RUN

```
12 14.6 178 92.66
```

The program uses four pieces of data which are actually placed in the program in the DATA line (line 30). Notice carefully that this data is associated with the four variables A, B, C and D by using a READ instruction (line 10) instead of INPUT. A **READ** statement in a program must be accompanied by one or more **DATA** lines, which contain as many items of information as are required to be read. We could if we wished have spread our data in the program above over four lines, thus:

```
DATA 12  
DATA 14.6  
DATA 178  
DATA 92.66
```

One advantage of using READ and DATA in a program instead of INPUT is that the program can get on with its work much more quickly, since it does not have to wait for us to enter the information it requires. It already has the information within the program. However we will usually use INPUT so that you can experiment with the programs yourself by varying the input from run to run.

## **Blankety Blank**

If you have been working diligently with the examples so far (and of course you have!) you have probably found that the screen has become cluttered with information. You would like to clear the screen from time to time. Easy! Just type

**CLEAR**

Immediately everything will be cleared from the screen and all you will see is a little blob (called a **cursor**) flashing away in the top left hand corner. But what about my program, you say! Don't worry. Your program is safe inside in your work-space. Only the screen has been cleared. To clear the work-space you must type NEW.

## **Program - come forth!**

If you have just cleared the screen how can you see your program? Easy again! Simply type

**LIST**

This causes a list of the instructions in the program in your work-space to appear on the screen.

You may also have just parts of a program listed if you wish, e.g.

LIST 20,70 causes lines 20 to 70 inclusive to be placed on the screen.

LIST, 50 causes all the instructions from the beginning to line 50 inclusive to appear.

LIST 40, causes all the lines from 40 to the end to appear.

*Note:* On some systems a minus sign or dash (-) is used instead of a comma in LIST commands, e.g. LIST 20-70, LIST-50, LIST 40- etc.

## **RENUMBER**

We may change the numbering of the lines of our program at any time by using the **RENUMber** command.

e.g. RENUM 100 causes the lines of the program to be renumbered 100, 110, 120, etc.

RENUM 100,3 causes the lines of the program to be renumbered 100, 103, 106, etc., i.e. starting at 100 and increasing in steps of 3.

RENUM 30:100,500,15 causes lines 30 to 100 inclusive to be renumbered to start at 500 and go up in steps of 15.

## **Errors and Editing**

Have you made any mistakes in entering the sample programs? You have! Good!! (If you haven't you should have - it's by mistakes you learn!)

How do we correct mistakes?

One way is to type in the offending line again from scratch. Suppose you wish to change line 40 in a program. Simply enter the number 40 followed by the instruction and when you press RETURN the new line 40 will replace the old one.

However it is often inconvenient and tedious to have to enter a whole line again. Suppose, for example, a program contains the line

40 PRINT "HAPPY BIRTHDAY", NAME\$

and you want to change the comma to a semicolon. Some systems allow you to do this quite easily by making use of a screen editing system. This gives you the facility of moving around the screen at will, deleting symbols, entering new symbols and changing symbols as you please.

Let us consider how we would accomplish the task of changing the above comma on the Commodore-64. (To see how it would be done on the Apple, see Appendix 5.)

On the upper right hand side of the Commodore-64 keyboard you will notice two keys marked CRSR and CRSR. Pressing these keys causes the cursor to move around the screen. Use the  $\uparrow$  key to move the cursor up the screen to line 40. Use the  $\Rightarrow$  key to move the cursor to the right until it is positioned over the comma on line 40. Now simply press the ; key to replace the comma by a semicolon. Press RETURN and the replacement is made. Now use the  $\downarrow$  key to move the cursor back down the screen.

You may make as many changes to a line as you wish, before you press RETURN. Also you may change as many lines as you wish, provided you press RETURN for each line.

## **The stubbornness of COMAL!**

You may have found that sometimes when you make a mistake in a line the COMAL system notices it and when you press RETURN the system will not accept the offending line; it tells you you have made an error and repeats the line with the cursor positioned somewhere near where the mistake has been detected. You cannot continue until you have corrected the error. (On the Apple, however, you can press ESC and the line will be erased and ignored.)

If you cannot work out the correct version of the line on the spot, you can simply delete the contents by using the DEL key and hope to correct it later when you have gathered your thoughts. Otherwise simply type the new characters over the wrong ones until the line is accepted.

## **Syntax Errors and Logical Errors**

There are essentially two kinds of error:

- (1) Syntax or grammatical errors
- (2) Logical errors.

Just as there are certain rules for forming correct sentences in English so there are rules for forming correct instructions in COMAL. When these rules are not obeyed a *syntax error* is made. "I done the job" is grammatically wrong in English. PRINT A:B:C is grammatically wrong in COMAL and if you enter it as an instruction the COMAL system will detect a syntax error and will stop at this line while you correct the error.

*Logical errors* occur when the instructions may be syntactically correct but do not achieve the effect which was intended. It is difficult to illustrate this with the simple programs we have developed so far, but as you write longer programs you will find that you frequently make logical errors. Not to worry! Eliminating logical errors (known as **debugging** programs) is an interesting and enjoyable task which can teach you a great deal.

While the COMAL system can readily detect syntax errors, it cannot detect logical errors.

## **Summary**

A computer is an information processing device.

A program is a list of precise instructions.

3 stages of processing: INPUT - PROCESS - OUTPUT.

Data can be numeric or string.

Identifiers are used to refer to information which can vary; they should be clear and meaningful.

The memory of a computer stores information.

COMAL strings must be dimensioned.

Errors are important learning tools.

Syntax errors can be corrected on entry or by using EDIT.

Logical errors can only be corrected using hard thinking!

### **COMAL KEYWORDS**

AUTO	CLEAR	DATA	EDIT	END	INPUT	
LIST	NEW	PRINT	READ	RENUM	RUN	TAB

## **QUESTIONS and EXERCISES**

1. What is a computer? Can you think of any way in which a computer is different from a calculator?
2. What is a program? Why is a program similar to a recipe or a knitting pattern? Can you think of other words with a similar meaning?
3. Compare the keyboard on your computer to a typewriter-keyboard. How many symbols are the same? How many are different? Examine the effect of pressing each key on your keyboard.
4. Explain clearly why data processing falls into three clear stages, input - process - output.
5. Why are special computer languages used? Find out the names of ten languages used in computer programming.
6. Why is a computer called an information processor? Can you think of any job which cannot be done by a computer? Is this job an information-processing job?

7. Write print statements to display the following:  
(i) 57   (ii) 78.9   (iii) 512.4173865   (iv) your name  
(v) your address   (vi) a single letter in the centre of the screen.
8. Write a program which accepts a person's name and prints a personal greeting.
9. Write programs to print each of the following patterns:

(i)      \*\*\*\*  
        \*\*\*\*  
        \*\*\*\*

(ii)      \*  
        \*\*\*  
        \*\*\*\*

(iii)      \*  
        \*  
        \*  
        \*

(iv)      \*  
        \*\*\*  
        \*\*\*\*\*

(v)      \*  
        \*\*\*  
        \*\*\*\*\*  
        \*\*\*\*\*  
        \*\*\*  
        \*\*\*\*

(vi)      \*  
        \* \* \*  
        \* \* \* \*  
        \*\*\*

10. Write a program to read a name and address and print it in indented form (i.e. starting a little bit further in from the left for each line), e.g.

BILLY D. BONES,  
THE OLD YARD,  
GRAVETOWN,  
NOWHERE.

11. Write a program to draw a circle.
12. Write a program which writes a letter to a friend.

## 2 Arithmetic on the Computer

### Calculating your advantage

Having seen how to get information into the computer and how to get it back out again, we now turn our attention to using the computer to perform arithmetic. A simple example using immediate execution mode will show how easy this is.

PRINT 23 +19

When RETURN is pressed, the number 42 will appear on the screen. The computer recognises the usual + sign for addition and automatically adds the two numbers. Similarly PRINT 12.2 + 8.73 will give 20.93,

PRINT 17 + 8.6 will give 25.6, etc.

Note that it is not sufficient to simply type 23 + 19 in order to get the two numbers added. PRINT must be included. Try entering 23 + 19 and see what happens. You will probably see a message such as ERROR 46: STATEMENT EXPECTED on the screen. What does this mean?

It is the computer's way of telling you it does not understand your instruction. It is really a rather dumb machine, even though it can be made to do some clever things. How does the COMAL system misinterpret your command? Well, it assumes that 23 is a line number for an instruction which forms part of a program (remember every line in a program is numbered). Then it sees the + sign following the line number (as it supposes). Since it is expecting an instruction to start after a line number and since no instruction can start with a + sign it gives out a howl of protest - STATEMENT EXPECTED.

So the rule is: to get arithmetic done in immediate execution mode, use PRINT.

Now what about the other arithmetic operations besides addition? They are all just as easily done:

the symbol for subtraction is the ordinary -

e.g. PRINT 27 - 19.5 gives 7.5

the symbol for multiplication is \*

e.g. PRINT 7 \* 9 gives 63

the symbol for division is /

e.g. PRINT 13/5 gives 2.6



## **Integer Arithmetic**

We have just introduced the operators for ordinary arithmetic, + (add), - (subtract), \* (multiply), / (divide). These operators work with whole numbers and real (decimal) numbers.

In addition to these four there are two special operators which work only on integers (whole numbers). These operators are represented by short words rather than single symbols. They are:

(1) DIV which gives just the integer quotient when one integer is divided by another

e.g. 17 DIV 5 gives 3

46 DIV 9 gives 5

(2) MOD which gives the remainder when one integer is divided by another

e.g. 17 MOD 5 gives 2

46 MOD 9 gives 1

Try these operations on your computer. Don't forget to use PRINT to see the results.

## **Arithmetic Programs**

So far we have used immediate execution mode to illustrate how arithmetic is performed in COMAL. Let us turn now to some simple programs.

```
0010 // COMELY KATE
0020 // TO READ IN TWO NUMBERS, ADD THEM AND OUTPUT THE ANSWER
0030 PRINT "ENTER FIRST NUMBER"
0040 INPUT NUMBER1
0050 PRINT
0060 PRINT "ENTER SECOND NUMBER"
0070 INPUT NUMBER2
0080 SUM:=NUMBER1+NUMBER2
0090 PRINT
0100 PRINT "THE SUM OF ",NUMBER1," AND ",NUMBER2," IS ",SUM
0110 END
```

Our program begins with a couple of remarks. Each remark begins with two strokes (//) which serve to inform the COMAL system that the line is only a remark and is not to be treated as an instruction. Remarks may be inserted anywhere in a program and do not affect the execution of the program. Their purpose is to make the program more readable and to help a human reader to understand more easily what it is about. As such they can be extremely helpful and should normally be included in every program. At the very least, each program should have two opening remarks:

- (1) giving the name of the programmer
- (2) indicating the aim of the program.

Following the opening remarks line 30 causes a message, ENTER FIRST NUMBER, to be printed on the screen. As we know, this is so that when input is requested (line 40), the person using the program will know what to do and not just be faced with a puzzling question mark glaring at him from the screen.

Remember that printing the message ENTER FIRST NUMBER does not cause

the computer to expect input - only the input statement can do this. If the input line were omitted the computer would just carry on with the rest of the program and the puzzled user would be left staring at the invitation to enter a number and wondering why the number he typed was not accepted.

When the first number is typed in, it is stored in the computer's memory in a location called NUMBER1.

On line 50 we have a simple PRINT command on its own. This causes a line to be skipped, or, if you prefer, a blank line to be printed. Why should we do this? Well, it enables us to space our output more pleasingly and not have information too cluttered on the screen. We will frequently use such PRINT (blank line) statements in our programs and you should be aware of their purpose.

Line 60 prints another message and line 70 accepts a second number, which is stored in NUMBER2.

Line 80 causes the values stored in NUMBER1 and NUMBER2 to be added together and the result stored in a location in memory called SUM. Once again we meet the symbol :=. Remember it stands for IS ASSIGNED THE VALUE. So the instruction reads 'SUM is assigned the value got by adding NUMBER1 and NUMBER2'.

Another blank line is printed when line 90 is executed.

Before commenting on line 100 let us see what happens when we run the program.

RUN

ENTER FIRST NUMBER

?

13

ENTER SECOND NUMBER

?

15.8

THE SUM OF 13 AND 15.8 IS 28.8

Line 100 causes the answer to be output in a very clear way. The sections between inverted commas are printed exactly as they appear, whereas the words NUMBER1, NUMBER2 and SUM refer to numbers which are recalled from the computer's store and printed.

## INPUT message

We have decided that it is best to accompany requests for input by printing an appropriate message on the screen before the INPUT statement. We can however include the message in the input statement itself,

e.g. INPUT "TYPE IN A NUMBER": N

The message TYPE IN A NUMBER will appear on the screen and the computer will wait for us to enter a number on the same line. Let us re-write Program 10 using this form of input.

```
0010 // PROGRAM 11
0020 //
0030 // COMELY KATE
0040 //
0050 // TO READ IN TWO NUMBERS,ADD THEM AND OUTPUT THE ANSWER
0060 //
0070 PRINT
0080 INPUT "ENTER THE FIRST NUMBER ": NUMBER1
0090 PRINT
0100 INPUT "ENTER THE SECOND NUMBER ": NUMBER2
0110 SUM:=NUMBER1+NUMBER2
0120 PRINT
0130 PRINT "THE SUM OF ",NUMBER1," AND ",NUMBER2," IS ",SUM
0140 END
```

RUN

ENTER THE FIRST NUMBER 13

ENTER THE SECOND NUMBER 15.8

THE SUM OF 13 AND 15.8 IS 28.8

This time we begin with six lines of comment. Lines 20, 40 and 60 are actually empty comments used merely to separate the other comments and make them easier to read.

Take careful note of the INPUT instructions – lines 80 and 100. We will mostly use this form of input from now on.

We have included one extra PRINT instruction on line 70.

It is obviously just as easy to write a program to multiply two numbers and Program 12 does just this

```
0010 // PROGRAM 12
0020 //
0030 // COMELY KATE
0040 //
0050 // TO READ IN TWO NUMBERS,MULTIPLY THEM AND OUTPUT THE ANSWER
0060 //
0070 PRINT
0080 INPUT "ENTER THE FIRST NUMBER ": NUMBER1
0090 PRINT
0100 INPUT "ENTER THE SECOND NUMBER ": NUMBER2
0110 PRODUCT:=NUMBER1*NUMBER2
0120 PRINT
0130 PRINT "THE PRODUCT OF ",NUMBER1," AND ",NUMBER2," IS ",PRODUCT
0140 END
```

RUN

ENTER THE FIRST NUMBER 13

ENTER THE SECOND NUMBER 19

THE PRODUCT OF 13 AND 19 IS 247

So far we have performed just one operation in each program. We can of course have as many operations as we like in an instruction. Suppose we have five numbers A, B, C, D, E and wish to calculate  $A + B - C + D - E$ . Program 13 performs this mighty task.

When you are running the program you should separate the values being input by commas.

```
0010 // PROGRAM 13
0020 //
0030 // COMELY KATE
0040 //
0050 // TO CALCULATE AN ARITHMETIC EXPRESSION
0060 //
0070 PRINT
0080 INPUT "ENTER 5 NUMBERS  ": A, B, C, D, E
0090 ANSWER:=A+B-C+D-E
0100 PRINT
0110 PRINT "THE VALUE OF ";A;" + ";B;" - ";C;" + ";D;" - ";E;" =""
0120 PRINT ANSWER
0130 END

RUN

ENTER 5 NUMBERS  8,3 6,9,12
THE VALUE OF 8  + 3  - 6  + 9  - 12  =
2
```

## Priority among Arithmetic Operators

In finding the value of an expression such as  $47 + 51 - 38$  it does not really matter in which order the operations are performed. We get the same answer whether we add 47 and 51 first and then subtract 38, or subtract 38 from 51 first and then add the answer to 47.

But consider  $4 + 6 * 2$ .

What is the correct answer here? Is it 20, got by adding 4 and 6 and then multiplying by 2; or is it 16, got by multiplying 6 by 2 and then adding the answer to 4? In other words, which operation should be done first, addition or multiplication?

In order to determine the precedence of arithmetic operations we use the following rule:

*Multiplication and division are done  
before addition and subtraction*

i.e. multiplication and division take precedence over addition and subtraction.

Multiplication and division are of equal priority, as are addition and subtraction.  
But what about multiplication followed by division (or vice versa)? For example does

$$A/B * C \text{ mean } \frac{A * C}{B} \text{ or } \frac{A}{B * C}?$$

In fact it means

$$\frac{A * C}{B}$$

The rule is:

*When operators of equal precedence follow one another,  
the one on the left is done first.*

Now what about brackets? The important point is that brackets can be used to alter the natural precedence. Expressions inside brackets are evaluated first, e.g.

$$\begin{aligned} A * B + C &\text{ means } ab + c \\ A * (B + C) &\text{ means } a(b + c) \end{aligned}$$

The integer operators DIV and MOD obey the above rules, having the same precedence as \* and /,

e.g.  $4 + 8 \text{ DIV } 5 = 4 + 1 = 5$   
 $2 + 13 \text{ MOD } 8 = 2 + 5 = 7$

Let us write a little program to test whether COMAL abides by the above rules in performing its arithmetic. We will simply calculate the value of the expression

$$a + b \left( c + \frac{d e}{c} \right)$$

```
0010 // PROGRAM 14
0020 //
0030 // COMELY KATE
0040 //
0050 // TO CALCULATE AN ARITHMETIC EXPRESSION
0060 //
0070 PRINT
0080 INPUT "ENTER 5 NUMBERS ":" A, B, C, D, E"
0090 ANSWER:=A+B*(C+D+E/C)
0100 PRINT
0110 PRINT "THE VALUE OF THE EXPRESSION A+B(C+DE/C) IS ",ANSWER
0120 END

RUN
```

ENTER 5 NUMBERS 8,3,6,9,12

THE VALUE OF THE EXPRESSION A+B(C+DE/C) IS 80

Are the results as you expected? Test them yourself.

## Four in One

Let us write a program which accepts two numbers and performs all four arithmetic operations on them.

```
0010 // PROGRAM 15
0020 //
0030 // COMELY KATE
0040 //
0050 // TO ADD,SUBTRACT,MULTIPLY AND DIVIDE TWO NUMBERS
0060 //
0070 PRINT
0080 INPUT "ENTER THE FIRST NUMBER ": NUMBER1
0090 PRINT
0100 INPUT "ENTER THE SECOND NUMBER ": NUMBER2
0110 //
0120 // CALCULATE AND PRINT SUM
0130 //
0140 SUM:=NUMBER1+NUMBER2
0150 PRINT
0160 PRINT "THE SUM OF ",NUMBER1," AND ",NUMBER2," IS ",SUM
0170 //
0180 // CALCULATE AND PRINT DIFFERENCE
0190 //
0200 DIFFERENCE:=NUMBER1-NUMBER2
0210 PRINT
0220 PRINT "THE DIFFERENCE BETWEEN ",NUMBER1," AND ",NUMBER2," IS ",DIFFERENCE
0230 //
0240 // CALCULATE AND PRINT PRODUCT
0250 //
0260 PRODUCT:=NUMBER1*NUMBER2
0270 PRINT
0280 PRINT "THE PRODUCT OF ",NUMBER1," AND ",NUMBER2," IS ",PRODUCT
0290 //
0300 // CALCULATE AND PRINT QUOTIENT
0310 //
0320 QUOTIENT:=NUMBER1/NUMBER2
0330 PRINT
0340 PRINT "THE QUOTIENT OF ",NUMBER1," AND ",NUMBER2," IS ",QUOTIENT
0350 END
RUN
```

ENTER THE FIRST NUMBER 15

ENTER THE SECOND NUMBER 8

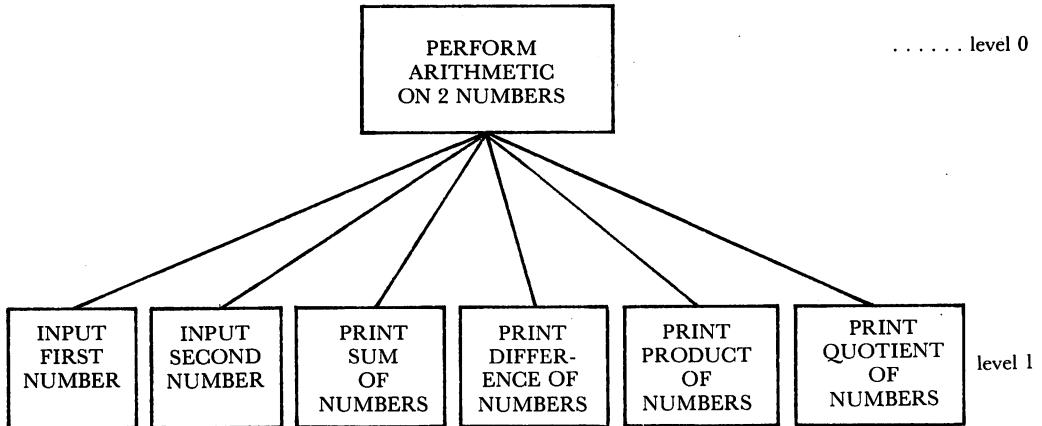
THE SUM OF 15 AND 8 IS 23

THE DIFFERENCE BETWEEN 15 AND 8 IS 7

THE PRODUCT OF 15 AND 8 IS 120

THE QUOTIENT OF 15 AND 8 IS 1.875

The above program is fairly lengthy and it may be a help to view the overall plan of the program by means of a diagram.



The top box (on level 0) represents the objective of the program. The main task breaks down into a sequence of six subtasks which are presented on level 1 of the diagram. The tasks are performed in sequence from left to right.

The diagram does not represent every single detail of the program: rather is it intended to display the **structure** of the solution. It is called a **structure diagram**. In future we will often find it convenient to draw such diagrams before we develop our programs. They will help us to clarify our thoughts and spell out the steps of the solutions to our problems.

## **Stopping and Starting**

It is often useful to be able to stop program execution temporarily and then to continue later. This may easily be done by inserting the instruction STOP into the program at suitable points. For example, suppose we wanted to halt Program 15 after each result is printed, we would simply insert the instruction STOP at lines 165, 225 and 285 say. When the program has halted we can get it going again by typing **CON**.

## **Saving your programs on disk**

When you type in your program at the keyboard it is stored in the memory of the computer. When you switch off the machine your program will be erased from the

memory. This is rather unfortunate

- (a) because you may not have finished working with the program and would like to return to it later
- (b) because you would like a permanent record of your honest endeavours.

Attached to your computer you will probably find a box called a **disk drive**. This will have a little door on the front through which you may place a flat disk of magnetic material on which you may record your programs. Your teacher will show you how to use the disk drive and how to prepare disks for recording programs.



In order to save a copy of your programs on disk all you have to do is issue the

## **SAVE**

command to the computer. Suppose for example you have typed in the last program (Program 15) and now want to save it on disk. You should perform the following actions:

- (1) Make sure you have a properly prepared disk in the disk drive.
- (2) Close the door of the disk drive.
- (3) Think of a name for your program e.g. ARITH, type in the command SAVE ARITH and press RETURN. You may give the program any name you wish. Most systems however will not accept any more than about eight letters for the name. If you use too many characters the last characters will be removed.
- (4) Notice that the light on the disk drive will come on and you will hear a whirring noise for a little while.
- (5) Type in the command **CAT** and you will see a list of the programs which are stored on the disk appearing on the screen. CAT is short for CATALOGUE. The program ARITH should appear in the catalogue.

You may now switch off the machine, safe in the knowledge that your program is recorded on disk and can be brought back into the main memory of the computer whenever you wish.

## **Retrieving your program from disk**

In order to bring a program from disk back into the main memory of your computer you use the

## **LOAD**

command. For example if you want to recover the program ARITH which you saved before, you simply type LOAD ARITH and press RETURN. The light on the disk drive will come on, a whirring noise will be heard and in a short while a copy of your program will be in main memory, ready to be listed or run or whatever you wish.

## **Dual Disk Drive**

You may find that you have two disk drives attached to your machine. In this case one of the drives is referred to as DK0: and the other as DK1:. You may refer explicitly to a particular drive when saving or loading a program,

e.g. SAVE "DK1: ARITH" would cause a copy of the program in the computer's memory to be saved under the name ARITH on a disk in disk drive DK1:

Usually DK0: is the default disk drive, so that in the absence of any specific reference it is DK0: which is automatically used for saving and loading.

e.g. SAVE ARITH is usually taken to mean SAVE "DK0: ARITH".

**N.B.** Saving a program on disk does not erase that program from the computer's memory. A *copy* of the program is saved on the disk. The program itself still resides in the memory and can be run or listed or changed as you wish.

Similarly when a program is loaded into the memory from a disk, a copy still remains intact on the disk. A program may be erased from a disk in one of the following ways:

- (a) Using the **DELETE** command,  
e.g. DELETE "DK1: ARITH.CSB"

The .CSB part of the name is put on by the computer system itself and must be used in the DELETE command. You do *not* use it when saving a program.

(b) Saving a new program with the same name as one already on the disk. The new program replaces the old one.

(c) Accidental corruption of the disk, e.g. by exposure to a magnetic field.

A program may be erased from memory in any of the following ways:

- (a) Switching off the machine
- (b) Typing the command NEW
- (c) Loading a new program into the memory from disk

(d) Deleting the program using the **DEL** command, e.g. DEL 10, 100 deletes all lines of the program in memory from 10 to 100 inclusive. Note that, of course, this does not effect the copy of the program on disk if it has been saved. Note too the difference between the DEL and the DELETE commands:

DEL deletes program lines from memory.

DELETE deletes programs from disk.

## Being concise



You are recommended to write your Programs in the style of Program 15, i.e.

- (a) Include comments to explain the action of the program.
- (b) Perform one step at a time.
- (c) Include meaningful messages with the output statements.

However it is sometimes desirable to write more concise programs. Program 15 could be shortened in several ways. Let us re-write it to illustrate this. Since you already know the program we will omit the opening comments, although this is not recommended as a way of being concise!

```

0010 INPUT "ENTER TWO NUMBERS ": N1, N2
0020 PRINT
0030 PRINT "THE FOLLOWING NUMBERS REPRESENT THE SUM,DIFFERENCE,PRODUCT AND
    QUOTIENT OF ",N1," AND ",N2
0040 PRINT
0050 PRINT N1+N2
0060 PRINT N1-N2
0070 PRINT N1*N2
0080 PRINT N1/N2
0090 END
RUN
ENTER TWO NUMBERS 15,8
THE FOLLOWING NUMBERS REPRESENT THE SUM,DIFFERENCE,PRODUCT AND QUOTIENT OF 15
AND 8
23
7
120
1.875

```

Notice that we can actually include some arithmetic in our PRINT statements. For example PRINT N1 + N2 will cause the number stored in N1 to be added to the number stored in N2 and the result printed immediately. Does this surprise you? Well, look back at our immediate mode examples of arithmetic at the beginning of this chapter.

If we wanted all four results printed on one line we could write

```

50 PRINT N1 + N2;
60 PRINT N1 - N2;
70 PRINT N1 * N2;
80 PRINT N1/N2

```

A semicolon or a comma at the end of a PRINT line causes the output mechanism to 'wait' on the same line for the next item to be printed.

We could, of course, achieve the same effect more economically by replacing lines 50 to 80 by just one line — PRINT N1 + N2; N1 - N2; N1 \* N2; N1/N2.

## A Simple Problem

Let us write a program to perform the following task: Convert a length given in yards, feet and inches to metres.

To begin with, we remind ourselves that we cannot just give the computer the figures for yards, feet and inches and ask it to go ahead and do the conversion. We must tell the computer exactly what to do. Once we have worked out the exact sequence of instructions, we can leave it to the computer to get on with the actual calculations. We can, in fact, solve the problem in a number of ways. The following is one possible method.

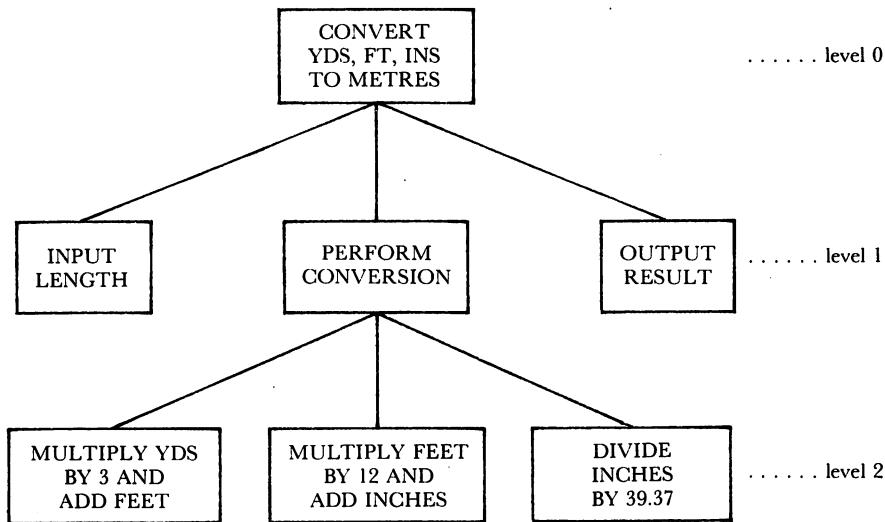
## Outline Solution

- (1) Input the number of yards, feet, inches.
- (2) Convert the number of yards to feet and add in the number of feet input at (1). This gives us the total number of feet.
- (3) Convert the total number of feet to inches and add in the number of inches input at (1) to get the total number of inches.
- (4) Convert the total number of inches to metres.
- (5) Print the answer.

What do we need to know in order to do all this? We need to know

- (a) the length to be converted
- (b) the number of feet in a yard, i.e. 3
- (c) the number of inches in a foot, i.e. 12
- (d) the number of inches in a metre, i.e. 39.37, or the number of metres in an inch, i.e. .0254

Let us now represent the solution in graphical form:



Our structure diagram contains two levels of refinement in this case. The first level spells out the main sequence of tasks to be done. The first and third of these tasks INPUT LENGTH and OUTPUT RESULT are simple and straightforward and need no further attention. The other task PERFORM CONVERSION does require further spelling out or refinement and this brings us to the second level. What the diagram tells us then is:

First input the length to be converted. Then perform the three tasks involved in doing the conversion. Finally output the result.

The program follows quite simply from the diagram

```

0010 // PROGRAM 17
0020 //
0030 // COMELY KATE
0040 //
0050 // TO CONVERT YARDS,FEET,INCHES TO METRES
0060 //
0070 PRINT
0080 INPUT "ENTER THE LENGTH - YARDS FIRST,THEN FEET,THEN INCHES "; YARDS, FEET,
    INCHES
0090 TOTALFEET:=3*YARDS+FEET
0100 TOTALINCHES:=12*TOTALFEET+INCHES
0110 METRES:=TOTALINCHES/39.37
0120 PRINT
0130 PRINT YARDS," YARDS ",FEET," FEET ",INCHES," INCHES = ",METRES," METRES "
0140 END
RUN
ENTER THE LENGTH - YARDS FIRST,THEN FEET,THEN INCHES 4,2,8
4 YARDS 2 FEET 8 INCHES = 4.470409 METRES
RUN
ENTER THE LENGTH - YARDS FIRST,THEN FEET,THEN INCHES 1,0,0
1 YARDS 0 FEET 0 INCHES = 0.9144018 METRES

```

The program follows closely from the diagram and you can see from the sample runs how the results turn out. Line 130 deserves some comment. Numeric values and strings are interspersed to give a readable, meaningful output. Note too that in line 80 the commas are part of the string enclosed in quote marks. They are not therefore used as in the usual manner to space the output. Instead they are actually printed as shown.

## Cutting decimals down to size

You are probably thinking that the results of the last program are somewhat ungainly! After all 4.470409 metres is a bit of an eyeful. So, our problem is – can we reduce the number of decimal places printed? In particular, could we express the above answer correct to two decimal places so that we could have metres and centimetres. For example we would like to write 4.470409 as 4.47.

To solve our problem we make use of a special operation or function provided by COMAL, namely **ROUND**. What does this operation do?

**ROUND** gives us a decimal number rounded off to the nearest integer,

e.g. **ROUND** (35.3) = 35

**ROUND** (35.8) = 36, etc.

Now consider 17.83721732, to be rounded to 2 decimal places. It is no use applying **ROUND** immediately because we would just get 18 instead of 17.84. But consider the following sequence of operations:

$$17.83721732 * 100 = 1783.721732$$

$$\text{ROUND}(1783.721732) = 1784$$

$$1784/100 = 17.84$$

Similarly  $1.63217843 * 100 = 163.217843$   
 ROUND (163.217843) = 163  
 $163/100 = 1.63$   
 Correct again!

You should try the strategy yourself on different numbers and check that it works every time.

A program to implement this strategy is absolutely straightforward.

```

0010 // PROGRAM 18
0020 //
0030 // COMELY KATE
0040 //
0050 // TO ROUND A NUMBER TO 2 DECIMAL PLACES
0060 //
0070 PRINT
0080 INPUT "INPUT A VALUE "; VALUE
0090 ROUNDEDVALUE:=ROUND(VALUE*100)/100
0095 PRINT
0100 PRINT VALUE," ROUNDED TO TWO DECIMAL PLACES IS ",ROUNDEDVALUE
0130 END

```

RUN

```

INPUT A VALUE 37.84793
37.84793 ROUNDED TO TWO DECIMAL PLACES IS 37.85

```

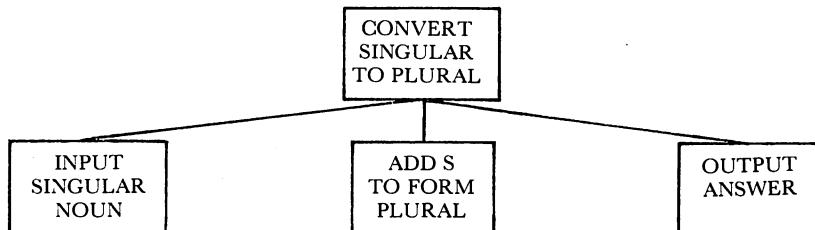
## Watch your grammar!

Naturally we cannot expect to perform arithmetic with strings – after all, what would “HELLO” divided by “GOODBYE” mean?!

However, one arithmetic operation does make sense with strings, namely the operation of addition. What would you expect to happen when two strings are added together,

e.g. what does “HELLO” + “GOODBYE” produce?  
 Not surprisingly, “HELLO” + “GOODBYE” = “HELLOGOODBYE”. Thus addition causes concatenation (i.e. chaining or linking together) of strings.

Let us write a little program which uses the addition operator to convert singular nouns to plural form.



```
0010 // PROGRAM 19
0020 //
0030 // COMELY KATE
0040 //
0050 // TO CONVERT A SINGULAR NOUN TO PLURAL FORM
0060 //
0070 DIM SINGULAR$ OF 20, PLURAL$ OF 20
0080 PRINT "ENTER A SINGULAR NOUN"
0090 INPUT SINGULAR$
0100 PLURAL$:=SINGULAR$+"S"
0110 PRINT
0120 PRINT "THE SINGULAR IS ",SINGULAR$
0130 PRINT
0140 PRINT "THE PLURAL IS ",PLURAL$
0150 END
```

RUN

ENTER A SINGULAR NOUN

?

BELL

THE SINGULAR IS BELL

THE PLURAL IS BELLS

RUN

ENTER A SINGULAR NOUN

?

CHILD

THE SINGULAR IS CHILD

THE PLURAL IS CHILDS

So, that clever shining computer is not so clever after all!!

Remember a computer will only do what it is told to do. It will do exactly what you tell it and it will do it at very high speed. But it has no imagination and no knowledge of its own. It is the programmer's responsibility to supply all the information and to give sufficiently exact instructions to get the job done properly.

If you look back at our program you will see that one of the instructions was to add the letter S to the word which was input (see line 100). This was intended to convert the singular form of a word to its plural form. The computer did exactly as it was told; it was not to know that merely adding S does not always work! Later we will see how we might improve this program.

## Summary

The arithmetic operators are:

+ add  
- subtract  
\* multiply  
/ divide

DIV integer divide

MOD get remainder after integer divide

:= means 'is assigned the value'

Comments beginning with // should be included in all programs.

Messages can be included in the INPUT statement.

The ordinary arithmetic precedence exists among the arithmetic operators.  
Precedence can be altered by the use of brackets.

Programs may be saved on disk and loaded from disk into main memory..

+ causes concatenation of strings.

### COMAL KEYWORDS

AUTO	CAT	CLEAR	CON	DATA	DEL
DELETE	DIV	EDIT	END	INPUT	LIST
LOAD	MOD	NEW	PRINT	READ	RENUM
ROUND	RUN	SAVE	STOP		

## QUESTIONS and EXERCISES

Before writing each program draw a structure diagram.

1. Why are magnetic disks used? What instruction would you give to store a program
  - (a) on the default disk
  - (b) on the other disk (i.e. a disk in the other drive)?
2. Perform each of the following calculations by hand and then check them on the computer:

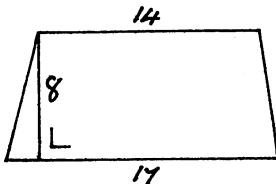
(i) $4 + 7 - 3 - 12 + 9$	(iv) $4 * (18 + 21 / 7)$
(ii) $3 * 4 + 2 * 6$	(v) $20 + 10 * (18 + 4 * (90 + 80 / 16 * 12))$
(iii) $6 * 4 / 3$	

3. Explain what the operations DIV and MOD do. Evaluate the following expressions:
  - (i) 16 DIV 3
  - (ii) 16 DIV 4
  - (iii) 89 MOD 10
  - (iv) 89 MOD 13
  - (v) 90 DIV 5 MOD 3
  - (vi) (42 MOD 11) DIV (15 MOD 4)
4. What is wrong with each of the following COMAL statements?
  - (i) IN A, B, C
  - (ii) OUT A, B
  - (iii) PRINT "THE ANSWER IS" A + B
  - (iv) PRINT 4A
  - (v) A - B := NUMBER
  - (vi) A := B ÷ C
  - (vii) JOE := BILL \* TOM
5. What would the output of each of the following programs be?
  - (i)
 

```
10 NUMBER := 84.72
20 PRINT NUMBER
30 PRINT "NUMBER"
40 END
```
  - (ii)
 

```
10 JOE := 48
20 BILL := 13
30 PRINT JOE MOD BILL DIV BILL
40 END
```
6. Write COMAL programs to perform each of the following tasks:
  - (a) Read 4 numbers and find their sum.
  - (b) Read 4 numbers and find their mean.
  - (c) Find the area of a square of side 13.67 m.
  - (d) Find the area of a circle of radius 14.3 m.
  - (e) Find the volume of a rectangular tank with dimensions 3.8 m, 2.9 m and 1.26 m.
  - (f) Find the weekly cost of coffee in a canteen where 100 people have two cups of coffee each, on each of 5 days. One tin of coffee makes 480 cups and costs £10.89.
7. Write a program which accepts a verb and forms its past tense by adding ED. Does this cater for all verbs?
8. In addition to the arithmetic operators already listed, COMAL makes use of an exponentiation operator  $\wedge$  (above the N on the keyboard). Thus e.g.  $2\wedge 3 = 8$ ,  $3\wedge 5 = 243$ , etc. Explore the action of this operator and determine its precedence with respect to the other operators.
9. Find the simple interest on £12,300 for 9 years at  $15\frac{3}{4}\%$  per annum.

10. Find the compound interest on £12,300 for 9 years at  $15\frac{3}{4}\%$  per annum.
11. Find the average speed at which a motorist travels if a journey from Killarney to Dublin (190 miles) takes him 5 hours.
12. Which of the following are right-angled triangles?
  - (i) 20, 52, 48
  - (ii) 29.6, 73.4, 78.2
  - (iii) .48, .64, .8
13. Find the area of the trapezium shown in the diagram.



14. Mr. Murphy purchased the following items in his local hardware store:  
8 rolls of wallpaper at £3.79 per roll  
1 pasting brush £1.80  
2.5 litres of paint at £3.76 per litre  
12 metres of wood at £1.24 per metre  
2 packets paste at £0.67 per packet.  
His habit of promptly paying his bills earned him 6.5% discount. What did he have to pay in total?
15. If 18% VAT was charged on the bill in the previous question how much did Mr. Murphy have to pay?
16. Billy Bones lives in Wicklow but works in Dublin. He works a 5-day week and has 4 weeks holidays in a year. He travels to work by car. How many miles does he travel each year?
17. Convert time as specified by a 24-hour clock to time as specified by a 12-hour clock.
18. Convert
  - (a) pounds and ounces to kilograms
  - (b) gallons to litres
  - (c) radians to degrees.
19. Gather some data on the running costs of a typical 10 H.P. car and write a program to estimate the yearly cost.

# 3 Selection

*Two roads diverged in a yellow wood . . .*

## If only . . .

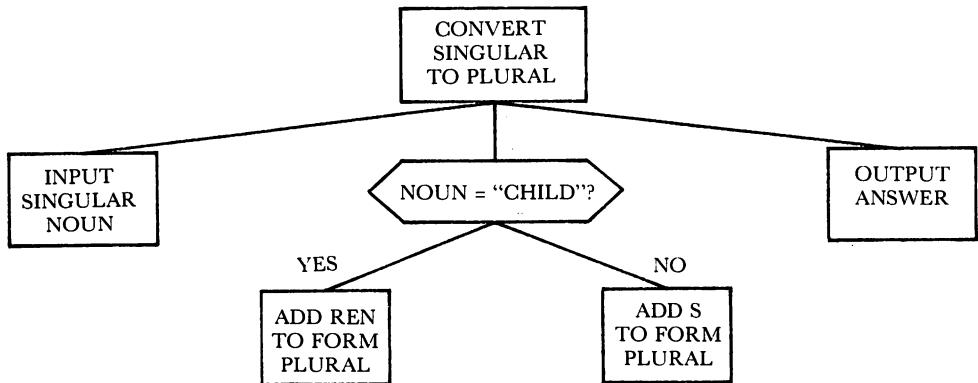
Look back at Program 19. It would be nice to get that plural for CHILD correct. What we must do is instruct the computer not to add S in this case (because it gives the wrong answer, CHILDS), but to do something else instead, namely, add REN to give the correct answer, CHILDREN.

How can we do this?

The answer lies in the ability to make a selection between alternative courses of action. So far all our programs have involved a simple sequential structure; we have been concerned solely with getting the *sequence* of instructions correct. We must now turn our attention to a second important programming structure, namely *selection*.

## IF . . . THEN . . . ELSE

Our solution to the 'CHILDS' problem uses a choice or selection and is embodied in the following diagram.



We use a new shape to represent the selection process. In the diagram a choice is made depending on whether the singular noun is CHILD or not. If it is, we choose the action at the end of the YES branch emerging from the selection box. If

not, we take the action at the end of the NO branch. Note carefully that only one of the two actions is done. This contrasts with the usual sequence of events on any given level of a diagram where all the tasks on a level are done in sequence, working from left to right.

To make a selection in a program we use the **IF ... THEN ... ELSE** structure in COMAL. Consider the following improved version of Program 19.

```
0010 // PROGRAM 20
0020 //
0030 // COMELY KATE
0040 //
0050 // TO CONVERT A SINGULAR NOUN TO PLURAL FORM
0060 //
0070 DIM SINGULAR$ OF 20, PLURAL$ OF 20
0080 PRINT "ENTER A SINGULAR NOUN"
0090 INPUT SINGULAR$
0100 IF SINGULAR$="CHILD" THEN
0110   PLURAL$:=SINGULAR$+"REN"
0120 ELSE
0130   PLURAL$:=SINGULAR$+"S"
0140 ENDIF
0150 PRINT
0160 PRINT "THE SINGULAR IS ",SINGULAR$
0170 PRINT
0180 PRINT "THE PLURAL IS ",PLURAL$
0190 END
```

RUN

```
ENTER A SINGULAR NOUN
?
```

THE SINGULAR IS BELL

THE PLURAL IS BELLS

RUN

```
ENTER A SINGULAR NOUN
?
```

THE SINGULAR IS CHILD

THE PLURAL IS CHILDREN

The part of the program to concentrate on is lines 100 to 140. If the string held in the variable SINGULAR\$ is equal to "CHILD" then line 110 is executed, i.e. the variable PLURAL\$ is assigned the value "CHILDREN" by the addition of "REN" to SINGULAR\$. The program then ignores lines 120 and 130, acknowledges that line 140 closes the IF construction and proceeds with line 150.

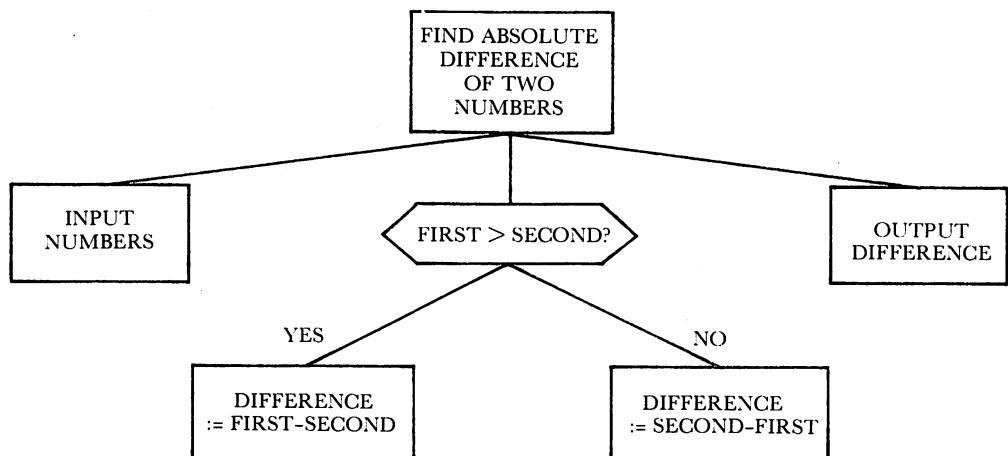
If, on the other hand, SINGULAR\$ is not equal to "CHILD" line 110 will be skipped and line 130 which is prefaced by the ELSE statement on line 120, is executed.

The little block of program starting with IF on line 100 and closing with **ENDIF** on line 140 neatly expresses the choice of one action or another, depending on whether the noun read in is CHILD or not. Notice that the statements on lines 110 and 130 are slightly indented from the left. This is done by the COMAL system itself and serves to emphasise the structure embodied in the IF ... THEN ... ELSE selection. We shall see more of this indentation as we go along.

In order to become practised in the use of this important selection structure let us take a few more examples.

## Absolutely Different!

**Problem:** Accept two numbers from the keyboard and find their absolute difference, i.e. subtract the smaller from the larger.



Naturally the computer does not know which way round to do the subtraction (it's not as clever as you, you must remember). That is why we have our choice box FIRST>SECOND and select the action according to whether the answer is YES or NO. The following COMAL program uses the IF ... THEN ... ELSE construction to make the choice.

```

0010 // PROGRAM 21
0020 //
0030 // COMELY KATE
0040 //
0050 // TO SUBTRACT A SMALLER NUMBER FROM A BIGGER NUMBER
0060 //
0070 PRINT
0080 INPUT "ENTER TWO NUMBERS "; FIRST, SECOND
0090 PRINT
0100 IF FIRST>SECOND THEN
  
```

```

0110 DIFFERENCE:=FIRST-SECOND
0120 ELSE
0130 DIFFERENCE:=SECOND-FIRST
0140 ENDIF
0150 PRINT "THE POSITIVE DIFFERENCE BETWEEN ",FIRST," AND ",SECOND," IS ",
        DIFFERENCE
0160 END

```

RUN

ENTER TWO NUMBERS 57,42

THE POSITIVE DIFFERENCE BETWEEN 57 AND 42 IS 15

RUN

ENTER TWO NUMBERS 42,57

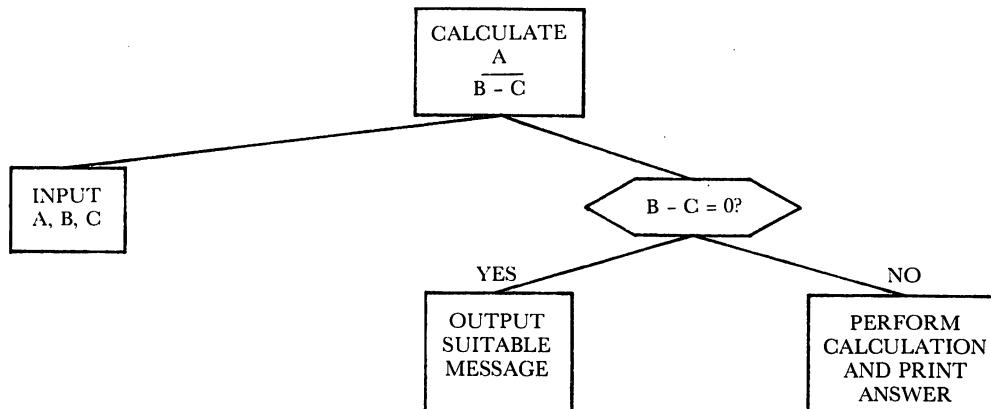
THE POSITIVE DIFFERENCE BETWEEN 42 AND 57 IS 15

## Dividing the Spoils (without spoiling the divide!)

**Problem:** Given three numbers  $a$ ,  $b$ ,  $c$ , calculate

$$\frac{a}{b - c}$$

The important point here is that we must not try to divide by zero. Therefore we must check that  $b - c$  is not zero before we attempt to divide. This is something we do not usually bother about when we are solving simple arithmetic problems by hand, since we can see as we go along whether the divisor is zero or not. In writing a program however we must consider all eventualities beforehand and write our instructions accordingly. In this case we must allow for the possibility that  $b - c$  might be zero. We therefore include a choice of path in the program, i.e. if  $b - c$  is zero print out a message to that effect, otherwise divide and print the answer.



```

0010 // PROGRAM 22
0020 //
0030 // COMELY KATE
0040 //
0050 // TO CALCULATE A/(B-C)
0060 //
0070 PRINT
0080 INPUT "VALUES OF A,B,C ? ": A, B, C
0090 PRINT
0100 IF B-C=0 THEN
0110   PRINT "DIVISOR IS ZERO, SO WE CANNOT DIVIDE"
0120 ELSE
0130   PRINT "VALUES OF A,B,C ARE ",A;B;C
0140   PRINT
0150   PRINT "VALUE OF A/(B-C) IS ",A/(B-C)
0160 ENDIF
0170 END

```

RUN

VALUES OF A,B,C ? 12,9,4

VALUES OF A,B,C ARE 12 9 4

VALUE OF A/(B-C) IS 2.4

RUN

VALUES OF A,B,C ? 12,5,5

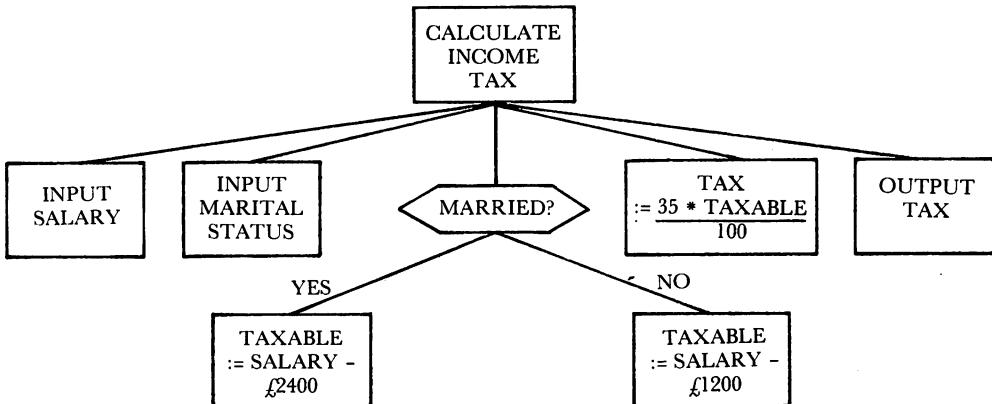
DIVISOR IS ZERO, SO WE CANNOT DIVIDE

The program is straightforward. You should notice however that we have one statement following IF and three statements following ELSE, in this case. These statements are indented to emphasise the structure. We can have as many statements as we require in each section.



### Pounds of Flesh

**Problem:** Calculate the income tax payable on a given salary, if the only allowance available is (a) £1200 for a single person, (b) £2400 for a married person and the rate of tax is always 35p in the £.



```

10 // PROGRAM 23
0020 //
0030 // COMELY KATE
0040 //
0050 // TO CALCULATE INCOME TAX
0060 //
0070 DIM STATUS$ OF 7
0080 PRINT
0090 INPUT "ENTER YOUR SALARY ": SALARY
0100 PRINT
0110 INPUT "ARE YOU MARRIED OR SINGLE ": STATUS$
0120 PRINT
0130 IF STATUS$="MARRIED" THEN
0140   TAXABLE:=SALARY-2400
0150 ELSE
0160   TAXABLE:=SALARY-1200
0170 ENDIF
0180 TAX:=35*TAXABLE/100
0190 PRINT "YOU MUST PAY $",TAX," ON A SALARY OF $",SALARY
0200 END

```

RUN

ENTER YOUR SALARY 8000

ARE YOU MARRIED OR SINGLE SINGLE

YOU MUST PAY \$2380 ON A SALARY OF \$8000

RUN

ENTER YOUR SALARY 10000

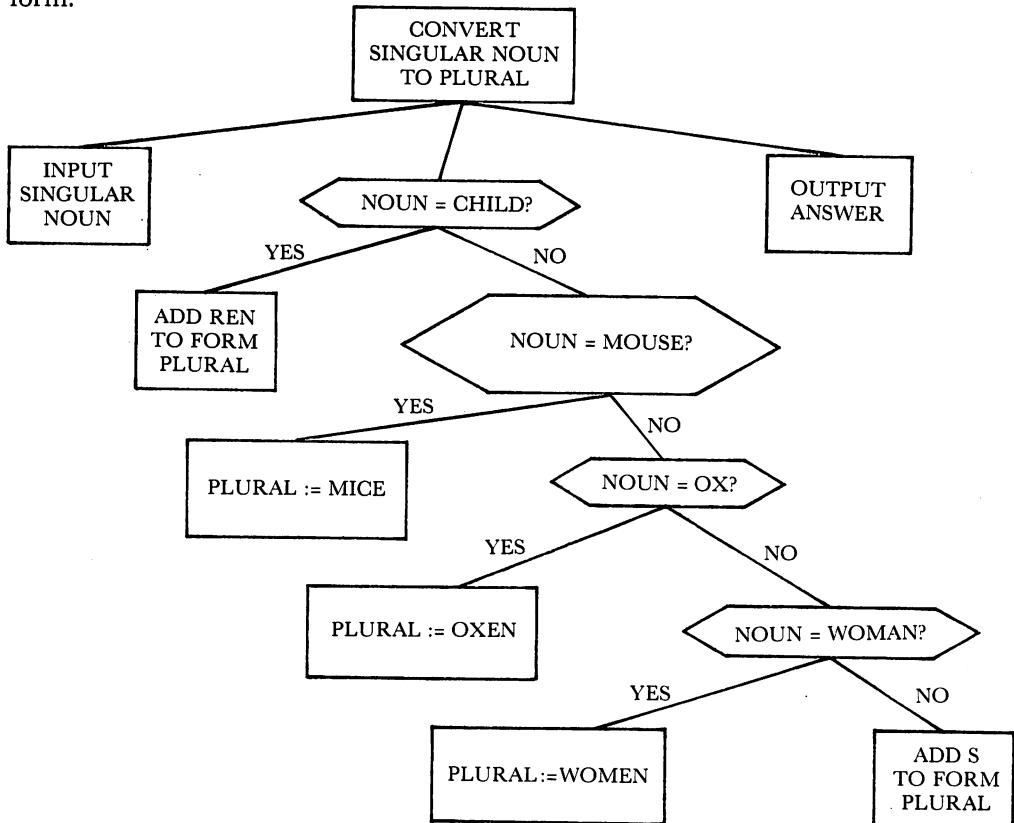
ARE YOU MARRIED OR SINGLE MARRIED

YOU MUST PAY \$2660 ON A SALARY OF \$10000

## Decisions! Decisions!

Let us return to our problem of converting singular nouns to plural form. No doubt you have said to yourself that CHILD is not the only singular noun whose plural is not formed by simply adding S. What about MOUSE, or OX, or WOMAN, you say! Yes, it would be better if we took account of these exceptions and made a further improvement on the solution expressed in Program 20.

This time we will have to make a choice among several possibilities. We will have to use the IF statement several times. First let us present the solution in diagrammatic form.



We have a veritable cascade of choices in the diagram. The questioning stops at the first YES. Consider, for example, what happens if the noun which is input is OX. We take the NO branch from the NOUN = CHILD? box, then the NO branch from the NOUN = MOUSE? box and finally the YES branch from the NOUN = OX? box. The plural form is given the value OXEN and then we go back to the first level to output the answer.

Note how deeply we may have to go before accomplishing a task at the first level -

in this case answering the question as to what kind of singular noun we have. But no matter how deeply we go we must return to the first level to perform the remaining task or tasks at that level.

Our COMAL program follows the logic of the diagram very closely.

```
0010 // PROGRAM 24
0020 //
0030 // COMELY KATE
0040 //
0050 // TO CONVERT A SINGULAR NOUN TO PLURAL FORM
0060 //
0070 DIM SINGULAR$ OF 20, PLURAL$ OF 20
0080 PRINT "ENTER A SINGULAR NOUN"
0090 INPUT SINGULAR$
0100 PRINT
0110 IF SINGULAR$=="CHILD" THEN
0120   PLURAL$:="CHILDREN"
0130 ELSE
0140   IF SINGULAR$=="MOUSE" THEN
0150     PLURAL$:="MICE"
0160   ELSE
0170     IF SINGULAR$=="OX" THEN
0180       PLURAL$:="OXEN"
0190     ELSE
0200       IF SINGULAR$=="WOMAN" THEN
0210         PLURAL$:="WOMEN"
0220       ELSE
0230         PLURAL$:=SINGULAR$+"S"
0240       ENDIF
0250     ENDIF
0260   ENDIF
0270 ENDIF
0280 PRINT "THE SINGULAR IS ",SINGULAR$
0290 PRINT "THE PLURAL IS ",PLURAL$
0300 END
```

RUN

```
ENTER A SINGULAR NOUN
?
OX
```

```
THE SINGULAR IS OX
THE PLURAL IS OXEN
```

RUN

```
ENTER A SINGULAR NOUN
?
CHILD
```

```
THE SINGULAR IS CHILD
THE PLURAL IS CHILDREN
```

RUN

```
ENTER A SINGULAR NOUN
?
BELL
```

```
THE SINGULAR IS BELL
THE PLURAL IS BELLS
```

The notable feature of the program is the sequence of IF ... THEN ... ELSE's. There are four IF's and a corresponding four ENDIF's closing off each IF. Brackets have been placed to show you which ENDIF goes with which IF.

Note that only one of the IF conditions will be satisfied and therefore only one of the statements obeyed, namely the one immediately following the satisfied condition. Once a condition is satisfied and the corresponding statement executed the program skips down to the appropriate ENDIF and then goes on through the rest of the text.

Because the need for multiple selection arises fairly often COMAL provides a shorter version of ELSE IF, namely **ELIF**. This enables us to write our programs more compactly as we see from the following version of our previous example.

```
0010 // PROGRAM 25
0020 //
0030 // COMELY KATE
0040 //
0050 // TO CONVERT A SINGULAR NOUN TO PLURAL FORM
0060 //
0070 DIM SINGULAR$ OF 20, PLURAL$ OF 20
0080 PRINT "ENTER A SINGULAR NOUN"
0090 INPUT SINGULAR$
0100 PRINT
0110 IF SINGULAR$=="CHILD" THEN
0120   PLURAL$:="CHILDREN"
0130 ELIF SINGULAR$=="MOUSE" THEN
0140   PLURAL$:="MICE"
0150 ELIF SINGULAR$=="OX" THEN
0160   PLURAL$:="OXEN"
0170 ELIF SINGULAR$=="WOMAN" THEN
0180   PLURAL$:="WOMEN"
0190 ELSE
0200   PLURAL$:=SINGULAR$+"S"
0210 ENDIF
0220 PRINT "THE SINGULAR IS ",SINGULAR$
0230 PRINT "THE PLURAL IS ",PLURAL$
0240 END

RUN
ENTER A SINGULAR NOUN
?
MOUSE
THE SINGULAR IS MOUSE
THE PLURAL IS MICE

RUN
ENTER A SINGULAR NOUN
?
CAT
THE SINGULAR IS CAT
THE PLURAL IS CATS

RUN
ENTER A SINGULAR NOUN
?
WOMAN
THE SINGULAR IS WOMAN
THE PLURAL IS WOMEN
```

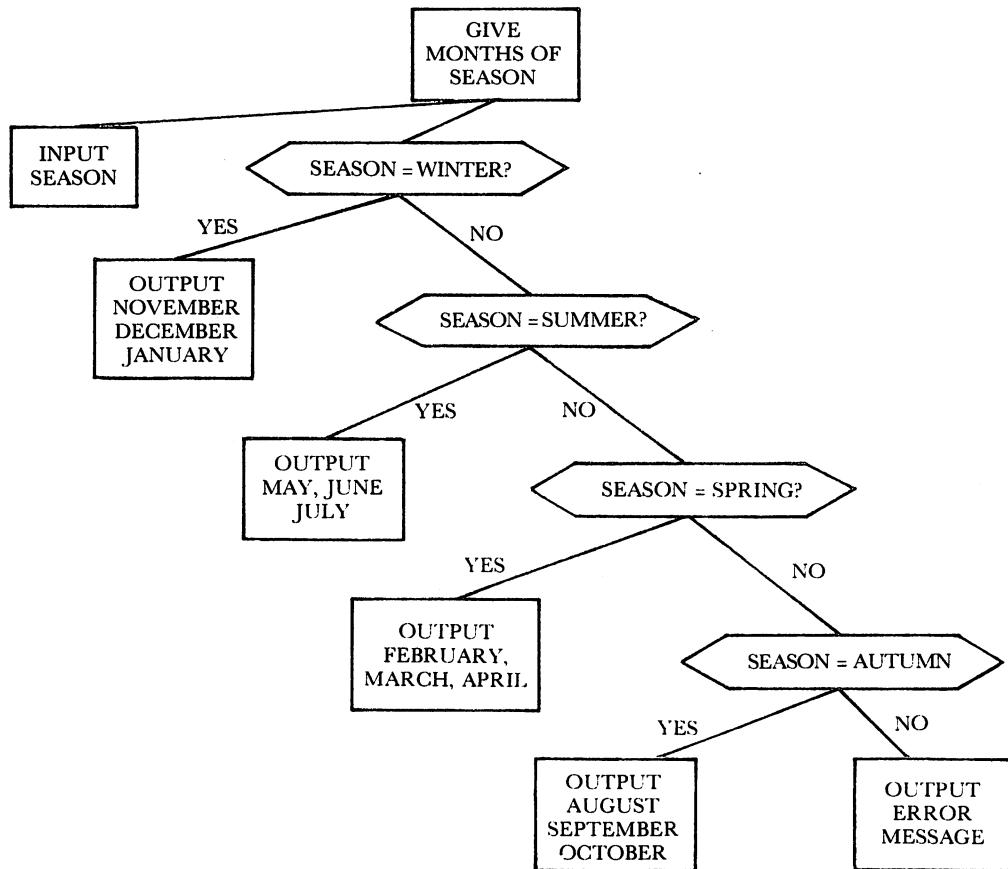
Notice that we make a double saving when we use ELIF:

- (i) ELIF is shorter than ELSE IF and requires one line instead of two.
- (ii) Only one ENDIF is required. Compare this with the four ENDIF's of Program 24.

Once again it is important to remember that only one of the statements governed by IF or ELIF will be executed, and the others will be ignored.

## Four Seasons

As a further illustration of multiple choices let us write a program to print the months of a given season on request, e.g. if the user requests WINTER the program will print NOVEMBER, DECEMBER and JANUARY.



The selection structure is very similar to our previous example. One feature worth noting is that we have made allowance for the making of an error on input. If a name other than WINTER, SPRING, SUMMER or AUTUMN is entered, the selection

process will work its way down to the rightmost box at the bottom level. Since the NO branch has been taken every time, we recognise that none of the valid season names has been used and so we output an error message.

In fact, in order to make our programs complete we should always include checks for errors in the input. However, since this would often make our program unduly long and perhaps obscure the main point being made, we shall only occasionally do so in this book. You are encouraged however to re-write the programs to include whatever error checks you feel are necessary.

```
0010 // PROGRAM 26
0020 //
0030 // COMELY KATE
0040 //
0050 // TO OUTPUT MONTHS OF A SEASON
0060 //
0070 DIM SEASON$ OF 6
0080 PRINT
0090 INPUT "WHAT SEASON ? ": SEASON$
0100 PRINT
0110 IF SEASON$="WINTER" THEN
0120   PRINT "NOVEMBER,DECEMBER,JANUARY"
0130 ELIF SEASON$="SUMMER" THEN
0140   PRINT "MAY,JUNE,JULY"
0150 ELIF SEASON$="SPRING" THEN
0160   PRINT "FEBRUARY,MARCH,APRIL"
0170 ELIF SEASON$="AUTUMN" THEN
0180   PRINT "AUGUST,SEPTEMBER,OCTOBER"
0190 ELSE
0200   PRINT "SORRY! NO SUCH SEASON"
0210 ENDIF
0220 END
```

RUN

WHAT SEASON ? SPRING

FEBRUARY,MARCH,APRIL

RUN

WHAT SEASON ? AUTUMN

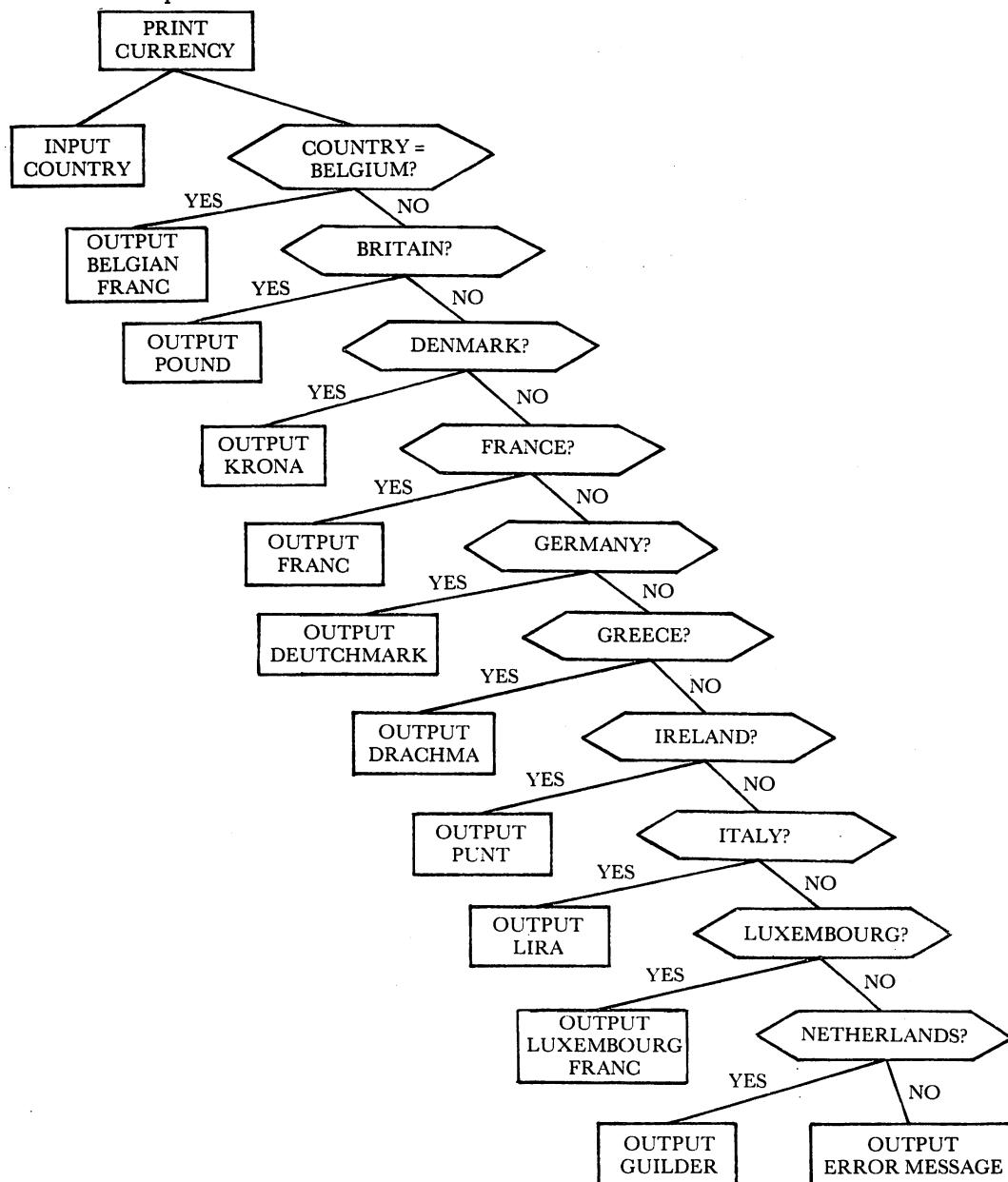
AUGUST,SEPTEMBER,OCTOBER

The structure should be familiar by now. One small point worth noting is that we do not have to specify the seasons in any particular order in our sequence of IF's and ELIF's. The computer does not know that SPRING follows WINTER, that SUMMER follows SPRING, etc. The logic of the program is unchanged no matter which order is chosen for the text.

The computer doesn't know either which months belong to each season and would quite cheerfully print "MAY, JUNE, JULY" for WINTER if we put this in our program. It would even print "MAY, DECEMBER, AUGUST" if we so specified. The computer has no knowledge of the world as we have. It is our responsibility to give it correct instructions.

## Currency Exchange

**Problem:** Write a program to print the unit of currency used in the countries of the EEC. This problem will serve as another illustration of the ELIF construct.



This is just an enlarged version of the structure we have met a couple of times already. Once again we have made allowance for an error on input.

```
0010 // PROGRAM 27
0020 //
0030 // COMELY KATE
0040 //
0050 // TO GIVE CURRENCIES OF EEC
0060 //
0070 DIM COUNTRY$ OF 11
0080 PRINT
0090 INPUT "WHAT COUNTRY ? ": COUNTRY$
0100 PRINT
0110 IF COUNTRY$="BELGIUM" THEN
0120 PRINT "BELGIAN FRANC"
0130 ELIF COUNTRY$="BRITAIN" THEN
0140 PRINT "POUND"
0150 ELIF COUNTRY$="DENMARK" THEN
0160 PRINT "KRONA"
0170 ELIF COUNTRY$="FRANCE" THEN
0180 PRINT "FRANC"
0190 ELIF COUNTRY$="GERMANY" THEN
0200 PRINT "DEUTCHMARK"
0210 ELIF COUNTRY$="GREECE" THEN
0220 PRINT "DRACHMA"
0230 ELIF COUNTRY$="IRELAND" THEN
0240 PRINT "PUNT"
0250 ELIF COUNTRY$="ITALY" THEN
0260 PRINT "LIRA"
0270 ELIF COUNTRY$="LUXEMBOURG" THEN
0280 PRINT "FRANC"
0290 ELIF COUNTRY$="NETHERLANDS" THEN
0300 PRINT "GUILDER"
0310 ELSE
0320 PRINT "SORRY! NOT IN EEC"
0330 ENDIF
0340 END
```

RUN

WHAT COUNTRY ? IRELAND

PUNT

RUN

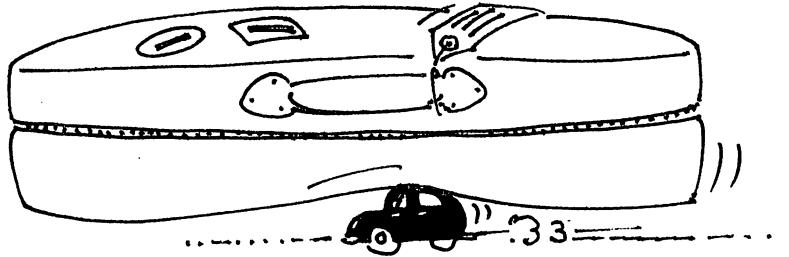
WHAT COUNTRY ? NETHERLANDS

GUILDER

RUN

WHAT COUNTRY ? CHILE

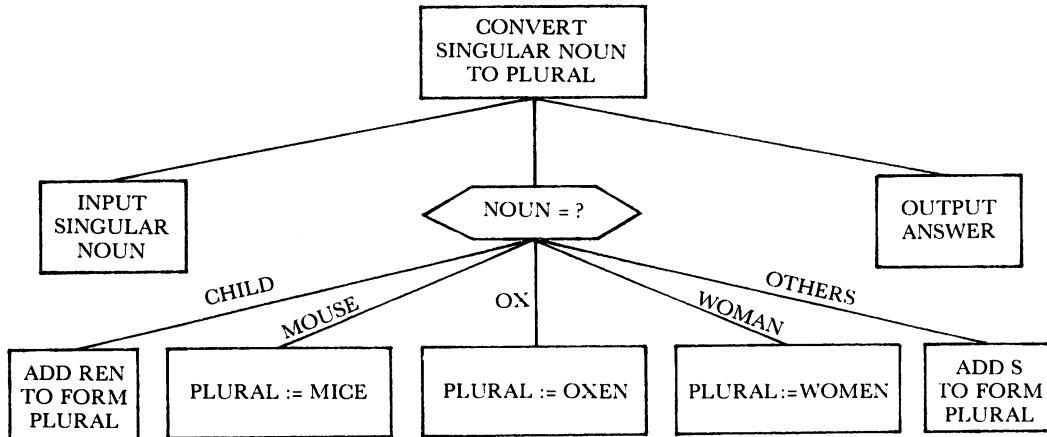
SORRY! NOT IN EEC



## What a Case!

Our last few examples have all been concerned with selecting one of a number of options. Each program dealt with a number of cases and the action taken by the program depended on which case applied. Essentially a sequence of IF ... ELSE decisions was used to run down through the various cases. COMAL provides us with an alternative method of handling this multiple selection process. It provides us with a **CASE** statement. The CASE statement is often a clearer and simpler method of representing the multiple choice situation.

The best way to study the working of the CASE structure is to see it in action in a program. Accordingly we will rewrite our program to convert singular nouns to plural (Program 24), this time using CASE instead of ELIF. We also re-draw our structure diagram in such a way as to represent the selection of one of a number of cases more accurately.



This time our selection box `NOUN = ?` does not indicate just two possibilities. It indicates a range of different selections. Only one of these selections will be made and the plural will be formed accordingly.

```

0010 // PROGRAM 28
0020 //
0030 // COMELY KATE
0040 //
0050 // TO CONVERT A SINGULAR NOUN TO PLURAL FORM
0060 //
0070 DIM SINGULAR$ OF 20, PLURAL$ OF 20
0080 PRINT "ENTER A SINGULAR NOUN"
0090 INPUT SINGULAR$
0100 PRINT
0110 CASE SINGULAR$ OF
0120 WHEN "CHILD"
0130   PLURAL$:="CHILDREN"
0140 WHEN "MOUSE"
0150   PLURAL$:="MICE"
0160 WHEN "OX"
0170   PLURAL$:="OXEN"
0180 WHEN "WOMAN"
0190   PLURAL$:="WOMEN"
0200 OTHERWISE
0210   PLURAL$:=SINGULAR$+"S"
0220 ENDCASE
0230 PRINT "THE SINGULAR IS ",SINGULAR$
0240 PRINT "THE PLURAL IS ",PLURAL$
0250 END

```

RUN

ENTER A SINGULAR NOUN  
?  
WOMAN

THE SINGULAR IS WOMAN  
THE PLURAL IS WOMEN

RUN

ENTER A SINGULAR NOUN  
?  
MOUSE

THE SINGULAR IS MOUSE  
THE PLURAL IS MICE

RUN

ENTER A SINGULAR NOUN  
?  
COMPUTER

THE SINGULAR IS COMPUTER  
THE PLURAL IS COMPUTERS

Line 110 comes straight to the point. In effect CASE SINGULAR\$ OF says - I am going to deal with the following cases of the string called SINGULAR\$. Then the program mentions each particular case:

“CHILD” on line 120  
 “MOUSE” on line 140  
 “OX” on line 160  
 and       “WOMAN” on line 180

Any other case that might arise is covered by the word **OTHERWISE**. Each particular case is preceded by the word **WHEN**. The whole structure is neatly closed off by the word **ENDCASE** (just as IF is closed by ENDIF).

The structure of the CASE statement is therefore

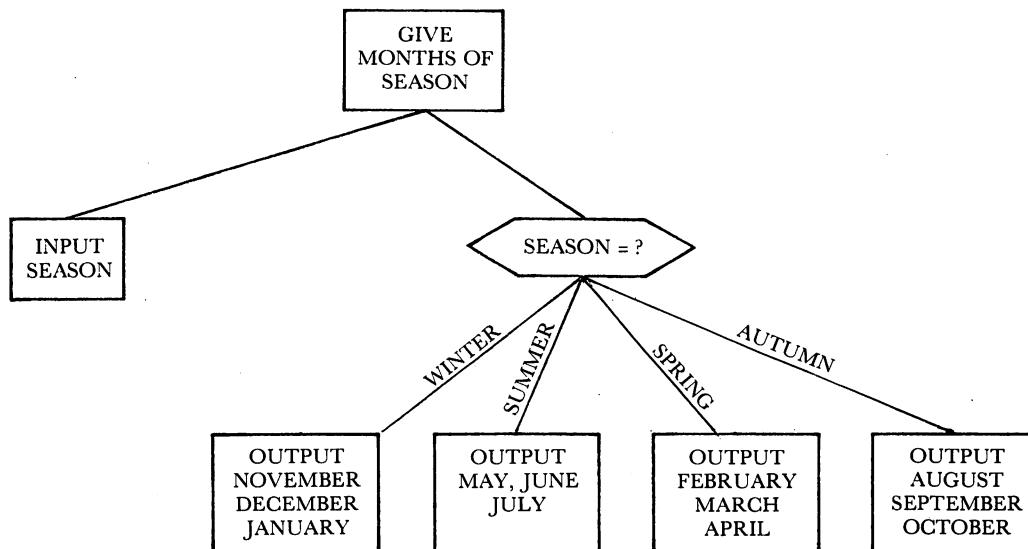
```
CASE . . . OF
    WHEN Case 1
        perform actions for case 1
    WHEN Case 2
        perform actions for case 2
    .
    .
    .
    ENDCASE
```

Note carefully that only one of the cases can arise and only one of the specified actions be performed in any particular run of the program. If none of the specifically mentioned cases arises then the action governed by OTHERWISE is performed.

It is not essential that every CASE statement include an OTHERWISE clause. If all the cases which can arise are mentioned explicitly, then there is no need for an OTHERWISE clause.

## Casing the other joints!

In pursuit of a deeper knowledge of the CASE construct let us repeat Programs 26 and 27 using the CASE statement instead of ELIF. The structure diagrams are redrawn to match the CASE construct more closely.



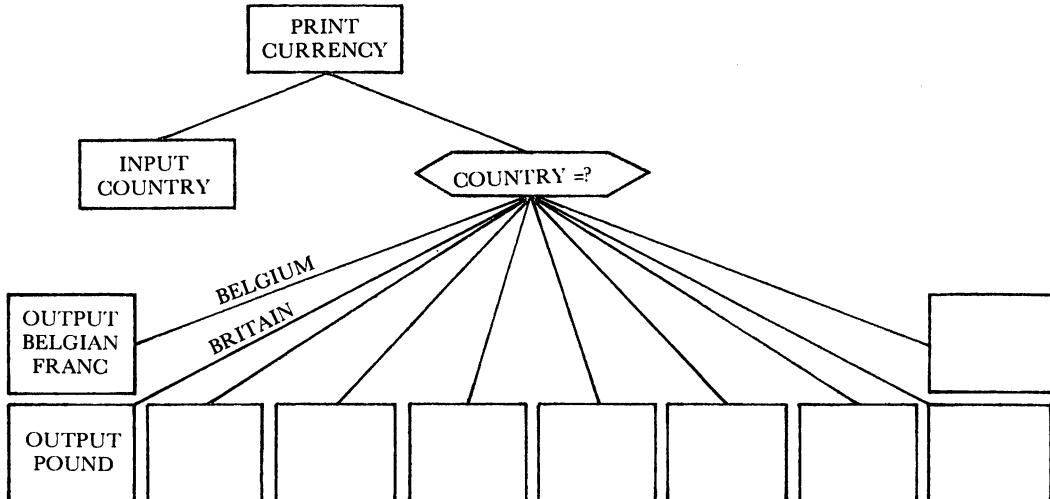
```

0010 // PROGRAM 29
0020 //
0030 // COMELY KATE
0040 //
0050 // TO OUTPUT MONTHS OF A SEASON
0060 //
0070 DIM SEASON$ OF 6
0080 PRINT
0090 INPUT "WHAT SEASON ? ": SEASON$
0100 PRINT
0110 CASE SEASON$ OF
0115 WHEN "WINTER"
0120 PRINT "NOVEMBER,DECEMBER,JANUARY"
0130 WHEN "SUMMER"
0140 PRINT "MAY,JUNE,JULY"
0150 WHEN "SPRING"
0160 PRINT "FEBRUARY,MARCH,APRIL"
0170 WHEN "AUTUMN"
0180 PRINT "AUGUST,SEPTEMBER,OCTOBER"
0190 OTHERWISE
0200 PRINT "SORRY! NO SUCH SEASON"
0210 ENDCASE
0220 END
RUN
WHAT SEASON ? SUMMER
MAY,JUNE,JULY
RUN
WHAT SEASON ? SPRING
FEBRUARY,MARCH,APRIL

```

Notice that we do not have an OTHERWISE clause in this program. There are only four seasons and each is covered by its own WHEN clause. Perhaps, however, you could think of a good reason why it might be useful to include an OTHERWISE clause!

And now our financial problem:



```
0010 // PROGRAM 30
0020 //
0030 // COMELY KATE
0040 //
0050 // TO GIVE CURRENCIES OF EEC
0060 //
0070 DIM COUNTRY$ OF 11
0080 PRINT
0090 INPUT "WHAT COUNTRY ? "; COUNTRY$
0100 PRINT
0110 CASE COUNTRY$ OF
0120 WHEN "BELGIUM"
0130 PRINT "BELGIAN FRANC"
0140 WHEN "BRITAIN"
0150 PRINT "POUND"
0160 WHEN "DENMARK"
0170 PRINT "KRONA"
0180 WHEN "FRANCE"
0190 PRINT "FRANC"
0200 WHEN "GERMANY"
0210 PRINT "DEUTCHMARK"
0220 WHEN "GREECE"
0230 PRINT "DRACHMA"
0240 WHEN "IRELAND"
0250 PRINT "PUNT"
0260 WHEN "ITALY"
0270 PRINT "LIRA"
0280 WHEN "LUXEMBOURG"
0290 PRINT "FRANC"
0300 WHEN "NETHERLANDS"
0310 PRINT "GUILDER"
0320 OTHERWISE
0330 PRINT "SORRY! NOT IN EEC"
0340 ENDCASE
0350 END
```

RUN

WHAT COUNTRY ? BELGIUM

BELGIAN FRANC

RUN

WHAT COUNTRY ? GERMANY

DEUTCHMARK

RUN

WHAT COUNTRY ? BRITAIN

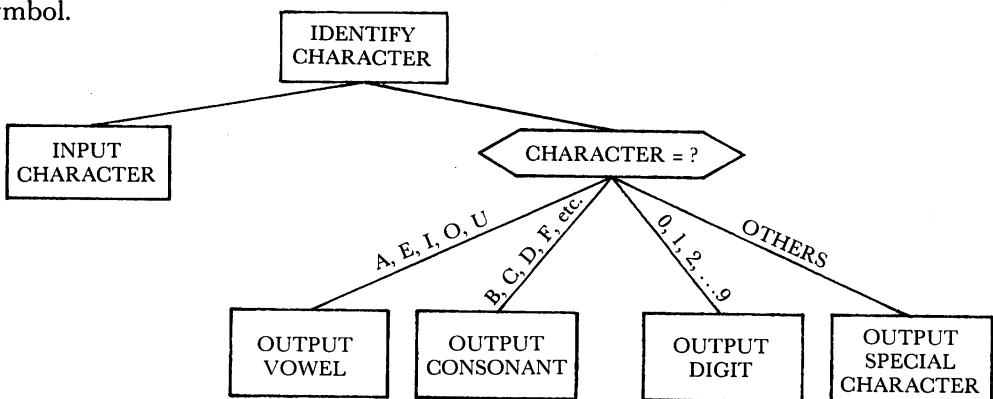
POUND

Once again we have not used an OTHERWISE clause and once again you are invited to think of a good reason for including one.

## Sticking to the letter

So far each occurrence of a case in a CASE construct has involved just one value, e.g. just one season or just one country, with the exception of course of the OTHERWISE case in our first example. We can however have a number of definite values covered in each WHEN clause. An example will illustrate this:

Let us write a program which invites a user to type in a single character at the keyboard and tells him whether it is a vowel, a consonant, a digit or some other symbol.



The logic should be clear from the diagram and the following program should be easy to follow.

```
0010 // PROGRAM 31
0020 //
0030 // COMELY KATE
0040 //
0050 // TO IDENTIFY SYMBOL ENTERED AT KEYBOARD
0060 //
0070 DIM CHARACTER$ OF 1
0080 PRINT
0090 INPUT "TYPE ANY CHARACTER ": CHARACTER$
0100 PRINT
0110 CASE CHARACTER$ OF
0120 WHEN "A", "E", "I", "O", "U"
0130 PRINT "VOWEL"
0140 WHEN "B", "C", "D", "F", "G", "H", "J", "K", "L", "M", "N", "P", "Q", "R",
      "S", "T", "V"
0150 PRINT "CONSONANT"
0160 WHEN "X", "Y", "Z"
0170 PRINT "CONSONANT"
0180 WHEN "0", "1", "2", "3", "4", "5", "6", "7", "8", "9"
0190 PRINT "DIGIT"
0200 OTHERWISE
0210 PRINT "SPECIAL CHARACTER"
0220 ENDCASE
0230 END
RUN
TYPE ANY CHARACTER U
VOWEL
```

```

RUN
TYPE ANY CHARACTER 3
DIGIT
RUN
TYPE ANY CHARACTER *
SPECIAL CHARACTER

```

On line 70 we declare CHARACTER\$ to be a string of just one character. When a value is input on line 90 we then go on to consider the various cases. Note that the possible values of the character in each WHEN clause are separated by commas, e.g. WHEN "A", "E", "I", "O", "U"

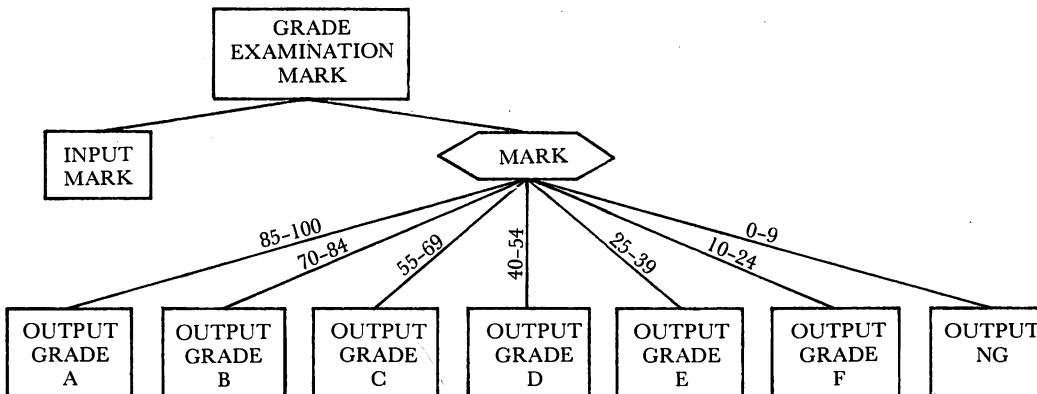
Note too that the list of consonants is too long to be included in one WHEN list and so we split it into two lists.

## A bad CASE indeed!

In spite of the natural way in which the CASE construct handles the multiple selection situations, it may not always be the most appropriate construct to use. Sometimes IF, ELIF is better. Consider the following:

**Problem:** Given an examination mark, assign a grade to it according to the following scheme –

85 ≤ MARK ≤ 100	GRADE A
70 ≤ MARK ≤ 84	GRADE B
55 ≤ MARK ≤ 69	GRADE C
40 ≤ MARK ≤ 54	GRADE D
25 ≤ MARK ≤ 39	GRADE E
10 ≤ MARK ≤ 24	GRADE F
0 ≤ MARK ≤ 9	NG



```
0010 // PROGRAM 32
0020 //
0030 // COMELY KATE
0040 //
0050 // TO GRADE EXAMINATION MARKS
0060 //
0070 PRINT
0080 INPUT "ENTER EXAM. MARK "; MARK
0090 PRINT
0100 IF MARK>100 THEN
0110   PRINT "ERROR!! IN INPUT"
0120 ELIF MARK>=85 THEN
0130   PRINT "GRADE A"
0140 ELIF MARK>=70 THEN
0150   PRINT "GRADE B"
0160 ELIF MARK>=55 THEN
0170   PRINT "GRADE C"
0180 ELIF MARK>=40 THEN
0190   PRINT "GRADE D"
0200 ELIF MARK>=25 THEN
0210   PRINT "GRADE E"
0220 ELIF MARK>=10 THEN
0230   PRINT "GRADE F"
0240 ELIF MARK>=0 THEN
0250   PRINT "NG"
0260 ELSE
0270   PRINT "ERROR!! IN INPUT"
0280 ENDIF
0290 END
```

RUN

ENTER EXAM. MARK 90

GRADE A

RUN

ENTER EXAM. MARK 70

GRADE B

RUN

ENTER EXAM. MARK 8

NG

The CASE statement could have been used here,  
e.g. CASE MARK OF

WHEN 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100 PRINT "A" etc.  
but I think you will agree that it would be very clumsy. The ELIF construction is  
better. Note that we cannot say something like

WHEN 85 TO 100

i.e. we cannot use an expression indicating a range of values; we must use individual  
constant values.

In the program we check for invalid marks, i.e. marks above 100 or below 0.

## **Summary**

Selecting between alternative courses of action is often important in constructing programs.

Before dividing in a program we should check the divisor is not zero.

Multiple selection is achieved by working through a sequence of choices, or by selecting one of a number of cases.

It is a good idea to check for erroneous input in programs.

### **COMAL KEYWORDS**

AUTO	CASE	CAT	CLEAR	CON	DATA	DEL
DELETE	DIV	EDIT	ELIF	END	ENDCASE	
ENDIF	IF...THEN...ELSE		INPUT	LIST	LOAD	
MOD	NEW	OTHERWISE	PRINT	READ	RENUM	
ROUND	RUN	SAVE	STOP	WHEN		

## **QUESTIONS and EXERCISES**

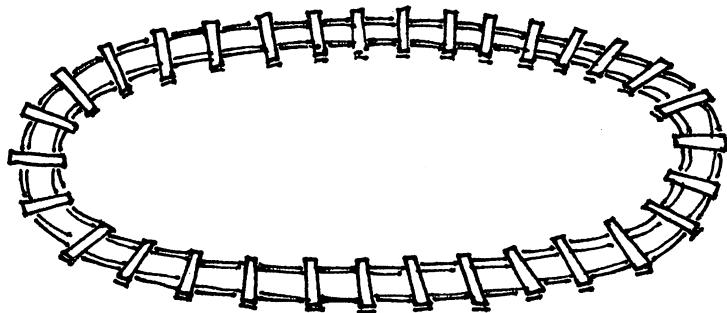
1. Write a program which accepts a number and checks whether it is divisible by 19 or not.
2. Write a program to calculate the value of the expression  $a + \frac{b}{c+d}$  for the following sets of values of  $a, b, c, d$ :  
(i)  $a = 4, b = 18, c = 3, d = 2$       (ii)  $a = 17, b = 92, c = -8, d = 8$ .
3. Write a program which accepts two numbers and prints them in the order larger first, smaller second.
4. Write a program which accepts one of the following names: BILLY, DILLY, JOE, ANGIE and writes a short note beginning DEAR SIR or DEAR MADAM as appropriate. What does your program do if one of those names is not entered?
5. Considering Program 24 or Program 28, what happens if you enter nonsense nouns? Could you overcome this? Is it easy to cater for all possible cases?
6. Write a program to read three numbers and output them in ascending order.
7. Write a program which accepts three lengths and determines if they form an equilateral triangle.
8. Repeat Question 7 but this time check for a right-angled triangle (Hint: use Pythagoras Theorem).
9. Write a program to accept a country of the EEC and output the name of its capital city.
10. Write a program to accept the name of a month of the year and output the number of days in that month.
11. Write a program to accept a verb and output its past tense. Take into account as many irregular verbs as you can.

# 4 Iteration I

## Over and over again

If you have typed in some of the programs in the earlier chapters, or if you have written programs of your own (which of course you have!) you have probably run each program several times. No doubt you considered it a bit of a nuisance to have to keep entering RUN to get the program to execute again. Is there any way, you wonder, to get the program to repeat automatically? The answer is *yes*.

Our main objective in this chapter is to discover ways to have sequences of program statements repeatedly executed without having to re-run the programs.



## Meaning the average

**Problem:** Find the mean or average of five numbers.

This is easily solved as follows:

```
0010 // PROGRAM 33
0020 //
0030 // COMELY KATE
0040 //
0050 // TO CALCULATE MEAN OF 5 NUMBERS
0060 //
0070 PRINT
0080 INPUT N1, N2, N3, N4, N5
0090 PRINT
0100 SUM:=N1+N2+N3+N4+N5
0110 MEAN:=SUM/5
0120 PRINT "THE AVERAGE IS ",MEAN
0130 END
```

RUN

```
?  
45,98,67,65,50
```

```
THE AVERAGE IS 65
```

Suppose however we want to find the mean of 50 numbers, or the mean of 500 numbers. It would be a bit clumsy, to say the least, to have to write - INPUT N1, N2, N3, N4, N5, N6, N7, N8, ... etc. Phew!

Is there a better way? There is!

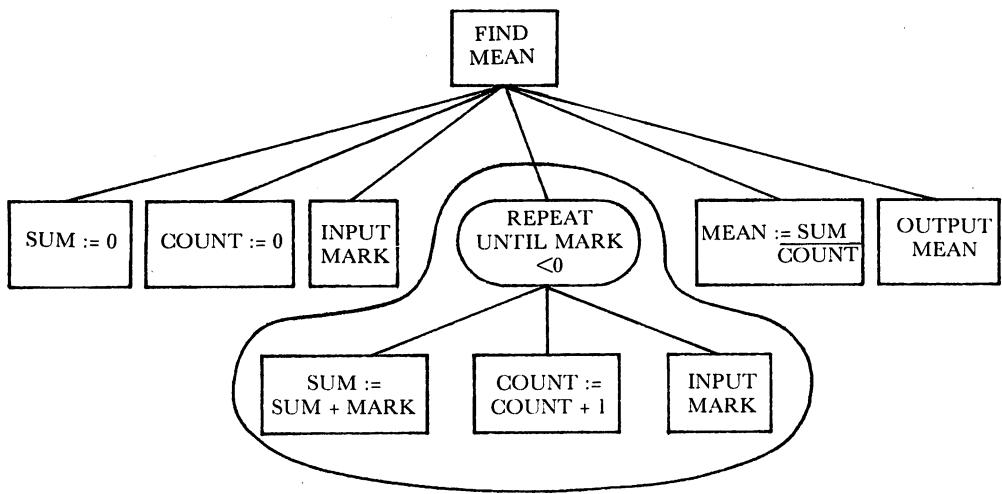
In order to illustrate this let us re-state the problem as follows: Find the mean of any set of positive numbers, e.g. examination marks.

The solution is to keep reading in numbers and adding them up until there are no more. The question is - how will the program know that there are no more numbers? Obviously we will have to tell it. (It is no good just pressing RETURN when it is waiting for more input). But of course we can't just say to it - 'That's it, old chap! Now go and calculate the mean.' Neither will it assume that if we don't enter a number when it asks for one we are finished and it should proceed to compute the mean.

So what can we do?

Well, when we are writing a letter or an essay, or even a book, how do we tell the reader that we have finished a sentence? Of course, we use a full stop. The full stop acts as a signal which indicates the end of a sentence. We will indicate the end of our set of numbers by giving the computer a similar full stop signal. We can't use an actual full stop character since the computer is expecting a number. What we will use is a number which indicates to the computer that we are finished entering the actual numbers to be averaged. For this particular example we will use a negative number. We do this because all our marks should be positive numbers and so a negative value is a good way to indicate that we have reached the end of the valid data. In other words, a negative number will act as a full stop. Since this negative value is deliberately not a valid piece of data, it is sometimes known as a **rogue value**.





The problem of finding the mean breaks down into six tasks as specified on level 1 of our structure diagram. Five of these are simple tasks:

**SUM := 0**

Give a variable called SUM an initial value of 0. SUM will be used to accumulate the sum of the numbers as they are input.

**COUNT := 0**

Give a variable called COUNT an initial value of 0. COUNT will be used to count the number of valid data values entered.

**INPUT MARK**

Input the first mark in the list.

**MEAN: =  $\frac{\text{SUM}}{\text{COUNT}}$**

Calculate the mean by dividing the sum of the marks by the number of marks.

**OUTPUT MEAN**

Output the answer.

The remaining task is in fact a repetition of three simple tasks. Notice that we introduce a new shape **oval** to represent repetition. The three statements which are executed over and over again until a negative number is entered are:

**SUM := SUM + MARK**

The mark which is entered is added to the existing value of SUM, to produce a new value of SUM. Note that the value of SUM on the right of the := is the old value of SUM and when the addition takes place the result constitutes the new value of SUM on the left.

**COUNT := COUNT + 1**

Increase the count by 1.

**INPUT MARK**

Take in a new mark.

Let us see now how COMAL copes with the logic of repetition or **iteration** as it is often called.

```

0010 // PROGRAM 34
0020 //
0030 // COMELY KATE
0040 //
0050 // TO CALCULATE MEAN OF EXAMINATION MARKS
0060 //
0065 PRINT
0070 SUM:=0
0080 COUNT:=0
0090 INPUT "FIRST MARK ? ": MARK
0100 REPEAT
0110   SUM:=SUM+MARK
0120   COUNT:=COUNT+1
0130 INPUT "NEXT MARK ? ": MARK
0140 UNTIL MARK<0
0150 MEAN:=SUM/COUNT
0160 PRINT
0170 PRINT "MEAN = ",MEAN
0180 END

```

RUN

```

FIRST MARK ? 65
NEXT MARK ? 48
NEXT MARK ? 87
NEXT MARK ? 67
NEXT MARK ? 56
NEXT MARK ? 90
NEXT MARK ? 85
NEXT MARK ? -5

```

MEAN = 71.14286

COMAL provides us with a straightforward no-nonsense method of expressing an iteration.

### **REPEAT . . . UNTIL**

i.e. repeat a statement or sequence of statements until some condition is satisfied. The program **loops** round and round through the same instructions until the condition is met (unless of course the condition holds after the first run through the loop, in which case the loop terminates and the program continues with the statement after **UNTIL**).

The condition which is used to bring the iteration to a halt is that a mark is negative. This is expressed quite simply as **MARK<0**. The ordinary mathematical sign ( $<$ ) is used to denote 'less than'.

Let us be absolutely clear on what happens when the **UNTIL** statement is reached with a positive mark. The program 'thinks' – have I to go round this loop again? Let me see! Is **MARK** less than zero? No. So off I go back to the first statement after **REPEAT** and round again through the three statements of the loop.

If however the **MARK** is negative, the program knows it must terminate the loop and not go round again. So what happens? The program automatically continues with the statement after **UNTIL**, which is line 150 in Program 34.

To clarify the action of this most important iterative construct, we will draw up a little table to show how the program works on the following data 40, 64, 50, -5. Note that we have used -5 as a rogue value to signal the end of input.

1st ITERATION					2nd ITERATION				3rd ITERATION			
After Line	Mark	Sum	Count	Condition Mark < 0	Mark	Sum	Count	Mark < 0	Mark	Sum	Count	Mark < 0
90	40	0	0	—	64	104	1	—	50	154	2	—
100	40	0	0	—	64	104	2	—	50	154	3	—
110	40	40	0	—	50	104	2	—	-5	154	3	—
120	40	40	1	—	50	104	2	—	-5	154	3	—
130	64	40	1	—	50	104	2	—	-5	154	3	—
140	64	40	1	FALSE	50	104	2	FALSE	-5	154	3	TRUE

We show the ‘value’ of the condition (MARK < 0) only at line 140 where it is actually tested. It is of course false until the number -5 is input. Note that since the condition is tested at the end, the loop is executed *at least once*. All REPEAT loops are executed at least once.

## At your request

Use of a rogue value is not the only way to terminate entry of data. Thinking again about our averaging problem, we might prefer to actually ask the user each time if he has another number to enter and keep looping round as long as he says YES. We will re-write our program to reflect this approach.

```

0010 // PROGRAM 35
0020 //
0030 // COMELY KATE
0040 //
0050 // TO CALCULATE MEAN OF EXAMINATION MARKS
0060 //
0070 DIM REPLY$ OF 3
0080 SUM:=0
0090 COUNT:=0
0100 REPEAT
0110   PRINT
0120   INPUT "MARK ? "; MARK
0130   SUM:=SUM+MARK
0140   COUNT:=COUNT+1
0150   PRINT
0160   INPUT "MORE(YES/NO) ? "; REPLY$
0170 UNTIL REPLY$="NO"
0180 MEAN:=SUM/COUNT
0190 PRINT
0200 PRINT "MEAN = ",MEAN
0210 END

```

RUN

MARK ? 65

```
MORE(YES/NO) ? YES
MARK ? 48
MORE(YES/NO) ? YES
MARK ? 87
MORE(YES/NO) ? YES
MARK ? 67
MORE(YES/NO) ? YES
MARK ? 56
MORE(YES/NO) ? YES
MARK ? 90
MORE(YES/NO) ? YES
MARK ? 85
MORE(YES/NO) ? NO
MEAN = 71.14286
```

Here we declare a string variable REPLY\$ to hold the answer to the question posed in line 160, i.e. does the user want to enter more data or not? If he says YES then the loop is executed again. If he says NO the loop is terminated because the condition for termination is that REPLY\$ should equal NO.

Note that we have brought the INPUT MARK instruction in at the beginning of the loop, even for the first mark. Why is this? See if you can figure out for yourself why we have made the change.

Iteration is a most important element in the construction of programs. It would do us no harm at all to consider a few more examples.

## Facts about Factorials

Find the value of 7! (factorial 7). You remember of course what 7! means (whisper - just in case you don't,  $7! = 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$ )

```
0010 // PROGRAM 36
0020 //
0030 // COMELY KATE
0040 //
0050 // TO CALCULATE 7!
0060 //
0070 FACTORIAL:=1
0080 NUMBER:=7
```

```

0090 REPEAT
0100   FACTORIAL:=FACTORIAL*NUMBER
0110   NUMBER:=NUMBER-1
0120 UNTIL NUMBER=0
0130 PRINT
0140 PRINT "7! = ",FACTORIAL
0150 END

RUN

```

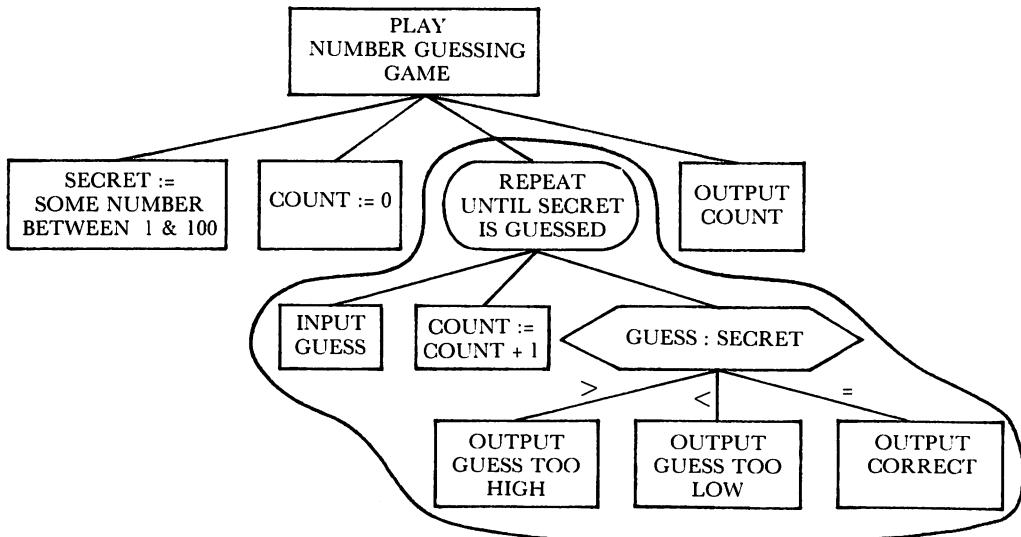
7! = 5040

FACTORIAL is given an initial value of 1 in line 70 (why would it be wrong to give FACTORIAL an initial value of 0?). NUMBER is given an initial value of 7 and is repeatedly decreased by 1 within the loop (line 110). Also within the loop FACTORIAL is continually multiplied by NUMBER until NUMBER has been reduced to zero. You are advised to work through the iteration to see exactly what happens and verify that the program really does calculate 7.6.5.4.3.2.1.

## Guess the Lucky Number!

Let us program the computer to play a little game.

The computer will decide on a number between 1 and 100 and will invite a user to guess the number. If a wrong number is guessed the computer will tell the user whether his guess is too high or too low and invite him to guess again. Guessing will continue until the correct number is arrived at. The number of guesses will be counted.



The REPEAT loop involves three activities:

- (1) accepting a guess
- (2) incrementing a counter by 1
- (3) checking whether the guess is too high, too low or just right. The entry in our selection box says - how does GUESS compare with SECRET? - and the branches allow for greater than, less than, or equal.

```
0010 // PROGRAM 37
0020 //
0030 // COMELY KATE
0040 //
0050 // TO PLAY A NUMBER GUESSING GAME
0060 //
0070 SECRET:=RND(1,100)
0080 COUNT:=0
0090 REPEAT
0100 INPUT "GUESS = ? "; GUESS
0110 COUNT:=COUNT+1
0120 IF GUESS>SECRET THEN
0130   PRINT "YOUR GUESS IS TOO HIGH. GUESS AGAIN"
0140   ELIF GUESS<SECRET THEN
0150     PRINT "YOUR GUESS IS TOO LOW. GUESS AGAIN"
0160   ELSE
0170     PRINT "BANG ON!!!"
0180   ENDIF
0190 UNTIL GUESS=SECRET
0200 PRINT
0210 PRINT "YOU TOOK ",COUNT," GUESSES"
0220 END
```

RUN

```
GUESS = ? 50
YOUR GUESS IS TOO LOW. GUESS AGAIN
GUESS = ? 75
YOUR GUESS IS TOO HIGH. GUESS AGAIN
GUESS = ? 62
YOUR GUESS IS TOO LOW. GUESS AGAIN
GUESS = ? 68
YOUR GUESS IS TOO HIGH. GUESS AGAIN
GUESS = ? 65
YOUR GUESS IS TOO HIGH. GUESS AGAIN
GUESS = ? 64
BANG ON!!!
```

YOU TOOK 6 GUESSES

The program follows the structure diagram very closely and should be easy to understand. You may however be puzzled by line 70.

Here we have used a handy little function provided by COMAL, the RND function. When we use the word RND, COMAL produces for us an unpredictable or *random* number. By using RND (1,100) we specify that we want a random number between 1 and 100. The interesting thing is that as the program stands we won't know what the random number is until it has been guessed, so SECRET is not a bad name for it at all. Of course we could have cheated and put a PRINT SECRET instruction into the program.

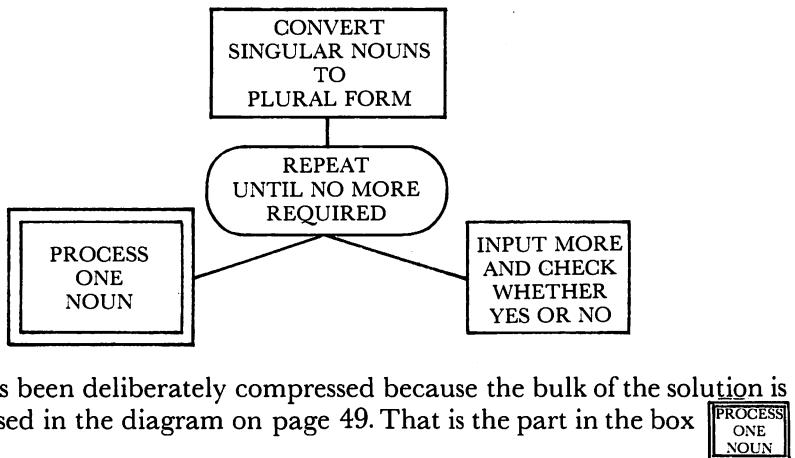
You may find if you run the program a few times that the computer keeps producing the same random number all the time, which of course is hardly unpredictable! Well, we can fix that. Simply put the instruction RANDOM in before line 70

e.g. 65 RANDOM

This will cause the program to generate a different random number *each* time it is run; this means that the number will *always* be unpredictable.

## Noun again

As a final example of iteration in this chapter, let us write our singular to plural conversion program, to allow several nouns to be processed in a single run. In fact the program will keep requesting more from the user until he has had enough and says NO.



The diagram has been deliberately compressed because the bulk of the solution is the same as expressed in the diagram on page 49. That is the part in the box

PROCESS  
ONE  
NOUN

It is often convenient to be able to present a sub-task of the overall solution in this way. We shall have a great deal to say about this later, when we talk about procedures.

```
0010 // PROGRAM 38
0020 //
0030 // COMELY KATE
0040 //
0050 // TO CONVERT SINGULAR NOUNS TO PLURAL FORM
0060 //
0070 DIM MORE$ OF 3
0080 DIM SINGULAR$ OF 20, PLURAL$ OF 20
0090 REPEAT
0100 PRINT "ENTER A SINGULAR NOUN"
0110 INPUT SINGULAR$
0120 PRINT
0130 CASE SINGULAR$ OF
0140 WHEN "CHILD"
0150   PLURAL$:="CHILDREN"
0160 WHEN "MOUSE"
0170   PLURAL$:="MICE"
```

```
0180 WHEN "OX"
0190   PLURAL$:="OXEN"
0200 WHEN "WOMAN"
0210   PLURAL$:="WOMEN"
0220 OTHERWISE
0230   PLURAL$:=SINGULAR$+"S"
0240 ENDCASE
0250 PRINT "THE SINGULAR IS ",SINGULAR$
0260 PRINT "THE PLURAL IS ",PLURAL$
0270 PRINT
0280 PRINT
0290 INPUT "MORE (YES/NO) ? ": MORE$
0300 PRINT
0310 UNTIL MORE$="NO"
0320 END
```

RUN

ENTER A SINGULAR NOUN

?

CAT

THE SINGULAR IS CAT

THE PLURAL IS CATS

MORE (YES/NO) ? YES

ENTER A SINGULAR NOUN

?

MOUSE

THE SINGULAR IS MOUSE

THE PLURAL IS MICE

MORE (YES/NO) ? YES

ENTER A SINGULAR NOUN

?

CHILD

THE SINGULAR IS CHILD

THE PLURAL IS CHILDREN

MORE (YES/NO) ? YES

ENTER A SINGULAR NOUN

?

WOMAN

THE SINGULAR IS WOMAN

THE PLURAL IS WOMEN

MORE (YES/NO) ? NO

Notice that most of the lines within the REPEAT...UNTIL loop (lines 100 to 260) are exactly the same as lines 80 to 240 in Program 28. The extra lines deal with a request to the user to indicate whether he wants more or not. There are a couple of PRINT instructions for spacing purposes, an INPUT instruction, another PRINT and finally the UNTIL condition.

## Summary

It is important to be able to use iteration in programs.

Iteration means the possible repeated execution of a set of instructions.

The symbol  is used in structure diagrams to indicate iteration.

REPEAT ... UNTIL is one way by which COMAL expresses iteration.

### COMAL KEYWORDS

AUTO	CASE	CAT	CLEAR	CON	DATA
DEL	DELETE	DIV	EDIT	ELIF	END
ENDCASE	ENDIF	IF...THEN...ELSE		INPUT	
LIST	LOAD	MOD	NEW	OTHERWISE	
PRINT	READ	RENUM	REPEAT	RND	
ROUND	RUN	SAVE	STOP	UNTIL	WHEN

### QUESTIONS and EXERCISES

1. Why is iteration important? What are the vital ingredients for the correct formation of an iterative loop?
2. Write programs to find out when the sum of each of the following series exceeds 1000. Count the number of terms needed in each case.
  - (a)  $1 + 2 + 3 + 4 + \dots$
  - (b)  $1 + 3 + 5 + 7 + \dots$
  - (c)  $1 + 4 + 9 + 16 + 25 + \dots$
  - (d)  $100 + 50 + 25 + 12.5 + 6.25 + \dots$
  - (e)  $512 + 256 + 128 + 64 + \dots$
  - (f)  $256 + 128 + 64 + 32 + \dots$  (Beware!)
3. If a number starts at 5 and is continually doubled, how long will it take before the answer is greater than one hundred million? Write a program to find out.
4. You can simulate the tossing of a pair of dice by calculating RND (1,6) twice. Write a program which repeatedly tosses a pair of dice and prints the results until two sixes are thrown. Print the number of throws.
5. How long will it take a given sum of money to double itself at 15% per annum compound interest? Write a program to find out.
6. Inflation is running at 12% per annum. What will the cost of a car which now costs £3,500 be in 6 years time?
7. Write a program to convert a number expressed in denary form to binary form.
8. Write a program to calculate and print the first 25 triangle numbers, i.e. numbers such as 1, 3, 6, etc. which form triangles.

# 5 Iteration II

## Real Square, Man

Taking up where we left off in the last chapter let us write a little program to print a table of the squares of the integers from 1 to 30.

```
0010 // PROGRAM 39
0020 //
0030 // COMELY KATE
0040 //
0050 // TO ILLUSTRATE A SIMPLE ITERATIVE LOOP
0060 //
0070 PRINT
0080 PRINT "NUMBER      SQUARE"
0090 PRINT
0100 NUMBER:=0
0110 REPEAT
0120   PRINT NUMBER,"      ",NUMBER*NUMBER
0130   NUMBER:=NUMBER+1
0140 UNTIL NUMBER>30
0150 END
```

RUN

NUMBER	SQUARE
0	0
1	1
2	4
3	9
4	16
5	25
6	36
7	49
8	64
9	81
10	100
11	121
12	144
13	169
14	196
15	225
16	256
17	289
18	324
19	361
20	400
21	441
22	484
23	529
24	576
25	625
26	676
27	729
28	784
29	841
30	900

## Variable and Fixed Iteration

There is an important distinction to be drawn between the iteration in the last program and the iteration in most of the examples in the last chapter.

In Program 39 the loop was executed a fixed number of times, i.e. 30 times. Furthermore this number was known before the loop was entered, since it had been decided to print squares of numbers from 1 to 30. In most of the programs in Chapter 4 however it was not known beforehand how many times a loop would be traversed. (Can you spot the exception?) For example in Program 38 there was no means of knowing when the user would answer NO to the question MORE? So we just had to keep repeating the loop until the answer was NO.

In situations where a loop is to be executed a fixed number of times, COMAL provides us with a more suitable method of expressing the loop.

This is the **FOR ... NEXT** construct.

Let us study how this works by rewriting Program 39 using FOR ... NEXT.

```
0010 // PROGRAM 40
0020 //
0030 // COMELY KATE
0040 //
0050 // TO PRINT SQUARES OF INTEGERS FROM 1 TO 30
0060 //
0070 PRINT
0080 PRINT "NUMBER      SQUARE"
0090 PRINT
0100 FOR NUMBER:=1 TO 30 DO
0110   PRINT "  ",NUMBER,"      ",NUMBER*NUMBER
0120 NEXT NUMBER
0130 END
RUN
NUMBER      SQUARE
1          1
2          4
3          9
4          16
5          25
6          36
7          49
8          64
9          81
10         100
11         121
12         144
13         169
14         196
15         225
16         256
17         289
18         324
19         361
20         400
21         441
22         484
23         529
24         576
25         625
```

26	676
27	729
28	784
29	841
30	900

Line 100 says in effect: 'starting with NUMBER = 1 and going up to 30 in increments of 1, do the following statement.' Line 120 - NEXT NUMBER - says: 'now is the time to step NUMBER up by 1 and check whether or not it has gone above 30. If it has not, then go back and do statement 110 again. If it has then go on to the next statement, i.e. line 130', which just happens to be the end of the program.

NUMBER is called a *control variable* because each time through the loop its value controls whether another iteration is to be done or not. Notice how conveniently compact the FOR ... NEXT structure is in this case. The number of iterations is completely fixed in line 100 and from there on everything is automatic.

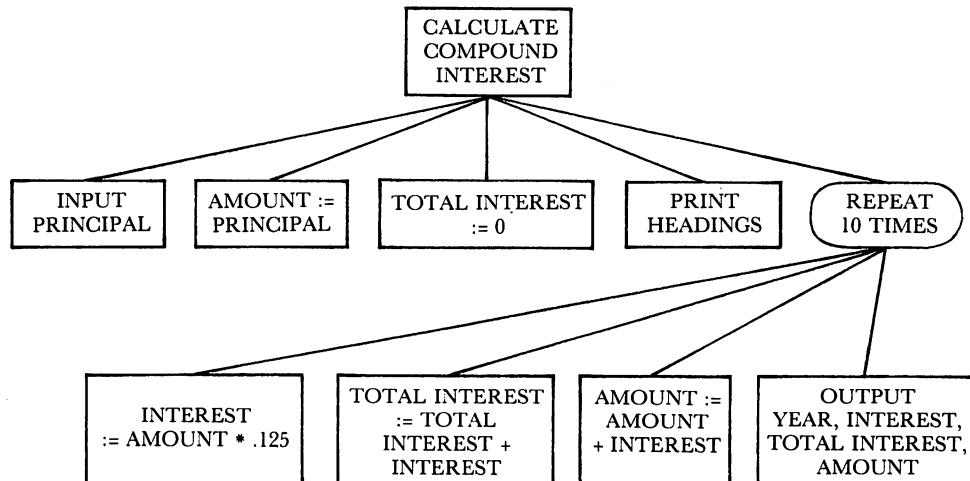
Four important elements are involved in this loop construction:

- (1) initialisation of a suitable control variable - NUMBER set to 1.
- (2) changing the value of the control variable - NUMBER increased by 1 by the statement NEXT NUMBER
- (3) testing the control variable against some limit value - NUMBER is tested against 30 at the statement NEXT NUMBER.
- (4) executing a statement - line 110 in this case.

We could just as easily have a compound statement (i.e. a group of statements) in our FOR loop as the next example shows.

## Interesting, eh!

A sum of money is invested for 10 years at  $12\frac{1}{2}\%$  per annum Compound Interest. Find the amount and the total interest earned. At the end of each year print out the interest for that year, the total interest earned so far and the amount.



The method is relatively straightforward. The initial values are set up and some suitable headings are printed on level 1. Then the same four tasks are repeated ten times, namely:

- (i) calculating a year's interest by  $\text{AMOUNT} \times \frac{12.5}{100}$
- (ii) adding this interest to the total interest so far
- (iii) adding the year's interest to the total amount so far
- (iv) printing the correct values.

```
0010 // PROGRAM 41
0020 //
0030 // COMELY KATE
0040 //
0050 // TO CALCULATE COMPOUND INTEREST
0060 //
0065 TAB:=6
0070 PRINT
0080 INPUT "SUM TO BE INVESTED = ": PRINCIPAL
0090 AMOUNT:=PRINCIPAL
0100 TOTALINTEREST:=0
0110 PRINT
0120 PRINT "      YEAR'S  TOTAL INTEREST"
0130 PRINT "YEAR INTEREST      SO FAR      AMOUNT"
0140 PRINT
0150 FOR YEAR:=1 TO 10 DO
0160   INTEREST:=AMOUNT*0.125
0170   TOTALINTEREST:=TOTALINTEREST+INTEREST
0180   AMOUNT:=AMOUNT+INTEREST
0190   PRINT YEAR,INTEREST,TOTALINTEREST,AMOUNT
0200 NEXT YEAR
0210 END
```

RUN

SUM TO BE INVESTED = 5000

	YEAR'S INTEREST	TOTAL INTEREST	SO FAR	AMOUNT
1	625	625	5625	
2	703.125	1328.125	6328.125	
3	791.0156	2119.141	7119.141	
4	889.8926	3009.033	8009.033	
5	1001.129	4010.162	9010.162	
6	1126.27	5136.433	10136.43	
7	1267.054	6403.487	11403.49	
8	1425.436	7828.923	12828.92	
9	1603.615	9432.538	14432.54	
10	1804.067	11236.61	16236.6	

Corresponding to the four tasks on level 2 of the structure diagram, we have four statements snuggled inside our FOR loop (lines 160 to 190). All of these are executed ten times.

## Countdown

So far our FOR loops have used a control variable or counter which is stepped *up* by 1 each time round the loop. We can in fact also step *down*. As an example let us rewrite our program to compute 7! (Did you spot this as the exception mentioned earlier?)

```
0010 // PROGRAM 42
0020 //
0030 // COMELY KATE
0040 //
0050 // TO FIND THE VALUE OF 7!
0060 //
0070 PRINT
0080 FACTORIAL:=1
0090 FOR COUNT:=7 DOWNTO 1 DO
0100   FACTORIAL:=FACTORIAL*COUNT
0110 NEXT COUNT
0120 PRINT "7! = ",FACTORIAL
0130 END

RUN
7! = 5040
```

Notice that we use the word DOWNTO instead of TO (see line 90) when we wish to count down.

## The FOR loop STEPS out

In the FOR ... NEXT examples up to this the control variable has always been increased or decreased by 1. It is possible however to change the control variable by a number other than 1. Suppose, for example we want to find the sum of the first 100 odd integers, i.e.  $1 + 3 + 5 + 7 + \dots + 199$ . The following program does the trick.

```
0010 // PROGRAM 43
0020 //
0030 // COMELY KATE
0040 //
0050 // TO EVALUATE 1+3+5+7+....+199
0060 //
0070 SUM:=0
0080 FOR NUM:=1 TO 199 STEP 2 DO
0090   SUM:=SUM+NUM
0100 NEXT NUM
0110 PRINT
0120 PRINT "1+3+5+7+....+199 = ",SUM
0130 END

RUN
```

```
1+3+5+7+....+199 =      10000
```

Line 80 is the line to note. We have used a new form of the FOR statement which includes a **STEP** part.

The STEP part indicates how the control variable is to be changed on each traversal of the loop. In this case the change is 2, so that the control variable NUM increases by 2 each time at the statement NEXT NUM.

We could, of course, have written our earlier examples with explicit statements that the STEP was to be 1,

e.g.           FOR NUMBER := 1 TO 30 STEP 1 DO

However, if the STEP part is omitted the computer will assume it to be 1. Therefore we normally take the lazy way out. Simple is best!

Some examples of valid FOR statements are:

- (a) FOR A := 5 TO 17 STEP 3 DO
- (b) FOR J := -7 TO 10 STEP 2 DO
- (c) FOR C := 4 TO 8 STEP 0.2 DO
- (d) FOR COUNT := 20 TO 0 STEP -4 DO
- (e) FOR T := 10 TO 4 DO

Examples (b) and (e) deserve comment. In (b), since J steps up 2 each time from a starting value of -7, it will not reach 10 exactly. Its successive values will be -7, -5, -3, -1, 1, 3, 5, 7, 9. The loop will be executed for all these values. After 9 the next value will be 11. Because this is above the upper limit of 10, the loop will not be executed and the program will carry on with the next statement after the loop.

In example (e) the assumed step is 1. But T cannot step *up* in increments of 1 from 10 to 4. The very first value of T is above the limit. The loop therefore will not be executed at all. It will be bypassed.

## Polynomial Polyphony

**Problem:** Tabulate values of the polynomial expression

$$10x^3 - 13x^2 + 19x - 25$$

for values of  $x$  from 0 to 2 in steps of 0.1

The following simple program does the job:

```
0010 // PROGRAM 44
0020 //
0030 // COMELY KATE
0040 //
0050 // TO TABULATE VALUES OF THE
0060 //
0070 //           3      2
0080 // EXPRESSION 10X - 13X + 19X - 25
0090 //
0100 PRINT
0110 PRINT "X           VALUE OF EXPRESSION"
0120 PRINT
0130 FOR X:=0 TO 2 STEP 0.1 DO
0140   EXPRESSION:=10*X*X*X-13*X*X+19*X-25
0150   PRINT X,"           ",EXPRESSION
0160   NEXT X
0170 END
```

RUN

X	VALUE OF EXPRESSION
0	-25
0.1	-23.22
0.2	-21.64
0.3	-20.2
0.4	-18.84
0.5	-17.5
0.6	-16.12
0.7	-14.64
0.8000001	-13
0.9000001	-11.14
1	-8.999998
1.1	-6.519995
1.2	-3.639994
1.3	-0.2999916
1.4	3.560009
1.5	8.000012
1.6	13.08001
1.7	18.86002
1.8	25.40002
1.9	32.76002

The program is straightforward and the computer dashes off the calculations at speed. However, although the computer does not mind doing all the work involved in calculating the value of the expression each time, we can, in fact, rewrite  $10x^3 - 13x^2 + 19x - 25$  in a way which makes it more economical to compute:

$$10x^3 - 13x^2 + 19x - 25 \equiv ((10x - 13)x + 19)x - 25$$

If we choose to write our expression in this form, line 140 would change to

```
EXPRESSION := ((10 * X - 13) * X + 19) * X - 25
```

Why is this form of the expression better than the original? To see why, count the number of operators (\*, +, -) in each expression. You will find that the original form involves 9 operations (6 multiplications, 2 subtractions and 1 addition) while the new form involves just 6 (3 multiplications, 2 subtractions and 1 addition). There is therefore a saving in work of about 33%.

The expression  $((10x - 13)x + 19)x - 25$  is an example of what is known as **nested** multiplication. It is usually better, even for mental calculations, to write expressions in nested multiplication form.

In general

$$\begin{aligned} & a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \cdots + a_2 x^2 + a_1 x + a_0 \\ & \equiv (\cdots (((a_n x + a_{n-1}) x + a_{n-2}) x + a_{n-3}) x + \cdots + a_1) x + a_0 \end{aligned}$$

## Rooting out the Root

What use can we make of the program for calculating the values of  $10x^3 - 13x^2 + 19x - 25$ ? One possible use is to find the value of  $x$  which makes the expression zero – in other words to solve or find a root of the equation  $10x^3 - 13x^2 + 19x - 25 = 0$ .

You have probably often solved simple equations (e.g.  $3x - 17 = 4$ ) and quadratic equations (e.g.  $3x^2 - 12x + 7 = 0$ ). However, the above equation is a cubic equation and is much more difficult to solve by ‘ordinary’ methods.

Yet our simple program gives us one possible solution (at least approximately) straight away!

We notice that  $10x^3 - 13x^2 + 19x - 25$  is less than zero (negative) when  $x = 1.3$  and greater than zero (positive) when  $x = 1.4$ . So, somewhere in between  $x = 1.3$  and  $x = 1.4$  the expression must actually be zero.

We might be satisfied with the approximate answer 1.3. (This is slightly better than 1.4: why?) After all, we have to be satisfied with approximate answers to most questions and we usually ask for answers correct to 2 or 3 decimal places at most. However, suppose we do wish to refine the above answer and get a little closer to the true result. How can we do this?

We simply repeat the program with one small change. We alter line 130 so that it reads

FOR X := 1.3 TO 1.4 STEP 0.01 DO

The STEP has been reduced to 1/100 so that we can get an approximation to within 1/100 of the true answer. Running the program results in the following:

```
0010 // PROGRAM 45
0020 //
0030 // COMELY KATE
0040 //
0050 // TO TABULATE VALUES OF THE
0060 //
0070 //
0080 // EXPRESSION  $10x^3 - 13x^2 + 19x - 25$ 
0090 //
0100 PRINT
0110 PRINT "X           VALUE OF EXPRESSION"
0120 PRINT
0130 FOR X:=1.3 TO 1.4 STEP 0.01 DO
0140   EXPRESSION:=((10*X-13)*X+19)*X-25
0150   PRINT X,"           ",EXPRESSION
0160 NEXT X
0170 END
```

RUN

X	VALUE OF EXPRESSION
1.3	-0.3000011
1.31	6.160736E-02
1.32	0.4284782
1.33	0.8006668
1.34	1.178238

1.35	1.561247
1.36	1.949759
1.37	2.343824
1.38	2.743515
1.39	3.148884
1.4	3.559994

We could now proceed further and get to within 1/1000 of the answer by changing line 130 to read FOR X = 1.3 TO 1.31 STEP 0.001 DO.

However it is a bit of a nuisance having to change the program each time. It would be better and more natural to input the different numbers as data. This is done in the following version.

```

0010 // PROGRAM 46
0020 //
0030 // COMELY KATE
0040 //
0050 // TO SOLVE THE CUBIC EQUATION
0060 //
0070 //      3      2
0080 // 10X - 13X + 19X - 25 = 0
0090 //
0100 PRINT
0110 PRINT "PLEASE ENTER THE STARTING VALUE, THE FINISHING VALUE AND THE STEP
LENGTH"
0120 PRINT
0130 INPUT "STARTING VALUE ": START
0140 PRINT
0150 INPUT "FINISHING VALUE ": FINISH
0160 PRINT
0170 INPUT "STEP LENGTH ": STEPVAL
0180 PRINT
0190 PRINT "X           VALUE OF EXPRESSION"
0200 PRINT
0210 FOR X:=START TO FINISH STEP STEPVAL DO
0220   EXPRESSION:=((10*X-13)*X+19)*X-25
0230   PRINT X, "           ", EXPRESSION
0240 NEXT X
0250 END

```

RUN

PLEASE ENTER THE STARTING VALUE, THE FINISHING VALUE AND THE STEP LENGTH

STARTING VALUE 1.3

FINISHING VALUE 1.4

STEP LENGTH 0.01

X VALUE OF EXPRESSION

1.3	-0.3000011
1.31	6.160736E-02
1.32	0.4284782
1.33	0.8006668
1.34	1.178238
1.35	1.561247
1.36	1.949759

1.37	2.343824
1.38	2.743515
1.39	3.148884
1.4	3.559994

RUN

PLEASE ENTER THE STARTING VALUE, THE FINISHING VALUE AND THE STEP LENGTH

STARTING VALUE 1.3

FINISHING VALUE 1.31

STEP LENGTH 0.001

X VALUE OF EXPRESSION

1.3	-0.3000011
1.301	-0.2640743
1.302	-0.2280941
1.303	-0.1920624
1.304	-0.1559772
1.305	-0.1198425
1.306	-0.0836525
1.307	-4.741287E-02
1.308	-1.111794E-02
1.309	2.522659E-02

Notice that in line 210 we have used variables START, FINISH and STEPVAL to represent the lower limit, upper limit and step length of the FOR loop. This means that by merely changing START, FINISH and STEPVAL we can change the scope and sensitivity of the loop.

Notice too that we have included the nested multiplication form of the expression in line 220.

The form of the last three values of the expression may seem a bit strange, e.g. what does 2.522659E - 02 mean? In fact it means  $2.522659 \times 10^{-2}$ . Similarly 4.63E + 05 means  $4.63 \times 10^5$ . So the E indicates '10 to the power of'.

## Improve it!

Rather than having to RUN the program again and again to refine the answer we could loop the program round on itself by using a suitable REPEAT ... UNTIL structure. We would have to think of a suitable condition to terminate the REPEAT loop. You are invited to rewrite the program incorporating these improvements.

## Nested Loops

We have seen how effective the FOR ... NEXT construct is in dealing with problems which require a fixed number of iterations. So far we have used just one FOR ... NEXT loop in each program, but we can of course use more than one if the problem so requires. We turn our attention now to such problems and in particular we examine how one FOR loop may be **nested** inside another.

## Ad Astra (Seeing Stars!)

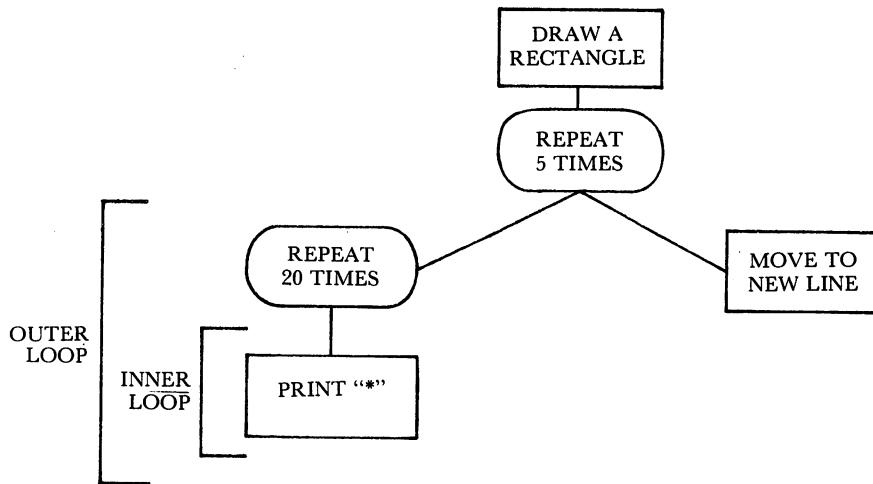
For a start, we will write a program to draw a rectangle like this. We have 5 lines of stars, so we will need a loop executed 5 times to draw these lines.

```
*****  
*****  
*****  
*****  
*****
```

Each line contains 20 stars and we will need a loop executed 20 times to draw each line.

Thus we need two counters, LINECOUNT to count the lines and STARCOUNT to count the stars within each line.

First we represent the process by drawing a structure diagram.



The outer loop is done 5 times and it completely encloses the inner loop which is done 20 times. The program reflects this structure exactly.

```
0010 // PROGRAM 47  
0020 //  
0030 // COMELY KATE  
0040 //  
0050 // TO DRAW A RECTANGLE  
0060 //  
0070 PRINT  
0080 FOR LINECOUNT:=1 TO 5 DO  
0090   FOR STARCOUNT:=1 TO 20 DO  
0100     PRINT "*",  
0110   NEXT STARCOUNT  
0120   PRINT  
0130 NEXT LINECOUNT  
0140 END
```

RUN

```
*****  
*****  
*****  
*****  
*****
```

Notice how one loop is **nested** inside the other. The inner loop is a very busy loop. It is executed 20 times for each traversal of the outer loop. Thus when LINECOUNT = 1, STARCOUNT is counted up from 1 to 20. Similarly when LINECOUNT = 2, etc.

The comma on line 100 ensures that successive stars are printed side by side. (Try the program without the comma and see what happens.) The PRINT instruction on line 120 is very important, since it causes the current line of stars to be terminated. If this were not included in the program, then when line 100 was reached again the next time through the outer loop, the next asterisk would be printed on the same line as the others and we would not get our rectangle. (Try it and see!)

It is important to observe carefully the method of nesting the FOR...NEXT loops. The STARCOUNT loop is nested inside the LINECOUNT loop. We must ensure that NEXT STARCOUNT comes before NEXT LINECOUNT, so that the correct

nested structure  is formed.

If NEXT LINECOUNT came first, the invalid structure  would be formed.

## Heavenly Triangle

No doubt you will have said to yourself that there is no need to use this nested loop structure to do the job in the last example. You are perfectly right!

We could have had just one loop (the outer loop of the program above) and printed a whole line of stars in one PRINT statement, using the fixed string “\*\*\*\*\*”, instead of using an inner loop.

However Program 47 provides a nice easy example of the nested loop structure.

There are times when we *must* use nested loops.

Consider the following task:

Write a program to print a right-angled triangle as in the sketch.

```
*  
**  
***  
****
```

Here we have a different number of stars on each line, so a single print statement using a fixed string would not work.

Let us first present the program and then offer some words of explanation.

```
0010 // PROGRAM 48
0020 //
0030 // COMELY KATE
0040 //
0050 // TO DRAW A TRIANGLE
0060 //
0070 PRINT
0080 FOR LINECOUNT:=1 TO 20 DO
0090   FOR STARCOUNT:=1 TO LINECOUNT DO
0100     PRINT "*",
0110   NEXT STARCOUNT
0120   PRINT
0130 NEXT LINECOUNT
0140 END
```

RUN

The program is almost exactly the same as in the previous example - in fact the only differences are the use of LINECOUNT instead of 20 in line 90 and the fact that the outer loop is traversed 20 times instead of 5 times in order to produce a large triangle on the screen.

The important point is the use of LINECOUNT to act as the upper limit for the inner loop each time. How does it work?

The first time through the outer loop, LINECOUNT has a value of 1. Consequently the inner loop will be done just once since the inner loop counter STARCOUNT counts just up to LINECOUNT.

The second time through the outer loop, LINECOUNT will have a value of 2 and will cause the inner loop to be done twice, thus printing two stars on the screen. And so on for all values of LINECOUNT up to 20.

You should trace the program through by hand a few times to ensure that you understand how the process works.

## Down to Earth!

To further illustrate the use of nested loops, let us write a program to draw an upside-down triangle, as shown in sketch:

```
0010 // PROGRAM 49
0020 //
0030 // COMELY KATE
0040 //
0050 // TO DRAW AN UPSIDE DOWN TRIANGLE
0060 //
0070 FOR LINE:=1 TO 22 DO
0080   FOR POINT:=1 TO 23-LINE DO
0090     PRINT "#",
0100   NEXT POINT
0110   PRINT
0120 NEXT LINE
0130 END
```

RUN

```
*****  
****  
***  
**  
*
```

Once again we have one FOR loop nested inside another and again our inner loop must be properly closed before the outer loop is closed, by having NEXT POINT before NEXT LINE.

LINE is used to count the number of iterations of the outer loop and also to set an upper limit each time for the number of traversals of the inner loop. But note how it does this. Instead of just using LINE we use the expression 23-LINE (see line 80).

When LINE = 1, 23 - LINE = 22 and so the inner loop will be done 22 times, producing a line of 22 stars. When LINE = 2, 23-LINE will be 21 and the inner loop will be done 21 times, producing a line of 21 asterisks. Thus, as LINE increases 23-LINE decreases, giving the lines of decreasing length which we require.

The fact that we have used the expression 23-LINE successfully here to specify the upper limit of our FOR variable, indicates that we are allowed in general to use expressions instead of single numbers or variables, to specify the upper limit. In fact, we may also use an expression to specify the starting value and the step. For example, the following is a valid FOR statement:

```
FOR J := A + B TO C * C - D/B STEP X - Y DO
```

## Summary

Iterations may be fixed or variable.

Fixed iterations are elegantly handled by the FOR ... NEXT construct.

A control variable is dealt with automatically in the FOR construct, i.e. initialised, changed, and tested against a limit.

Different STEP values may be used in changing the control variable.

Nested multiplication form is an efficient way of expressing polynomials.

FOR loops may be nested inside each other.

## COMAL KEYWORDS

```
AUTO CASE CAT CLEAR CON DATA  
DEL DELETE DIV DO EDIT ELIF END  
ENDCASE ENDIF FOR IF...THEN...ELSE  
INPUT LIST LOAD MOD NEW NEXT  
OTHERWISE PRINT READ RENUM REPEAT  
RND ROUND RUN SAVE STEP STOP  
UNTIL WHEN
```

## QUESTIONS and EXERCISES

1. What is the difference between fixed and variable iteration?
2. What are the four important elements in a FOR loop?
3. Write programs to compute and print the following:
  - (a) the squares of integers from -10 TO +10
  - (b) the cubes of integers from -10 TO +15
  - (c) 10!      (d) 20!

4. Write programs to find the sum of 100 terms of each of the following series:
- $1 + 3 + 5 + 7 + 9 + 11 \dots$
  - $1 + 3 + 6 + 10 + 15 \dots$
  - $1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} \dots$
  - $1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} \dots$
  - $1 - \frac{1}{2} + \frac{1}{4} - \frac{1}{8} + \frac{1}{16} \dots$
5. Write a program which calculates
- the mean of a set of 30 numbers
  - the means of 5 sets of 30 numbers.
6. Write a program to simulate the tossing of a die 100 times and find the mean of the scores. Is the answer roughly what you would expect?
7. Find the compound interest on £8,000 for 20 years at  $14\frac{3}{4}\%$  per annum.
8. Write a program to read in 50 numbers and count and print the number of positive values and the number of negative values.
9. The following sequence is the famous Fibonacci sequence: 1, 1, 2, 3, 5, 8, 13, ...  
Write a program which calculates (a) the 100th term, (b) the 1000th term.
10. Write a program to draw a triangle with its apex in the centre of the screen e.g.
- 
11. Write a program which converts Punds to Deutchmarks for values from £1 to £50 in steps of £0.50.
12. Write a program to draw a line of stars from
- the upper left hand corner of the screen to the lower right hand corner
  - the upper right hand corner to the lower left hand corner.
13. Write a program to draw the triangle shown in the diagram, with its apex in the centre of the screen.
- 
14. Write each of the following expressions in nested multiplication form:
- $4x^2 + 6x + 7$
  - $19x^3 + 11x^2 - 10x + 13$
  - $4x^3 - 7x + 3$
  - $ax^4 + bx^3 + cx^2 + dx + e$
15. Write programs to find a root of each of the following equations:
- $x^3 - x^2 + 3x - 5 = 0$
  - $17.36x^2 + 14.68x - 22.913 = 0$
16. Write a program to display a message on the screen for a certain length of time, then blank out the screen for a while, then display the message again, and so on.

# 6 Iteration III

## Repeat again

Following our extended discussion of the elegant FOR ... NEXT construct in the last chapter you may be under the impression that the best way to construct all loops in COMAL is to use FOR ... NEXT. But beware! – this is not so. Remember FOR ... NEXT is designed to deal with cases where we have a known fixed number of iterations.

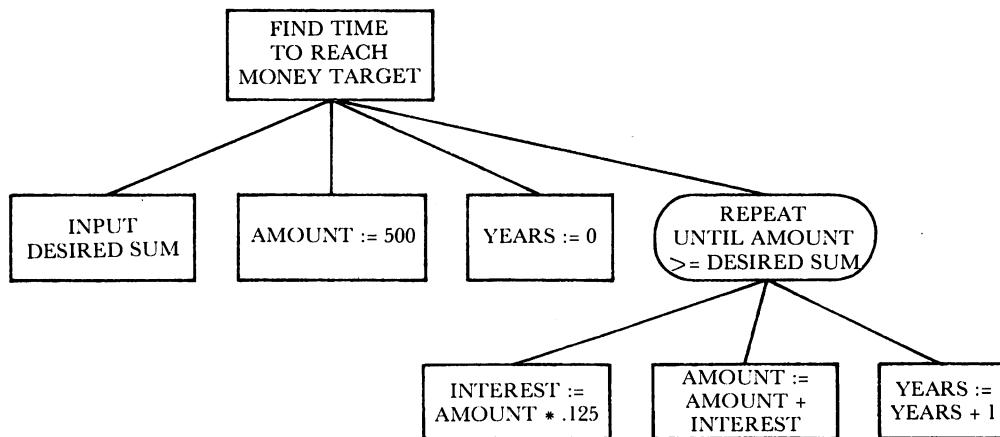
In many situations we are required to repeat a sequence of instructions until a certain condition is satisfied *but we do not know beforehand how many iterations are required*. In such situations the FOR loop cannot conveniently be used and we are obliged to consider the REPEAT ... UNTIL or some similar structure. You should now revise Programs 34 to 39 and convince yourself that 34, 35, 37 and 38 require the REPEAT ... UNTIL structure, whereas 36 and 39 could be done using FOR ... NEXT.

Let us consider another example for which the FOR ... NEXT construct is not appropriate.

## Interesting for a time

**Problem:** How long will it take £500 to amount to a desired sum at  $12\frac{1}{2}\%$  per annum compound interest?

To begin with we present the solution in the form of a structure diagram.



Because we do not know beforehand the number of years for which the calculations must continue we cannot cast our solution in the form of a FOR loop. The following program uses a REPEAT loop.

```

0010 // PROGRAM 50
0020 //
0030 // COMELY KATE
0040 //
0050 // TO FIND TIME TO REACH MONEY TARGET
0060 //
0070 INPUT "DESIRED SUM ": REQUIRED
0080 PRINT
0090 AMOUNT:=500
0100 YEAR:=0
0110 REPEAT
0120   INTEREST:=AMOUNT*0.125
0130   AMOUNT:=AMOUNT+INTEREST
0140   YEAR:=YEAR+1
0150 UNTIL AMOUNT>=REQUIRED
0160 PRINT "IT WILL TAKE ",YEAR;" YEARS BEFORE AN INVESTMENT OF $500 EQUALS OR
EXCEEDS $",REQUIRED
0170 END

RUN

DESIRED SUM 4000

IT WILL TAKE 18 YEARS BEFORE AN INVESTMENT OF $500 EQUALS OR EXCEEDS $4000

RUN

DESIRED SUM 400

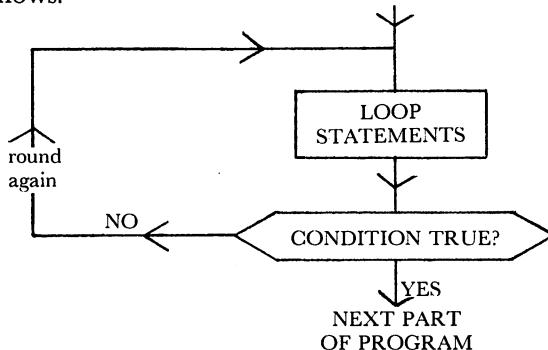
IT WILL TAKE 1 YEARS BEFORE AN INVESTMENT OF $500 EQUALS OR EXCEEDS $400

```

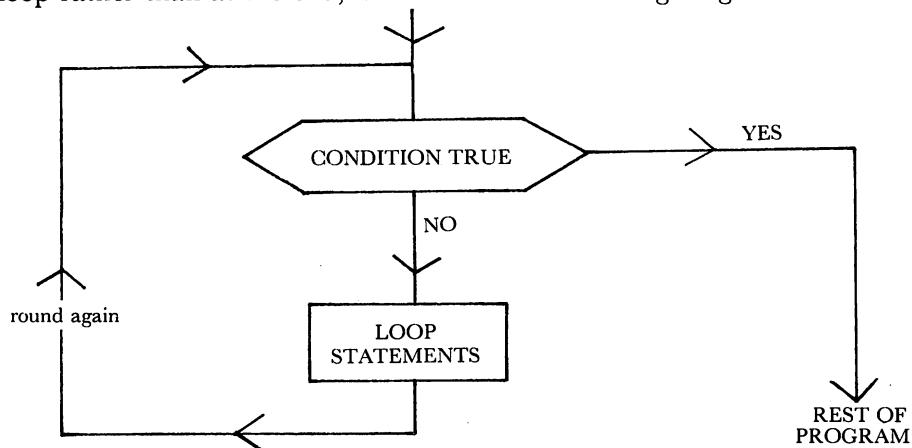
Our algorithm calculates the answer in years. Let us examine the results.

The first answer seems OK, but the answer for the second run is obviously wrong, since the original amount of £500 is already greater than the desired sum of £400 before any interest is earned. The correct answer is zero years. Why does the program produce the wrong result?

The reason is that a REPEAT loop must always be done at least once. Why is this? Because the condition which terminates execution of the loop is not tested until the end of the loop, i.e. until the UNTIL statement. Therefore even if this condition is true on first entering the loop, the loop will not be ignored since the program won't be aware of the condition until it meets the UNTIL statement. We can express the situation as follows:



So how *do* we cope with the situation where we don't know whether the loop should be done at all or not. We would obviously have to test our condition at the beginning of the loop rather than at the end, as shown in the following diagram.



COMAL has the very structure we need for this type of iteration –  
the **WHILE** loop

## Whiling away the time

Let us recast our Compound Interest example using the WHILE statement instead of REPEAT.

```

0010 // PROGRAM 51
0020 //
0030 // COMELY KATE
0040 //
0050 // TO FIND TIME TO REACH MONEY TARGET
0060 //
0070 INPUT "DESIRED SUM "; REQUIRED
0080 PRINT
0090 AMOUNT:=500
0100 YEAR:=0
0110 WHILE AMOUNT<REQUIRED DO
0120   INTEREST:=AMOUNT*0.125
0130   AMOUNT:=AMOUNT+INTEREST
0140   YEAR:=YEAR+1
0150 ENDWHILE
0160 PRINT "IT WILL TAKE ",YEAR;" YEARS BEFORE AN INVESTMENT OF $500 EQUALS OR
EXCEEDS $",REQUIRED
0170 END

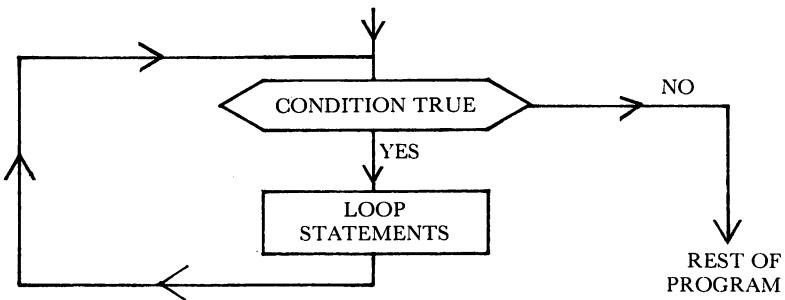
RUN
DESIRED SUM 4000
IT WILL TAKE 18  YEARS BEFORE AN INVESTMENT OF $500 EQUALS OR EXCEEDS $4000
RUN
DESIRED SUM 400
IT WILL TAKE 0  YEARS BEFORE AN INVESTMENT OF $500 EQUALS OR EXCEEDS $400
  
```

The important lines are 110 which opens the WHILE loop and 150 which closes it off. Line 110 says 'do the following instructions (down as far as ENDWHILE) as long as the value of AMOUNT is less than the value required'. But if AMOUNT is not less than REQUIRED, the loop will not be entered at all.

This is the crucial point. The WHILE loop is more general than the REPEAT loop since the REPEAT loop must be done at least once, whereas the WHILE loop may be done zero times (i.e. not at all). In fact every REPEAT loop could be recast as a WHILE loop, but not the other way round.

Why then do we bother with REPEAT at all, since WHILE appears to do an even better job? Well, REPEAT is often a more natural way to express our iterations and may make a program easier to read. Essentially, if we have the choice, we select whichever structure is clearer and easier to think about.

**N.B.** A most important point to notice is that the condition of the REPEAT loop in Program 50 is reversed in the WHILE loop in Program 51, i.e. UNTIL AMOUNT  $\geq$  REQUIRED is changed to WHILE AMOUNT  $<$  REQUIRED. This is because a WHILE loop is executed as long as a condition is true, whereas a REPEAT loop is executed as long as a condition is false (i.e. until a condition *becomes* true). We should therefore re-draw our WHILE diagram (page 87) as follows:



We have simply exchanged the words YES and NO but we must remember that the condition being tested is now the reverse of the previous condition.

Let us summarise the differences between REPEAT and WHILE:

- (1) REPEAT loop must be done at least once;  
WHILE loop may be done zero times.
- (2) Condition governing execution of loop is tested at *end* of REPEAT loop, at *beginning* of WHILE loop.
- (3) WHILE loop is done while a condition is true; REPEAT loop is done while a (reverse) condition is false.

## While we have the power

Let us exercise our knowledge of the WHILE construct on another little **problem**: Find the smallest positive power of 2 greater than a given number.

Our solution is very simple. We will simply keep multiplying 2's together until the answer exceeds the given number.

Suppose the given number is 27. Starting with  $2^0 = 1$  we keep multiplying by 2.

Thus  $2^1 = 2 \cdot 1 = 2$  which is less than 27

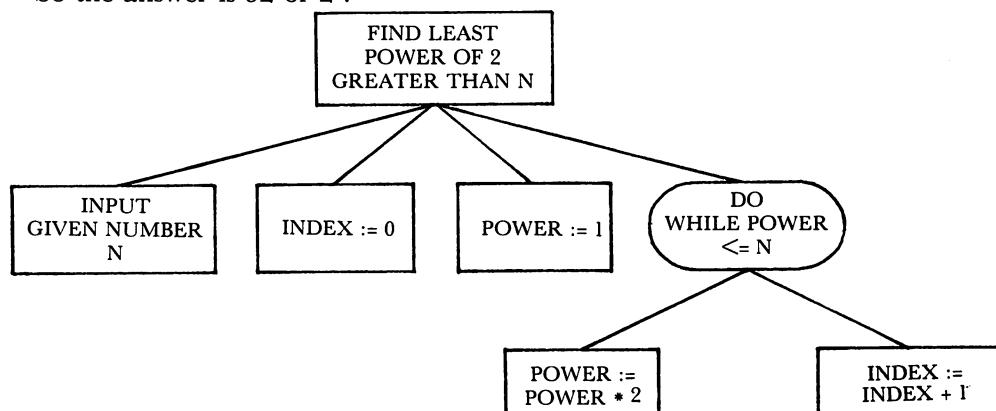
$2^2 = 2 \cdot 2 = 4$  which is less than 27

$2^3 = 2 \cdot 4 = 8$  which is less than 27

$2^4 = 2 \cdot 8 = 16$  which is less than 27

$2^5 = 2 \cdot 16 = 32$  which is greater than 27.

So the answer is 32 or  $2^5$ .



```
0010 // PROGRAM 52
0020 //
0030 // COMELY KATE
0040 //
0050 // TO FIND SMALLEST POWER OF 2 GREATER THAN A GIVEN NUMBER
0060 //
0070 INPUT "ENTER YOUR NUMBER ": N
0080 PRINT
0090 POWER:=1
0100 INDEX:=0
0110 WHILE POWER<N DO
0120   POWER:=POWER*2
0130   INDEX:=INDEX+1
0140 ENDWHILE
0150 PRINT "SMALLEST POWER OF 2 GREATER THAN ",N," IS ",POWER
0160 PRINT "IT IS 2 TO THE POWER OF ",INDEX
0170 END
```

RUN

ENTER YOUR NUMBER 800

```
SMALLEST POWER OF 2 GREATER THAN 800 IS 1024
IT IS 2 TO THE POWER OF 10
```

## Integer Variables

Up to this point we have met two types of identifiers or variable names in COMAL:

- (i) string identifiers which end in a \$ sign,  
e.g. NAME\$, ADDRESS\$, etc.
- (ii) number identifiers,  
e.g. VALUE, POWER, etc.

The number identifiers have been used for both integers (whole numbers) and decimal numbers. However it is often convenient to distinguish between integers and other numbers and COMAL allows us to do this. We can specify that an identifier is to stand for integer values only by using the # sign as the last symbol, e.g. VALUE#, INDEX#.

Since all the values, except possibly N, dealt with in the last program were integers, we could have used integer identifiers. Let us rewrite the program to show this:

```
0010 // PROGRAM 53
0020 //
0030 // COMELY KATE
0040 //
0050 // TO FIND SMALLEST POWER OF 2 GREATER THAN A GIVEN NUMBER
0060 //
0070 INPUT "ENTER YOUR NUMBER ": N
0080 PRINT
0090 POWER#:=1
0100 INDEX#:=0
0110 WHILE POWER#<N DO
0120   POWER#:=POWER#*2
0130   INDEX#:=INDEX#+1
0140 ENDWHILE
0150 PRINT "SMALLEST POWER OF 2 GREATER THAN ",N," IS ",POWER#
0160 PRINT "IT IS 2 TO THE POWER OF ",INDEX#
0170 END

RUN

ENTER YOUR NUMBER 834.67

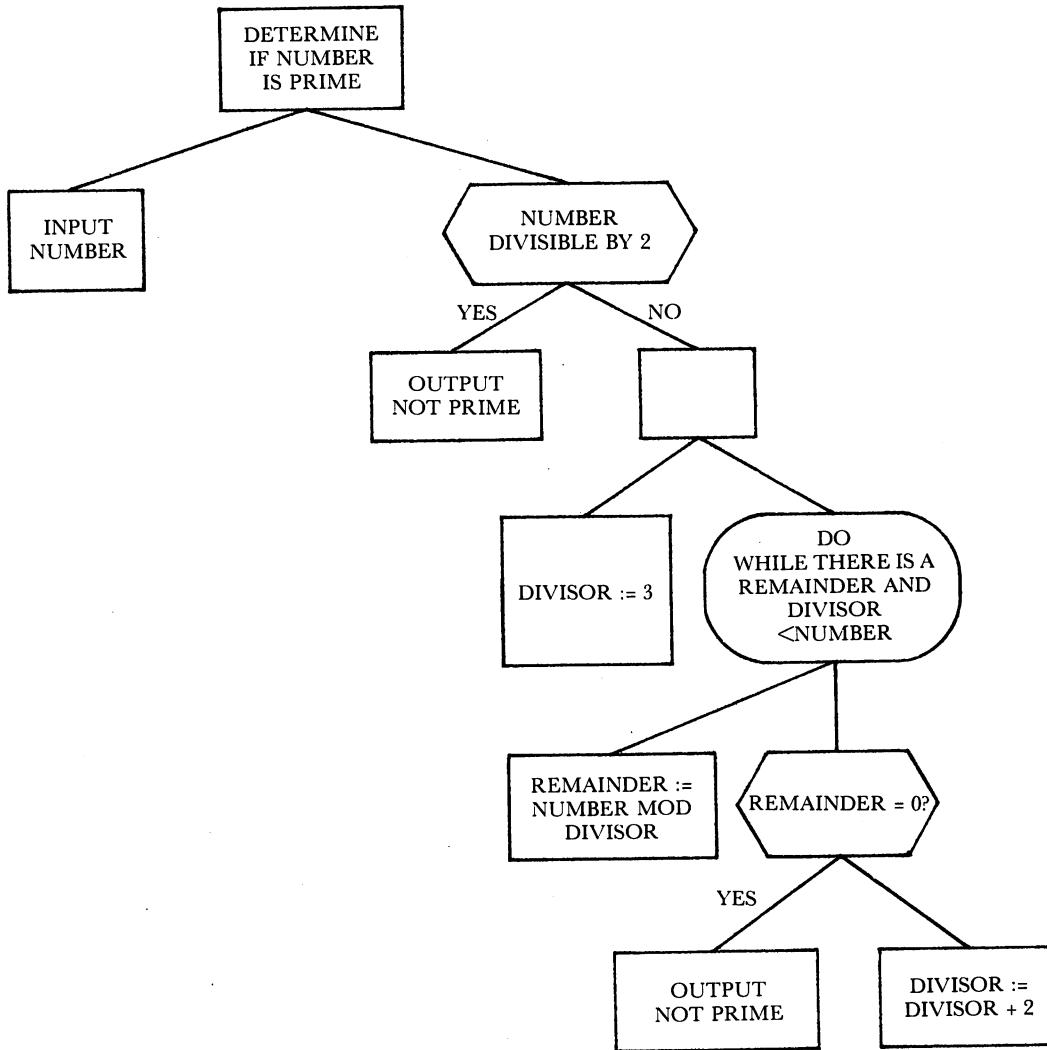
SMALLEST POWER OF 2 GREATER THAN 834.67 IS 1024
IT IS 2 TO THE POWER OF 10
```

## Prime Time

For our final example in this chapter let us consider the problem of determining whether a given positive whole number is a prime number or not. Remember a prime number is a number which cannot be divided exactly by any number except itself and 1. Thus 2, 3, 5, 7, 17, 53, etc. are prime numbers, whereas 4, 9, 15, 18, etc. are not. 2 is the only even prime number since every other even number is divisible by 2.

In order to find out if a number is prime we will use a straightforward method. We will first divide by 2 and see if there is any remainder. If there isn't we will divide by 3,

then by 5 and so on with the odd numbers (why only odd numbers?) until either  
 (a) one of these numbers divides exactly into our given number  
 (b) the divisor becomes greater than or equal to the given number.



Notice the empty box. The branch labelled NO from the previous selection box leads to more than one statement. Since we cannot put these statements side by side on the same level as the OUTPUT NOT PRIME box, we provide a connection box and develop our structure on the next level down.

```

0010 // PROGRAM 54
0020 //
0030 // COMELY KATE
0040 //
0050 // TO FIND IF A NUMBER IS PRIME
0060 //
0070 INPUT "ENTER YOUR NUMBER ": NUMBER#
0075 PRINT
0080 REMAINDER#:=NUMBER# MOD 2
0090 IF REMAINDER#=0 THEN
0100 PRINT NUMBER#, " IS NOT PRIME"
0110 ELSE
0120 DIVISOR#:=3
0130 WHILE (REMAINDER#<>0) AND (DIVISOR#<NUMBER#) DO
0140 REMAINDER#:=NUMBER# MOD DIVISOR#
0150 IF REMAINDER#=0 THEN
0160 PRINT NUMBER#, " IS NOT PRIME"
0170 ELSE
0180 DIVISOR#:=DIVISOR#+2
0190 ENDIF
0200 ENDWHILE
0210 IF DIVISOR#>=NUMBER# THEN
0220 PRINT NUMBER#, " IS PRIME"
0230 ENDIF
0240 ENDIF
0250 END

```

RUN

ENTER YOUR NUMBER 93

93 IS NOT PRIME

RUN

ENTER YOUR NUMBER 97

97 IS PRIME

A few comments are in order.

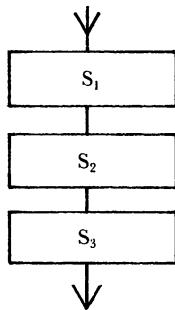
- (1) Note the span of the ELSE statement on line 110. It governs all the instructions from line 120 to 240.
- (2) The ENDIF on line 240 closes off the IF on line 90. The ENDIF on line 230 closes IF on line 210.
- (3) The WHILE condition on line 130 is in fact a double condition. The two conditions
  - (a) REMAINDER <> 0
  - (b) DIVISOR < NUMBER
 are joined by the word AND. Both conditions must be true for the WHILE loop to be done. If one of the conditions fails, the WHILE loop will not be re-iterated.
- (4) By starting the DIVISOR at 3 before the loop and increasing it by 2 on line 180 we use only the odd numbers as divisors. We treat 2 as a special case at the beginning of the program.
- (5) When the iteration governed by the WHILE condition ends we know that one or other of the conditions must now be false. The program tests to see if in fact the

divisor has got up as far as the number itself and if it has prints out that the number must therefore be prime. If the other condition is false the program will already have printed out that the number was not prime and so no further action needs to be taken.

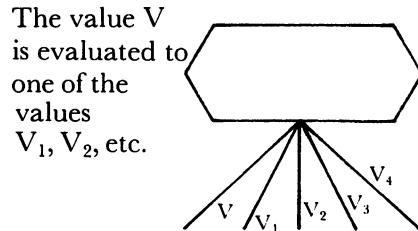
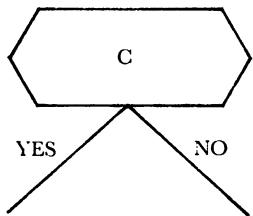
- (6) We have used the integer operator MOD in the program. You will remember that MOD finds the remainder when the number on the left of MOD is divided by the value on the right.
- (7) We have used integer identifiers throughout.

We have now studied the third of the three fundamental programming structures, viz looping or iteration. Thus we now have at our disposal -

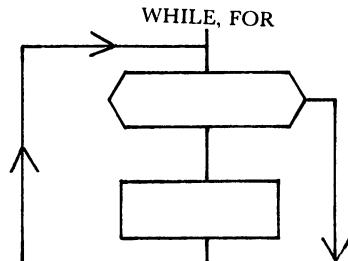
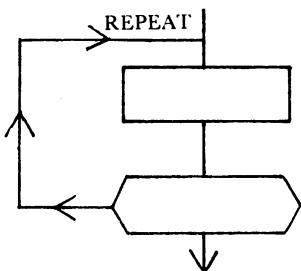
- (1) Sequencing**  
Statements  $S_1$ ,  $S_2$ ,  $S_3$ , etc.  
are done in sequence.



- (2) Selection**  
The condition C is tested to give the answer YES or NO.



- (3) Iteration**



A most important observation is that for each structure (even the selection structures) there is a *single point of entry* and a *single point of exit*.

## **Summary**

It is necessary to take into account that sometimes iterations may be done zero times. WHILE allows for this.

WHILE is more general than REPEAT, but we choose whichever structure is the more apt for our particular problem.

There are some significant differences between the WHILE and REPEAT constructs.

Sequence, Selection and Iteration are the three fundamental programming constructs.

Integer variables are distinguished by a # at the end.

### **COMAL KEYWORDS**

AUTO	CASE	CAT	CLEAR	CON	DATA	DEL
DELETE	DIM	DIV	DO	EDIT	ELIF	
END	ENDCASE	ENDIF	ENDWHILE	FOR		
IF...THEN...ELSE	INPUT	LIST	LOAD	MOD		
NEW	NEXT	OTHERWISE	PRINT	READ		
RENUM	REPEAT	RND	ROUND	RUN		
SAVE	STEP	STOP	TAB	UNTIL	WHEN	
WHILE						

## **QUESTIONS and EXERCISES**

1. What are the differences between WHILE and REPEAT loops? Why is WHILE more general than REPEAT?
2. Draw structure diagrams for the programs which are not accompanied by structure diagrams in this chapter.
3. Write programs to determine how many terms of each of the following series are required to exceed a given number in each case.
  - (a)  $1 + 3 + 5 + 7 + 9 + \dots$
  - (b)  $1 + 3 + 6 + 10 + 15 + \dots$
  - (c)  $1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots$
  - (d)  $1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \dots$
  - (e)  $1 + 1 + 2 + 3 + 5 + 8 + 13 + \dots$

4. Find the ratio of successive terms of the Fibonacci sequence 1, 1, 2, 3, 5, 8, 13, ...  
The program should iterate until there is no significant difference between the ratios.
5. Write a program to determine what % interest must be earned for a sum of money to double itself in ten years.
6. Re-write the programs given as exercises at the end of Chapter 4, using the WHILE construct.
7. Write a program to find the cube root of a number.
8. Write a program to convert Roman numerals to ordinary decimal notation.
9. In some COMAL systems if a WHILE loop contains just one statement the ENDWHILE may be omitted e.g.

```
10  SUM:= 0  
20  WHILE SUM<= 100 DO SUM := SUM + 5  
30  PRINT "ANSWER =", SUM  
40  END
```

Note that the statement is on the same line as the WHILE ... DO.

What does this program do?

If your system allows this kind of WHILE write programs incorporating 'short' WHILE statements to

- (a) add the first 50 odd positive integers
- (b) Find how many terms of the series  $1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots$  are required to give 3.

# 7 Hardware

## INPUT - PROCESS - OUTPUT

Now that we have begun to learn the art of communicating with that dumb but marvellous machine called a computer, let us make a brief tour of inspection of the actual machinery itself.

Every computer must have input, output and processing equipment. Let us begin by examining some input and output machinery.

### Input Devices

The computer you have in your school probably has a typewriter-like keyboard through which it receives information from you (INPUT) and it probably uses a screen (or visual display unit - VDU) to send information back to you (OUTPUT). It may in fact use an ordinary UHF television set as a visual display unit. However, there are other means of entering information into the computer and receiving information back from the computer. We will take a look at some of these now. Just before doing that let us consider what happens when we press one of the letters or other characters on the keyboard.



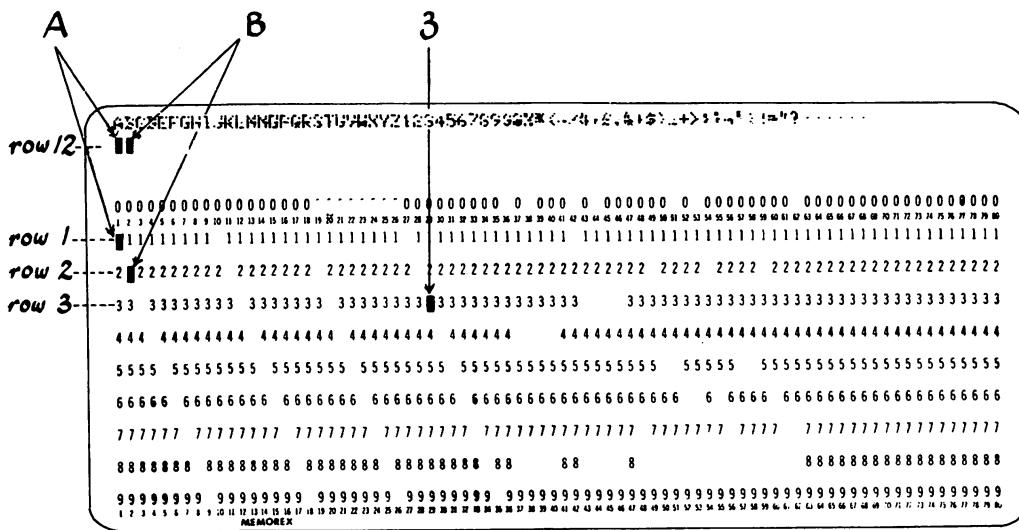
## Getting through

When a keyboard key is pressed, an electrical contact is made. This causes a set of electrical pulses to be generated and transmitted to a special place inside the computer. Each key gives rise to a different pattern of pulses, called the **code** for the character corresponding to the key. Different codes have been used to represent information in computers. The most widely-used code at present is probably the ASCII Code. ASCII stands for American Standard Code for Information Interchange.

The information represented by the ASCII codes for the different characters is stored in the computer's memory. We will have more to say about this essential component of every computer later.

## Punched Cards and Card-Readers

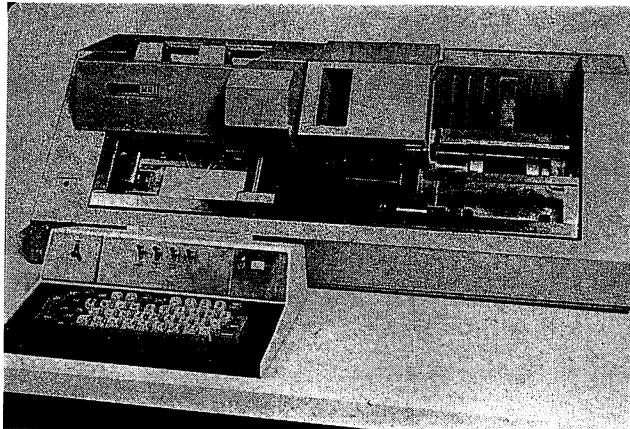
The punched card was perhaps the earliest medium used to feed information to computers. It is still widely used throughout the world.



Punched card

Information is coded as patterns of holes in a card. Each character (letter, digit or other symbol) has its own code. For example, the digit 3 is represented by a single hole in the row numbered 3 (see illustration). Similarly each of the other digits is coded by a single hole in the appropriate row. The letter A is coded by a pattern of two holes,

Card-punch machine



one under the other in rows 1 and 12 (the top row). B is coded by two holes in rows 2 and 12 respectively, and so on for other letters. Characters other than digits and letters are usually represented by 3 holes in a column, e.g. the question mark, ?, is represented by holes in rows 0, 7 and 8.

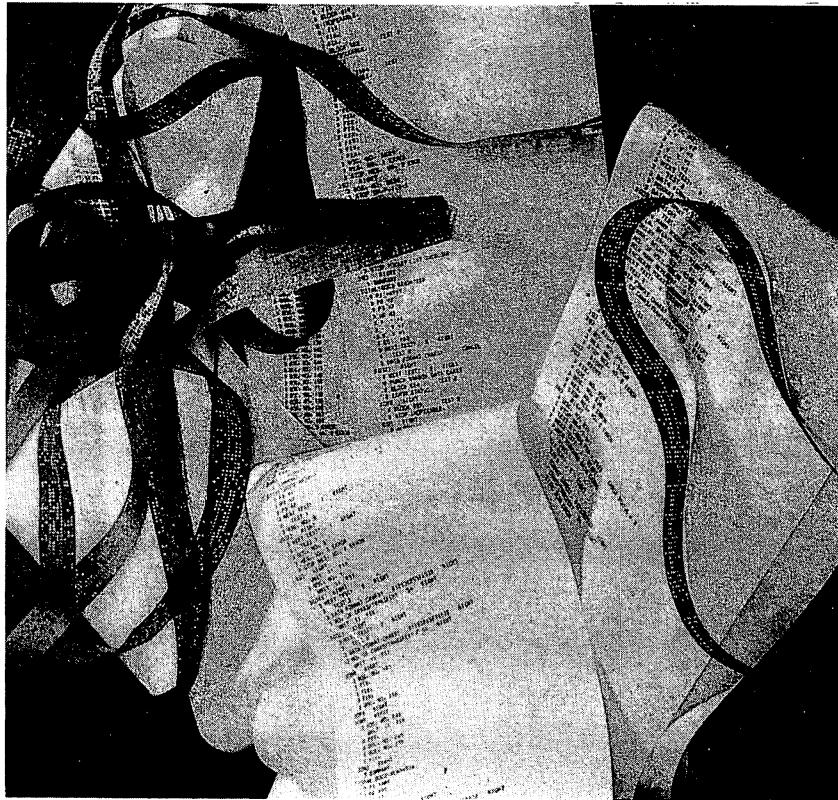
A card-punch machine is used to put the symbols onto the cards. It has a keyboard similar to a typewriter. When a key is pressed the appropriate pattern of holes is automatically punched on the card. Very often the symbol itself is also printed at the very top of the card.

The information on the cards is fed into the computer by means of a card-reader. In the card-reader, the cards are placed in a hopper. From the hopper they pass at high speed through a reading unit where beams of light pass through the holes in the card and fall on light-sensitive cells. The cells convert the light into electrical signals which transmit the information into the computer's memory.

## Paper Tape

Paper tape can also be used to hold coded information and feed it to the computer. It is similar to punched cards in that holes are punched in the tape to represent letters, digits and other characters.

A paper tape reader is used to transmit the information on the tape to the computer. As in the case of punched cards, the tape is read by passing it between a light source and a collection of photo-electric cells.



## **Teletype**

A teletype is a sophisticated typewriter device, through which information can be fed directly to the computer and which can accept information output by the computer. Very often the teletype can read and punch paper tape. If this is the case, the teletype can be set to read a tape and transmit information automatically to the computer, or to receive information automatically from the computer and punch it onto the tape.

## **Optical Character Recognition (OCR)**

Optical readers read print of special type. Each character is formed by a unique combination of black and white areas which give rise to a unique electrical signal when read by photo-electric means. Optical character readers still tend to be rather expensive.

## **Magnetic Ink Character Recognition (MICR)**

Magnetic ink encoding of characters is widely used on cheques. It has the advantage of being legible to humans and also readable by a computer. Each character is printed in an ink which has special magnetic properties. These induce electrical signals when the document is passed through an MICR reader, and these signals carry the information into the memory of the computer.

Documents can be read at a very high speed - up to 1200 per minute. This is essential in order to process the very large number of cheques which pass through the banking system each day.

# **Output Devices**

## **Visual Display Unit (VDU)**

You are probably already familiar with a form of Visual Display Unit used by the computer in your school. In general a VDU serves as both an input and output device and is equipped with a keyboard for input purposes.

The output from the computer is displayed on a screen in the form of lines of text. Microcomputer VDU's typically display about 24 lines of text on a full screen, each line containing 40 characters. Each character is formed by a matrix of tiny dots on the screen.

One of the principal advantages of a VDU is its speed of response. A full screen of information can be displayed in a fraction of a second, whereas with mechanical output devices the speed is limited by the printing mechanism.

## **Graphic Display Device**

Graphic Display terminals are special forms of visual display units which are used to present information in graphical or pictorial form. They are usually used interactively, which means that the person sitting at the terminal can actually modify the drawings and have the changes recorded and stored by the computer.

A device known as a *light pen* is used to alter the diagrams. When the pen touches the screen it emits an electrical pulse which is recorded by the computer as defining the position of a point to be drawn. A moving light pen can trace out diagrams.

VDU's as described in the first section can be used to display pictorial information, but there is no facility to change the pictures by means of a light pen.

## Graph-Plotters

Graph-plotters are used to print drawings or graphs on paper, from information stored in the computer. Moveable pens (possibly in different colours) move over paper mounted on a cylindrical roller. Accuracy to within hundredths of a centimetre can be achieved.



CALCOMP  
Graph-Plotter

## Printer

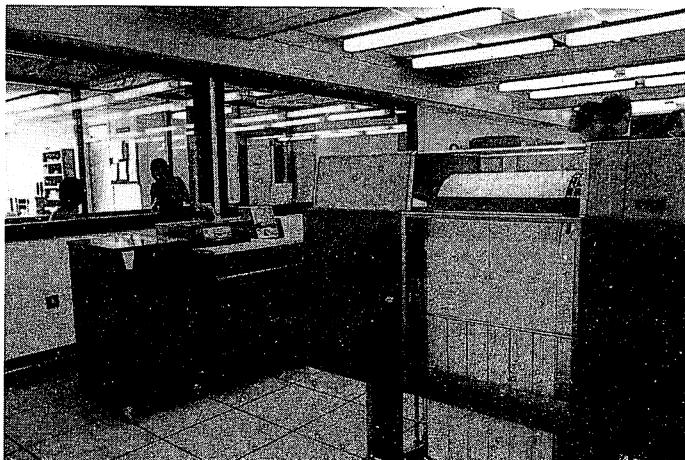
Printers are extremely important output devices because they provide permanent records of programs and results which can be read by people. This form of output is known as **hard copy**. There are many different kinds of printer.

- (a) Computer-operated electric typewriters which use cylindrical or 'golf' ball printing heads and print somewhat slowly - 10 to 30 characters per second.
- (b) Daisywheel printers where the print characters are arranged as a wheel of light 'petals'. These petals are struck by a hammer when the wheel rotates into position. Speeds of about 50 characters per second are typical.
- (c) Matrix printers where the printing is done by a column of styli or needles. These printers can operate at high speed.

## Line Printers

High-speed printers print a full line of up to 150 characters at a time and are used to output large quantities of information. Speeds of up to 2,000 lines per minute can be achieved. Among the important types of line printers are the following:

- (a) *Drum Printers* which have the print characters embossed on a continually rotating drum. Hammers strike the paper against a carbon ribbon which in turn is pressed against the drum.
- (b) *Chain Printers* where the characters are embossed on a band or chain which moves at right angles to the direction in which the paper moves.
- (c) *Heat Printers*: these are matrix-type printers which use electric sparks instead of needles to burn the dots forming the characters onto the paper. The lines printed are usually fairly short, typically about 30 characters.
- (d) *Xerographic Printers*: here the text is deposited electrostatically on the surface of a rotating drum. Powdered ink is then transferred to the drum by electrostatic attraction from which it is printed on the paper. Xerographic printers operate at very high speeds - up to 20,000 lines per minute - but are rather expensive.



Card-reader  
and Line-printer

Other output media which are used in many applications are

Punched Cards  
Paper Tape  
Microfiche - much used by banks  
Microfilm

Input/output devices are often referred to as 'peripheral devices'.

## **Storing of information**

Having seen how information may be got into and out of a computer let us turn our attention now to the storing of information by the system.

## **Coding of Information**

When we communicate with each other we use a language of some kind. Normally it is a language like Irish or English, but sometimes it may be Morse Code, or a sign language as used by the deaf and dumb, or a language of touching. Even a language like English, of course, uses symbols or signs to represent information and we may consider English words to be codes to represent objects and ideas. Thus the word 'TABLE' may be thought of as a code for the concept *table*.

Information is represented in computers by means of codes also. What code does a computer use?

Computers use binary codes to represent information. These are codes which use just two symbols, 0 and 1. The ASCII Code, which we have already mentioned, is a binary code.

The ASCII Code for A is 1000001.

The ASCII Code for \$ is 0100100.

Isn't it amazing that with just two symbols we can encode all the information in all the books ever written!

The two symbols 0 and 1 are known as **binary digits**, or **bits** for short.

Bits are preferred for the presentation of information in a computer for several reasons:

- (1) They can be easily represented by electrical pulses or magnetic states.
- (2) They allow simple electronic circuits to be designed.
- (3) They enable more reliable and accurate circuits to be used.
- (4) They reduce arithmetic and other information-processing to very basic forms.

## **Computer Memory**

A computer needs a memory to store the information encoded in binary form. In fact we distinguish between two kinds of memory:

- (1) primary or main memory which is used to hold the programs and data on which the computer is currently working.
- (2) secondary memory, or back-up memory, which is used to store information that is not immediately being used. We will consider this backing store in a later chapter.

## Some Important ASCII Codes

Character	Decimal Code	Binary Code	Character	Decimal Code	Binary Code
BEL	7	0000111	=	61	0111101
Space	32	0100000	>	62	0111110
!	33	0100001	?	63	0111111
"	34	0100010	@	64	1000000
#	35	0100011	A	65	1000001
\$	36	0100100	B	66	1000010
%	37	0100101	C	67	1000011
&	38	0100110	D	68	1000100
,	39	0100111	E	69	1000101
(	40	0101000	F	70	1000110
)	41	0101001	G	71	1000111
*	42	0101010	H	72	1001000
+	43	0101011	I	73	1001001
,	44	0101100	J	74	1001010
-	45	0101101	K	75	1001011
.	46	0101110	L	76	1001100
/	47	0101111	M	77	1001101
0	48	0110000	N	78	1001110
1	49	0110001	O	79	1001111
2	50	0110010	P	80	1010000
3	51	0110011	Q	81	1010001
4	52	0110100	R	82	1010010
5	53	0110101	S	83	1010011
6	54	0110110	T	84	1010100
7	55	0110111	U	85	1010101
8	56	0111000	V	86	1010110
9	57	0111001	W	87	1010111
:	58	0111010	X	88	1011000
;	59	0111011	Y	89	1011001
<	60	0111100	Z	90	1011010

## Main Memory

The main memory of a computer is often called its 'immediate access store' because information can be placed in this memory or retrieved from it extremely quickly. Putting data in memory is called 'writing to memory' and retrieving data is called 'reading from memory'.

The memory may be considered to consist of many thousands of locations, each capable of holding a pattern of binary digits or bits.

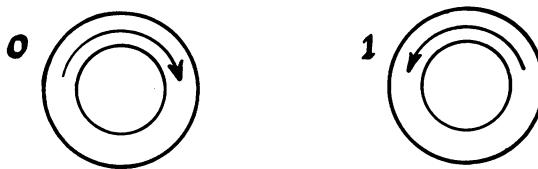
Main Memory	
Address	Contents
0	00010110
1	01010001
2	01000001
3	
4	
5	
6	
7	

Each location has an 'address' which gives its position within the memory. If there are 8192 such locations the addresses would run from 0 to 8191. Memory locations are often called memory 'words'. However if the location holds 8 binary digits, it is more usual to call it a **byte**. Thus a byte is an eight-bit word.

The size of main memory is usually expressed in terms of blocks of 1024 words. Memory containing 1024 words is called a **1K Memory**. 2048 words constitute a **2K Memory**, 8192 words constitute an **8K Memory**, and so on.

Up to the mid 1970's, main memory consisted of tiny rings of magnetic material called cores. These cores could be magnetised in two opposite directions. One direction represented 0; the opposite direction represented 1.

Magnetic Cores



A memory byte consisted of eight of these tiny cores, with conducting wires running through them.

Nowadays, particularly in microcomputers, memories are made of tiny electronic elements and are known as semiconductor memories. A popular form is metal oxide on semiconductor, known as MOS memory. A major difference between semiconductor and core memories is that when the power to the machine is switched off, core memories retain their information, whereas information is immediately erased from semiconductor memories. However, besides the volatile semiconductor memory, known as RAM (**Random Access Memory**), microcomputers also have a small amount of non-erasable memory known as ROM (**Read Only Memory**). The reason this type of memory is called ROM is because information can only be read from it, and not written to it.

ROM memory is vitally important because it contains information necessary for the proper operation of the computer. For example it may contain vital information for the understanding and execution of COMAL programs.

The use of the terms RAM and ROM in connection with computer memory is a little misleading. Both types of memory are random access memories in that any location can be accessed almost immediately at random. So ROM does not mean that the read-only memory is not random access. Perhaps it would have been better to keep the name ROM but use RWM (Read Write Memory) instead of RAM.

## **Secondary Memory**

We have seen that a storage facility or memory is a vital component of a computer system. Programs and data which are being processed are held in the main memory. Apart from the information held in ROM, the data held in the main memory of microcomputers is lost when the power is switched off. However, we have seen that programs can be saved on cassette tape or magnetic disk. Tape and disk form two types of secondary memory or backing store.

Any medium which is capable of holding information permanently can effectively be used as backing store. However, it is usual to use the term only in connection with computer readable media such as

- punched cards
- punched paper tape
- microfiche
- magnetic tape
- magnetic disk, etc.

Magnetic tape and magnetic disk are by far the most important kinds of secondary memory because of (a) their speed and (b) their capacity.

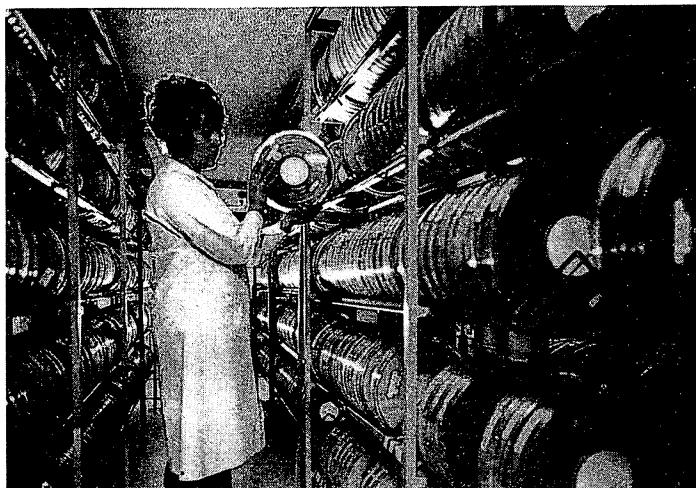
## **Magnetic Tape**

On large computer systems large reels of tape, perhaps 700m long, are used, while on microcomputers the smaller cassette tapes have become universally popular. A typical large tape reel can contain as much information as is contained in several copies of the whole Bible.

Magnetic tape is made of polyester plastic coated with magnetic oxide. Data is recorded on the tape by magnetising tiny areas to represent binary digits. Magnetisation in one direction represents 0 and in the opposite direction represents 1. The data is recorded in blocks with a gap between the blocks to allow the tape to speed up to full speed for correct reading and writing.

Tapes are known as serial devices. In order to access information on the tape all the data prior to the required information must be read. Data cannot be accessed directly or randomly.

Data can be transferred to or from tape at speeds up to a million bits per second.



Tape library

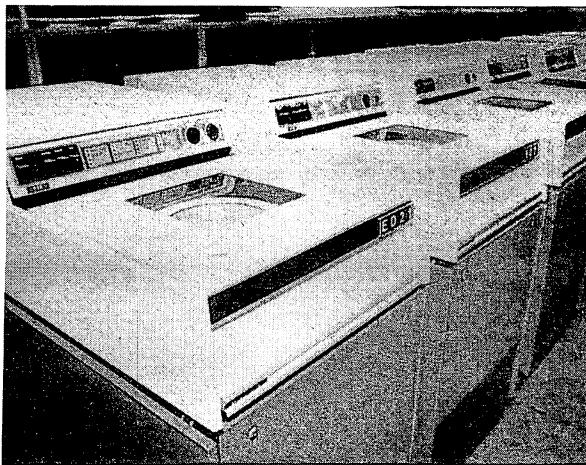


Paper Tape Reader

## Magnetic Disk

Disks are known as 'direct access devices' because data can be written directly to or read directly from any position on a disk. There is no need to work through from the beginning to the required information. Small flexible disks known as 'floppy disks' are usually used with microcomputer systems, while larger hard disks are used in mini or main-frame systems.

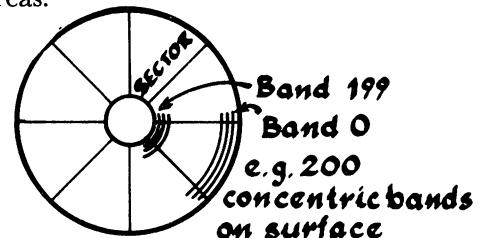
When a disk is in use it is kept rotating continuously and a read-write head moves in and out over the surface of the disk. With most disk drives the head does not touch the



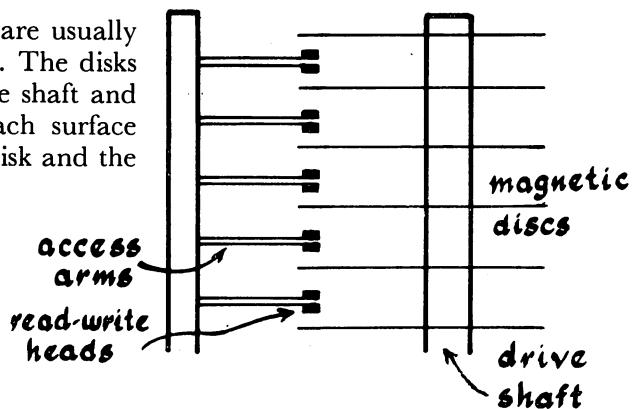
Disk Drives

surface but rides on a very thin film of air. It is used to form tiny areas of magnetisation which record data bits, or to read from such areas.

A disk surface is usually divided into a number of concentric bands, which in turn are divided into a number of sectors. A sector is often used to hold a block of data.



On larger systems several disks are usually used together to form a disk pack. The disks rotate together on a common drive shaft and there is a read-write head for each surface except the top surface of the top disk and the bottom surface of the bottom disk.



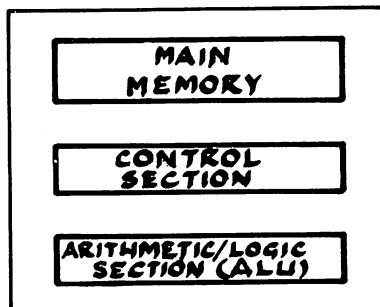
Data can be transferred to or from a disk at a speed of several million bits per second.

Note that in addition to serving as back-up memory, tape and disk systems can also serve as input/output devices.

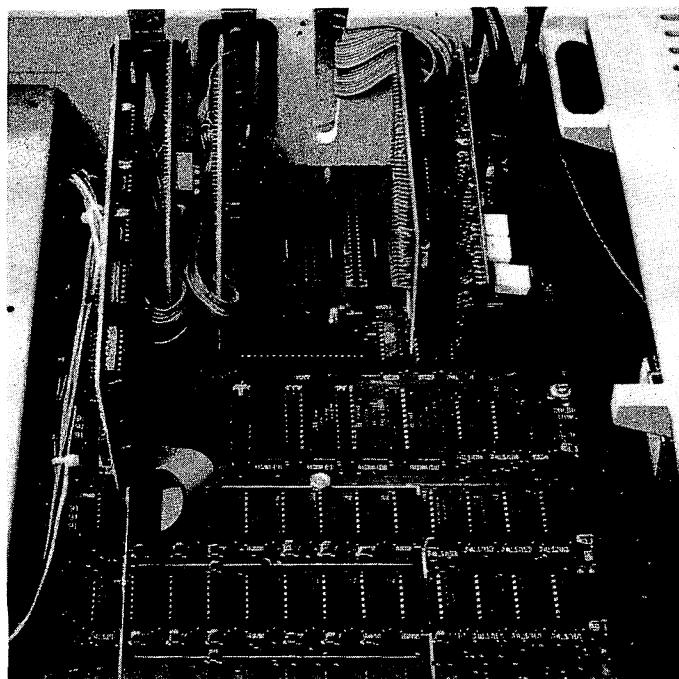
## Processing of information

Once information is stored in the main memory of a computer it may be processed. We examine now some of the hardware involved in this processing.

The Central Processing Unit or CPU is the part of the computer where all the arithmetic and logical processing is done. It may be represented by the following outline diagram:



The main memory is often physically separate from the other sections, although it is logically intimately connected to them in contributing to the overall performance of the computer. The main memory is where the information being processed - instructions and data - is kept.



Microprocessor board

In a modern microcomputer the processing unit, known as the microprocessor, is housed in one semiconductor chip, while the main memory resides on a number of other chips.

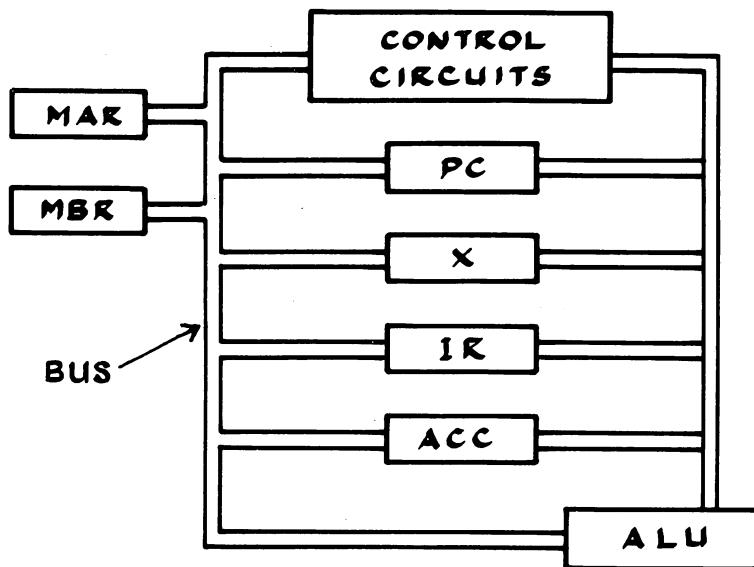
The memory is connected to the other sections of the computer by means of conducting highways known as **buses**. The different parts of the processor are likewise connected by buses. Information flows along these buses from one location to another.

The **Control Unit** has the vital job of controlling and co-ordinating the entire work of the computer. It must

- (1) oversee the transmission of signals to and fro within the computer
- (2) interpret instructions extracted from memory
- (3) direct the results of instruction interpretation to other components for further processing
- (4) co-ordinate and synchronise various activities by means of an electronic clock.

The **Arithmetic Logic Unit (ALU)** is where the arithmetic and logical operations required to carry out program instructions are performed. We have already seen examples of arithmetic processing. Later we will see how logical operations play their part in helping to solve problems.

Outline of CPU



In the above diagram we have indicated in simple outline some of the details of the important components of a computer's processing unit.

**Registers** are used to store information temporarily while it is being processed or used in some way. Some of the important registers are:

*Program Counter Register (PC)*: this is used to hold the ‘address’ of the location in memory of the next instruction in a program, so that it can be fetched from memory when the time comes.

*Index Register (X)*: the contents of the index register are sometimes used to help in calculating the address of the memory location where data is stored. There may in fact be more than one index register available in a processor.

*Accumulator*: this is used to hold one piece of data while another piece is added to it, or subtracted from it, etc.

*Instruction Register (IR)*: this is used to hold an instruction which has just been retrieved from memory, while it is being decoded and interpreted.

*Memory Address Register (MAR)*: when information is to be retrieved from or sent to the main memory, the address of the location to be used is sent down along the bus lines to the memory from the MAR.

*Memory Buffer Register (MBR)*: when information is retrieved from memory, it is first placed in the Memory Buffer Register before being despatched to the accumulator, or Instruction Register, or wherever it is to go. Similarly information to be stored in memory is first placed in the MBR before being sent along the bus to the memory.

An important property of registers is that information can be placed in them and read from them at high speed. They serve a vital role in the processing activities of the CPU.

## Summary

A computer system consists of input, output and processing machinery.

Both immediate and back-up or secondary memories are required.

Magnetic tape and disk systems can serve as secondary memory and as input/output devices.

Among a wide range of input devices are

Punched Card Readers	OCR Readers
Paper Tape Readers	Magnetic Tape Drives
Teletypes	Disk Drives
MICR Readers	

Among a wide range of output devices are

VDU's	Printers
Graphic Display Devices	Magnetic Tape
Graph Plotters	Disk Drives

The ASCII Code is a widely used code for representing information in computers.

A bit is a binary digit, 0 or 1.

A byte is an 8-bit unit.

Memory is usually expressed in multiples of 1K = 1024 bytes or words.

ROM means Read Only Memory.

RAM means Random Access Memory, which can be read from and written to.

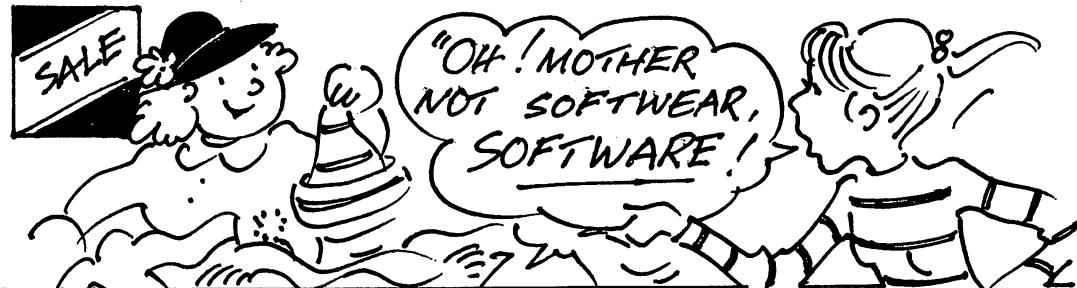
The Central Processing Unit (CPU) is the busy 'brain' of the computer. It contains the important Control Unit and Arithmetic Logic Unit.

Registers are high-speed memory cells which are of vital importance in the activity of the CPU.

## **QUESTIONS and EXERCISES**

1. Write a short description of each input and each output device listed in this chapter.
2. Nowadays many computer magazines and papers are published. Perhaps your school gets one. If it does, see how many advertisements for printers you can find. How many different kinds of printer are advertised? How fast can each operate?
3. Approximately how long would it take a daisywheel printer, operating at 50 characters per second, to print this book? How long would it take a line printer, operating at 2000 lines of 132 characters each per minute, to do the same job?
4. How many characters can be stored in a 48K memory? What is the purpose of dividing memory into ROM and RAM?
5. Find how much data can be held on (a) a cassette tape, (b) a 5" floppy disk.
6. Find out how fast information may be transferred from the disk drive to the main memory of the computer in your school.
7. What are the disadvantages of using punched cards as back-up memory?
8. Explain the role played by each of the important registers mentioned in this chapter.
9. Discuss the function of each of the main components of the CPU of a computer.

# 8 Software



## Software

We have already used the term **hardware** to describe the actual machinery used by a computing system, e.g. keyboard, printer, electronic circuits, etc. But what is **software**?

**Software is programs.**

You will recall that we mentioned earlier that ROM memory was important because it contained information vital to the operation of the computing system. Such information consists mostly of programs or software. Normally when we use the term *software* to refer to programs we mean *the important programs used by the system to carry out its work*, rather than the programs which you and I write, which are usually known as 'applications programs'. Let us briefly consider some of this important systems software.

## Compilers

We have seen that information is represented inside the computer in binary form. The computer 'understands' and executes instructions, only if they are expressed in binary code. This implies that it cannot understand instructions in a high-level language like COMAL. So how does it get to grips with high-level instructions?

The answer is that these instructions must be translated into binary code before they are executed. This translation is usually done by a program called a compiler. It translates from high-level down to a lower level, from whence it is further translated eventually to binary form.

On large computer systems there are **compilers** for many different languages, e.g. Pascal, FORTRAN, COBOL, ALGOL etc. COMAL, oddly enough, is not usually compiled by a compiler, but is translated by another type of translation program called an 'interpreter'.

## Interpreters

The main difference between a compiler and an **interpreter** is that, whereas a compiler translates a whole program completely to a lower form before it is executed, an interpreter leaves the program in source form (or an intermediate form close to source form) and just translates each statement when it is required to be executed. An interpreter may, in fact, translate the same statement many times, e.g. in a loop.

Interpreters usually cause programs to run more slowly than compilers, but allow better notification of errors.

## Assemblers

**Assemblers** are like compilers, except that they do not translate high-level languages, but instead operate on a lower level language to reduce it to machine code. Such low-level languages are called 'assembly languages'. They can be fairly easily read and understood by humans, but at the same time they have instructions which are close to machine instructions. You may already have studied an assembly-type language, CSSPK. A typical small assembly-language program might be as follows:

```
START
READ      A
READ      B
CLEAR    ADD A
ADD       B
STORE     SUM
WRITE     SUM
STOP
```

## Operating systems

An operating system consists of a suite of complex programs which coordinate and control all the different activities which take place in a computer system. It is designed to maximise the overall performance of the computer system and to enable users to run their programs without dealing with or knowing anything about the electronic details of the hardware.

The following is a list of some of the tasks which an operating system may perform:

- (1) supervise and control the running of a batch of programs in succession, without requiring the intervention of a human operator
- (2) share the resources of the computer system among a number of users linked to the machine by means of terminals
- (3) call into action different compilers, assemblers, etc., as they are needed by different programs

- (4) provide a library of utility programs for calculating SIN, COS, RND, etc.
- (5) allow a number of programs to reside in memory simultaneously and keep track of where each program is stored and what each program is doing
- (6) supervise and control the reading in and outputting of information from different I/O devices such as card readers, keyboards, tapes, disks, printers, etc.
- (7) manage all the information files used by the system and the programmers
- (8) keep an account for each user, so that he may be charged correctly for the amount of time and resources used
- (9) keep a log of the activities of the computer system
- (10) allow access to system from remote terminals miles away, perhaps in other countries
- (11) schedule the use of the different resources by, for example, queueing jobs waiting to use the line printer
- (12) ensure that confidential data is kept securely and allow only authorised access.

On a large computer system an operating system can be quite a complex affair, with so much housekeeping and resource management to take care of. Even on a small microcomputer there will be an operating system, which is sometimes known as a **monitor**.

If your system has a disk drive, you will probably find that in order to use the drive the computer has at least a Disk Operating System, often called DOS for short.

One of the most popular operating systems for microcomputers is CP/M (Control Program/Microprocessors). The COMAL system which was used to test the programs in this book was designed to work in conjunction with the CP/M operating system.

## Multiprogramming

If you look back at (5) in the list above, you will see that an operating system may allow a number of programs to reside in main memory and to be executing simultaneously. In fact they wouldn't actually be running simultaneously. The operating system would cause the computer to switch from one to the other, running each program for a while before moving on to the next. Why should the computer bother to do this?

The answer is: to make more efficient use of the system resources.

The CPU of a computer works at very high speeds and may be able to perform many thousands of calculations while you are typing a single letter on the keyboard. It would be grossly inefficient in a large system with many jobs to do, to sit there waiting for human input. So, if a job requires input from a peripheral device before it can continue, then the CPU is switched to another job and comes back to the first job later. This concurrent and *apparently* simultaneous running of jobs is known as **multiprogramming**.

It all happens so quickly that even if there are many people using the system each one thinks that he has sole control. During multiprogramming, special attention must be paid to the following:

- (1) No program must be allowed to take too much time and monopolise computer resources.
- (2) The memory section occupied by one program must not be violated by another program, so memory spaces must be protected from each other.

## **Editors**

Have you ever had to make major changes to a program, either because you had errors in it or because you wanted to modify it to perform some new tasks? Essentially you wanted to *edit* your program.

Editors are special programs designed to assist in the editing of programs or other pieces of text. There are two broad classes of editors:

- (a) text editors
- (b) screen editors.

A text editor treats everything, programs included, as a sequence of lines of text. It usually provides some or all of the following functions:

- appending more text to a given text
- deleting characters or lines
- printing selected portions of text
- finding parts of text
- inserting text
- altering text
- moving parts of text from one place to another
- substituting one piece of text for another.

A screen editor allows text to be manipulated by movement of a cursor over a screen. Thus many of the above functions are available through simple cursor movement.

## **Utilities**

Programs to perform many useful tasks are often provided under the heading *general software utilities*. Among these tasks may be:

- formatting a blank magnetic disk so that it may be used for reading and writing information
- copying information from one disk to another
- finding the logical errors in programs which are not behaving properly, i.e. debugging programs.

## **Software Packages**

Over the years programs have been written to perform many important commercial, scientific and management tasks. These have been carefully refined and tested and are often available as packages from computer suppliers or software firms. Some examples are:

- Information Storage and Retrieval packages or Data Bases
- Mathematical and Statistical packages
- Stock control systems
- Work planning and control packages (often in the form of critical path analysis programs)
- School Timetabling programs
- Information Sorting and Merging packages
- Special Graphics and Design programs.

### **Summary**

Software ranges from programs which are absolutely vital to the operation of a computer system, such as parts of an operating system, to those which enable certain regular jobs to be done more conveniently and expertly, such as software packages.

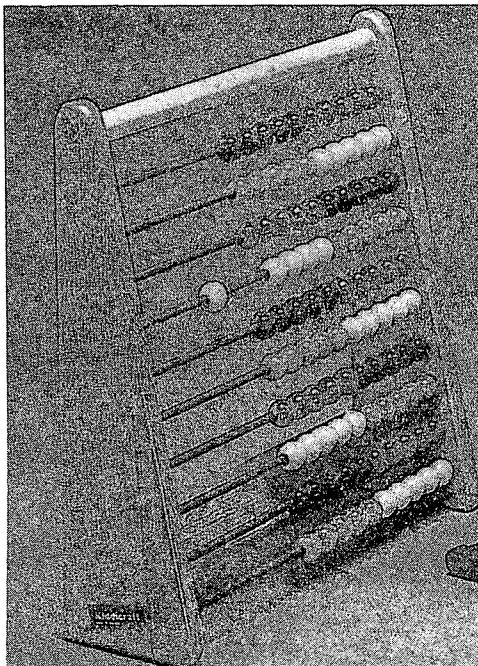
## **QUESTIONS and EXERCISES**

1. Describe briefly all the pieces of software discussed in this chapter.
2. Why is software needed? What would happen if a computer had no software at all? What is the very minimum software a computer needs, in your opinion?
3. Which type of editor would you prefer, a text editor or a screen editor? Why?
4. You are asked to design a geography of Ireland information package. Draw up a list of important items for the design.

# 9 A Brief History of Computing

## Origins

Although the automatic digital computer is a 20th century phenomenon its origins lie far back in history. The abacus, which dates back to very ancient times, was perhaps the first device used by man to assist him in calculating.



Abacus

Here in Ireland the abacus serves nowadays mainly as a children's toy, or perhaps an educational aid in infant classes, but in the Far East, e.g. in Japan, it is still used in serious business calculations, etc. The Japanese abacus is known as a *soroban*. The fact that the abacus can be used with great skill and accuracy was clearly demonstrated in 1946, when, in a calculating contest, a Japanese soroban expert soundly defeated a skilled U.S. army man operating an electric desk calculator.

In the 17th century, advances were made in the construction of mechanical aids to calculation by three mathematicians on the continent of Europe, **Wilhelm Schichard** (c. 1623), **Blaise Pascal** (c. 1642) and **Gottfried Von Leibniz** (c. 1694). Schichard's device was a wooden mechanical calculator, but the metal devices of Pascal and Leibniz achieved greater fame. Pascal's calculator could perform addition

and subtraction and take account of a ‘carry’ from the units place to the tens place, etc. The device constructed by Von Leibniz could perform all four arithmetic operations: addition, subtraction, multiplication and division. These calculators served as prototypes for the desk calculators which were later to become so widespread.

Apart from the invention of mechanical calculators it is also considered that the invention and development of logarithm tables by **John Napier** (1550–1617) and **Henry Briggs** (1561–1631) were greatly influential in the advancement of automatic computation.

## Charles Babbage

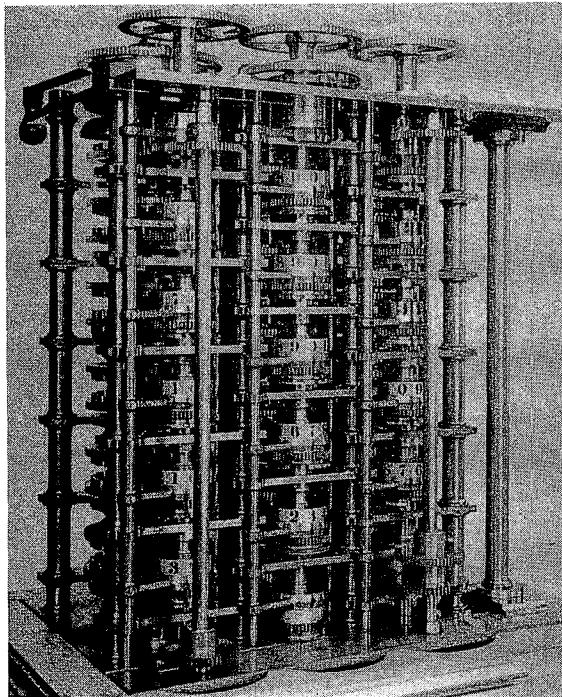
One of the greatest pioneers of automatic computers was the English mathematician and inventor Charles Babbage (1792–1871). When he was about twenty years old he began to work on the idea of mechanising the calculation of mathematical tables.



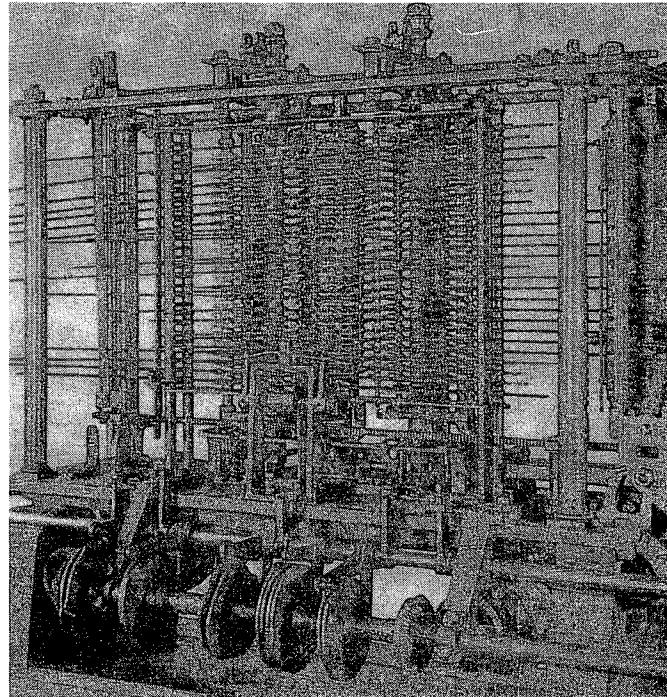
He formulated two good designs for computing machines, known as

- (1) The Difference Engine
- (2) The Analytical Engine

These were to be constructed entirely from mechanical devices such as wheels, gears, cams, etc. However, Babbage’s ideas were far ahead of his time and his engines were never constructed, partly because the parts could not be made with sufficient accuracy with the available 19th century technology. Babbage himself succeeded in making just a small-scale model of the Difference Engine. However, his ideas included many of the ideas incorporated in modern computers. One of his closest collaborators, **Countess Ada Lovelace**, is considered to be the world’s first programmer and one of the most recently developed high-level programming languages is called ADA in her honour.



The Difference Engine



Analytical Engine

Although for various reasons Babbage did not succeed in building working versions of his automatic calculators, his ideas were taken up by George Scheutz and his son Eduard, of Stockholm, and with Babbage's encouragement they built a difference engine which worked.

Curiously enough, an Irishman called **Percy Ludgate** published a design for an analytical engine in the Scientific Proceedings of the Royal Dublin Society in 1909. Ludgate does not appear to have been aware of Babbage's work and his own design is completely independent.

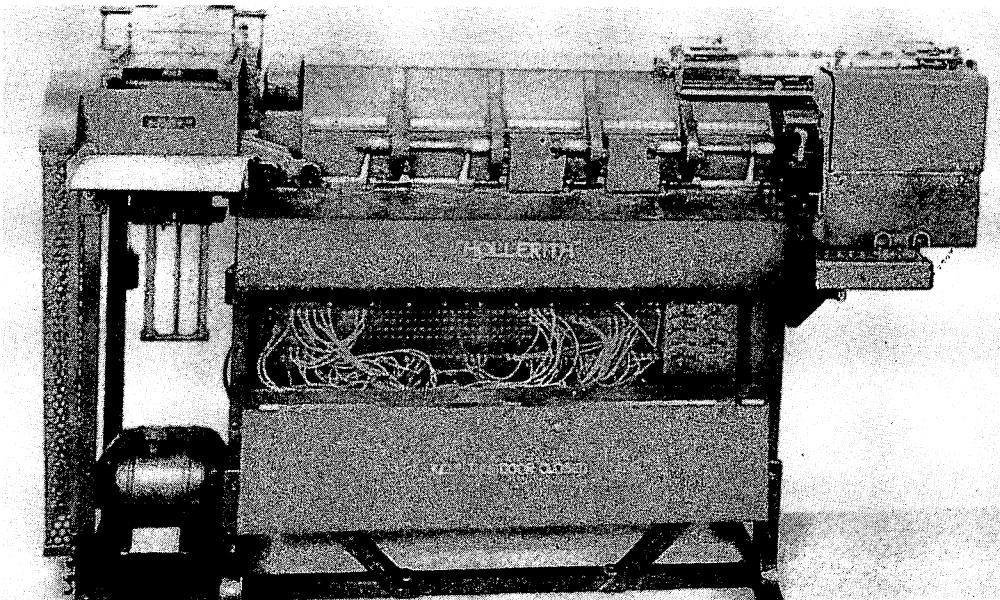
Another Irish link with the history of computing was forged by **George Boole**, an English logician, who was Professor of Mathematics at Cork in the first half of the 19th century. His work on logical algebra, using the binary symbols 0 and 1, now forms the basis on which modern computing is based.

## The role of the Punched Card

Punched cards were first used in the weaving of cloth. An early reference to this dates back to 1728 when **Falcon** constructed an apparatus which made use of holes punched in rectangular cards to weave complex designs. The idea was further

developed by **Joseph Marie Jacquard**, who used strings of cards to construct complete patterns. Repeating patterns could be formed by joining the cards in a loop.

During the 1880's the realisation that the information gathered in the U.S. census of 1880 could not be fully processed before the 1890 census, stimulated **Herman Hollerith** to search for a means of speeding up the calculations. He realised that punched cards could be used to hold information and that the information could then be readily sorted into different categories. He developed machines for punching, reading, sorting and tabulating cards and later formed the Tabulating Machine Company which was the forerunner of IBM, the largest computer manufacturer in the world to-day.



Hollerith Tabulator and Listing Machine

## The New Age

A resurgence of interest in the design of general-purpose digital computers occurred in the 1930's. In 1936 the English mathematician **A. M. Turing** published a brilliant mathematical paper which discussed the problem of automatic computation and contained the fundamental idea of a stored program. In 1938 in Germany **Konrad Zuse** built a mechanical computer called the Z1 which used binary arithmetic instead of the decimal arithmetic favoured by Babbage. Zuse subsequently built two further machines the Z2 and Z3, but his work was interrupted by the Second World War and was of little subsequent influence.

In the United States, however, the early developments quickly built up into an

irresistible flood. One of the pioneers in the field was **Howard Aiken** of Harvard University. During the years 1939 to 1944 he and some engineers from the International Business Machines Company (IBM) constructed a fully-automatic computer which could perform table look-up operations as well as addition, subtraction, multiplication and division. It was a decimal arithmetic electromechanical device and received its information through the medium of punched cards. It was known as the ASCC (Automatic Sequence Controlled Calculator) or, sometimes, the Harvard Mark I.

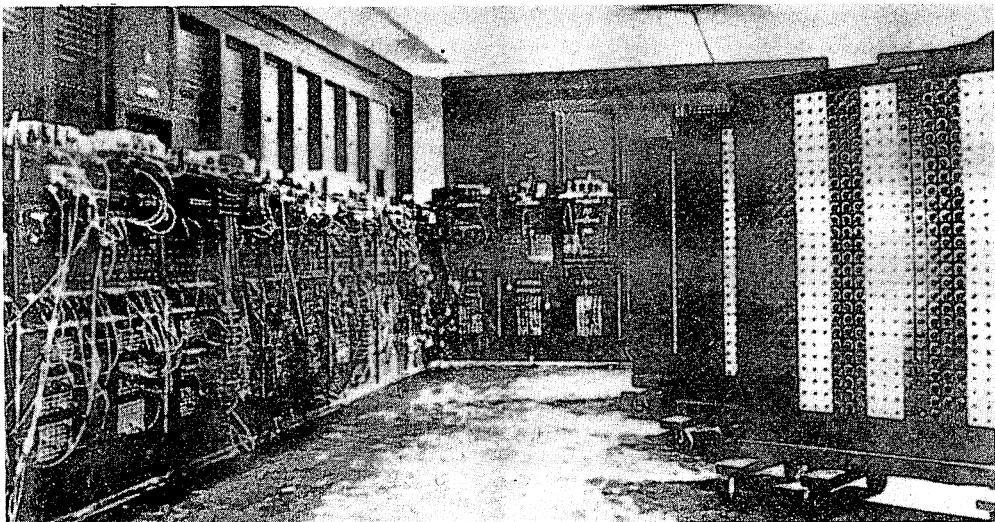
The success of the Harvard Mark I was overshadowed by the development of the first *electronic* general-purpose computer by two scientists from the University of Pennsylvania - **J. P. Eckert** and **J. W. Mauchly**. This was an electronic rather than an electromechanical machine in that it used electronic vacuum tubes or valves and it heralded a new and dynamic era in the development of automatic computers. It was completed in 1946 and was called the ENIAC (Electronic Numerical Integrator and Calculator).

It was the use of electronics in building computers which stimulated the extraordinary developments which have continued to the present day. Mechanical computers had two main drawbacks:

- (a) the calculating speed was limited by the inertia of their moving parts
- (b) the movement of information by gears, levers, etc. was cumbersome and unreliable.

Electronic devices were both faster and more reliable.

In fact, the first electronic computer is credited to **John Atanasoff** of Iowa State University, but his was a special-purpose computer used to solve algebraic equations only, rather than a general-purpose machine like the ENIAC.



ENIAC

The ENIAC could be used in the solution of many different kinds of problem. Some of its notable features were:

- (a) It was 1000 times faster than the Harvard Mark I and took 3/1000 of a second for a ten-digit multiplication.
- (b) It weighed 30 tons and contained 18000 valves.
- (c) It used the decimal representation of numbers rather than the binary form which is universally used nowadays.
- (d) Programs and data were held in separate memories, whereas now they are always held in the same memory.
- (e) It had to be programmed manually by setting switches and plugging and unplugging cables.

## Three Generations of Computers

The Second World War was a major factor in pushing ahead the development of fast automatic computers. In Britain the main effort went into the development of machines for breaking secret codes. The most famous of these machines was known as the *Colossus*.

A feature of the early machines was the separation of programs and data. The great Hungarian-born mathematician **John Von Neumann** was influential in having the stored program concept (the idea of storing programs and data in the same memory) universally adopted when he and Mauchly and Eckert (of ENIAC fame) published a design for a new computer called EDVAC (Electronic Discrete Variable Computer) in 1945. We have seen that this idea was contained in A. M. Turing's famous theoretical paper of 1936 and in fact Turing proposed a design himself for a computer called ACE (Automatic Computing Engine) in 1945. In any event the stored program concept became a standard feature on computers from the mid 1940's onward.

## The First Generation

The late 1940's and early 1950's are known as the time of *the first generation of modern computers*. The computers of the time were characterised by the following features:

- (a) They were very large, consumed a great deal of energy and gave off much heat.
- (b) They made use of valve technology.
- (c) They were difficult to program because instructions had to be written in low-level code.

Some notable machines of this first generation were:

EDVAC

WHIRLWIND 1

ATLAS

EDSAC (Electronic Delay Storage Automatic Computer)

- BINAC (one of the first machines to use self-checking devices)  
UNIVAC 1 (perhaps the first machine to process numeric and alphabetic information with equal ease).  
LEO 1 (Lyons Electronic Office which used magnetic tape as a storage medium).

Among the great pioneers of automatic computing at the time were:

A. M. Turing	Howard Aiken
John Von Neumann	H. H. Goldstine
J. P. Eckert	A. W. Burks
J. W. Mauchly	M. V. Wilkes

## The Second Generation

The 1950's saw a very rapid growth in the development and use of digital computers. The *second generation of computers* is usually thought to date from 1955. In that year the magnetic core store was invented and quickly became the standard form of main memory.

The second generation was characterised by the following features:

- (a) a shift from vacuum tube (valve) to transistor technology, bringing about greater reliability and smaller size
- (b) the replacement of cathode ray tube and delay-line memories by ferrite core and magnetic drums
- (c) the widespread use of index registers and floating point arithmetic
- (d) the introduction of high-level languages - FORTRAN, COBOL, ALGOL, etc.
- (e) the introduction of special processors to help the main central processing unit to manage the input and output of information
- (f) the development of sophisticated system software such as compilers, subroutine libraries, etc.

Some of the important machines of this era were:

The IBM 704, 709, 7090, 7094, 1620 and 1401 machines

RCA 501

Remington Rand 1107

Burrough B5000

English Electric KDF-9

A few supercomputers with huge computing power were developed, notably LARC (Livermore Atomic Research Computer) and STRETCH which was designed by IBM. These machines used techniques for carrying out many processes in parallel with one another. Although they were commercial failures they had a considerable influence on the next generation of computers.

## The Third Generation

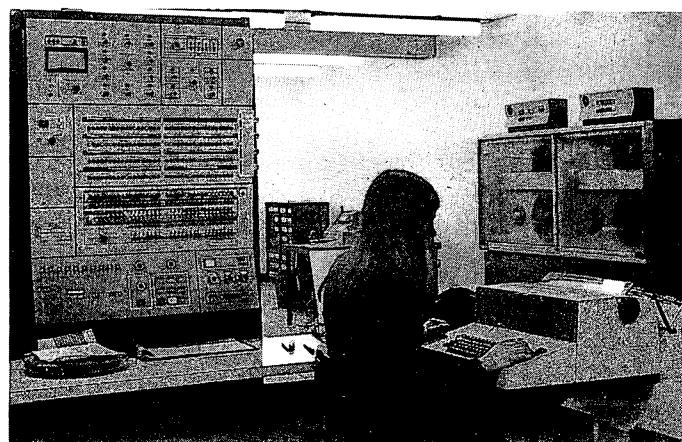
1965 is generally taken as the year which separates the second and third generations of modern computers. However, as is usual with historical developments there is no clear-cut distinction between the two and one generation actually merged into the next.

The third generation was characterised by the following features:

- (a) the widespread introduction of integrated circuits, resulting in substantial reductions in size and cost
- (b) the development of sophisticated operating systems
- (c) the use of integrated circuit memories to augment and supplement ferrite core memories
- (d) a technique called *microprogramming* came into widespread use, to simplify the design of logical circuits and increase their flexibility
- (e) in order to further increase the already incredible speed of large computers, many parallel processing techniques were introduced.

Perhaps the most influential computers of the 3rd generation were those of the IBM 360 series. Another important development was the mass production of low cost minicomputers such as the DEC PDP-5 and PDP-8.

Further large very powerful computers, in the tradition of LARC and STRETCH were designed. Notable among these were the Control Data Corporation series of machines, the Texas Instruments ASC (Advanced Scientific Computers) and the ILLIAC IV (Illinois Automatic Computer).



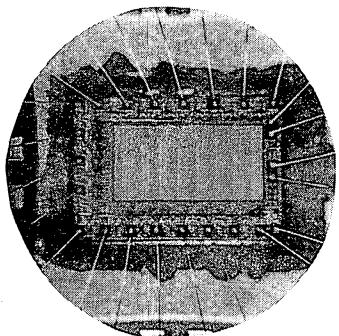
IBM 360 Console

## The Fourth Generation

Computer experts are not too clear on what constitutes the fourth generation of modern computers, but if any single development may be said to characterise it, it must surely be the universal employment of **microprocessors**.

Perhaps the most important piece of technology in the world today is the **micro chip**. The main components in the computer in your school are micro chips. These extraordinary flexible devices are now being included in cars, washing machines, cookers, typewriters, etc., and are potentially capable of totally transforming the world in which we live and work.

A micro chip is a piece of semiconductor material, usually silicon, which contains many thousands of electronic logic circuits.



The first such device was developed by the Intel Corporation of America about 1970. The first one to be commercially available was the Intel 4004 which appeared in 1971. Since then many, many companies have gone into the development and production of micro chip devices and the world market is growing at a very fast rate. Ireland, too, has joined in this growth and during the last few years some micro technology firms have set up factories here.

The process of putting thousands of logic circuits on a tiny chip, no bigger than a finger nail, is called *Large Scale Integration* (LSI). On some chips there may be up to 100,000 such circuits. A microprocessor is a complete central processing unit of a computer, fabricated on a single chip.

## Language Development

The history of modern computing is not concerned solely with the development of smaller, faster, more powerful and more sophisticated hardware. Great advances were also made in methods of communicating with this machinery. In the 1940's programmers were obliged to communicate with computers by means of tedious

machine code, i.e. sequences of 0's and 1's. To do this they had to have an intimate knowledge of machine architecture. Gradually, however, mnemonic instructions (e.g. LOAD, ADD, etc.) were developed to replace the purely numeric code. Of course since computers really 'understand' only numeric code, translators had to be written to translate the mnemonic code into machine code. These were the assemblers and compilers we have spoken of.

Nowadays almost all programming is done in high-level languages such as FORTRAN, COBOL, BASIC, Pascal, COMAL, etc. These languages are easier for humans to use and understand. Structured languages like Pascal and COMAL are nowadays the most favoured. COMAL was designed in 1973 by two Danes, Borge Christensen and Benedict Loefstedt and is currently attracting a great deal of attention in educational circles.

### **Summary**

Following thousands of years of slow development in automatic computation, computers 'took off' with the employment of electronics in the 1940's.

Four generations of computers may be traced since the mid 1940's, each with its own distinguishing characteristics.

### **QUESTIONS and EXERCISES**

1. Name (a) ten famous people, (b) ten famous machines, in the development of computers.
2. Write essays on the lives of three famous computer pioneers.
3. Why was a punched card useful in weaving? Why is it useful for representing information? What advantages did Hollenith discern in the representation of information on punched cards?
4. Find out all you can about Charles Babbage and his computing designs. What connection had Lady Lovelace with Charles Babbage?
5. Find out what you can about the high-level languages FORTRAN, ALGOL, COBOL and BASIC.
6. Why has the micro chip become so universally popular?

# 10 Lists, Arrays, Strings

## Teachers' Tiff

Mary and Joan are the best of friends but they have an occasional disagreement. 'Do you know, Joan,' says Mary one day as they were both preparing the Summer reports, 'the Green Class is a terrific class this year. Their average mark was 65 in Computer Studies. I suppose it must have been all the extra work after school.'

'Isn't that a coincidence now, Mary,' says Joan, 'I had Green last year and they also scored an average of 65 in Computer Studies.'

'Ah,' says Mary, 'but I had no stragglers. They were all very close together and there was only a small difference between all their marks.'

'Mine were close too ...,' says Joan.

'They weren't as close as mine,' says Mary.

'Yes they were.'

'No they weren't.'

'Yes they were' ...

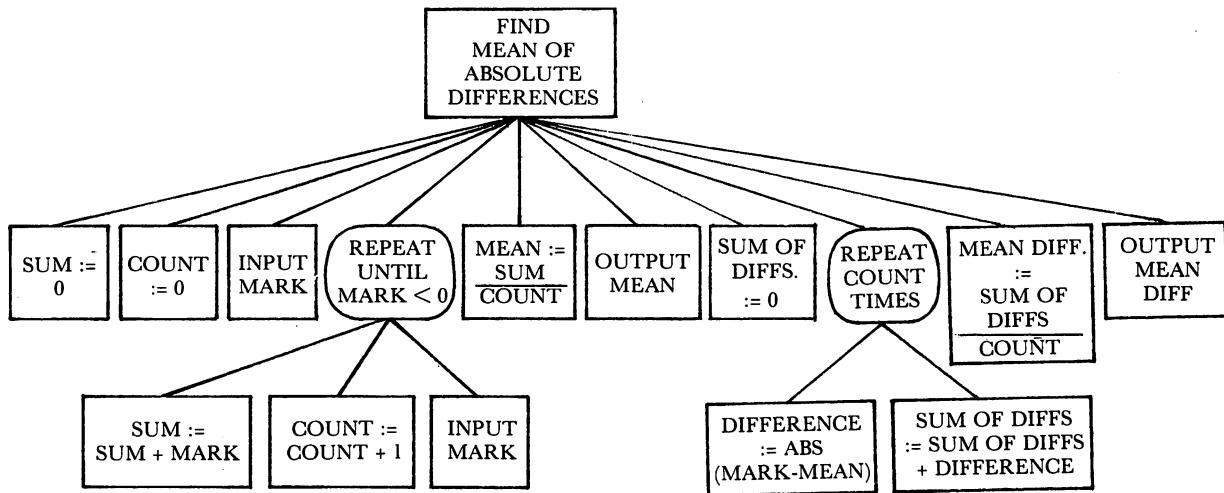
Being a peace-loving fellow I decided to step in and offer my Solomon-like gifts in the service of resolving the dispute. I suggested that a program be written to determine the degree of 'closeness' of each class. 'Write it yourself,' they both snapped at once, and Solomon, his dignity in shreds, retired to wrestle with the problem.



## Meaning the differences

The first method of attack which suggested itself was to find the difference between each mark and the mean and then to find the average of these differences. However, after trying a few experiments I discovered that the sum of the differences was always zero. (Try this yourself for some sets of numbers, or better still prove that it always happens.) This was because the negative differences for those marks below the mean cancelled the positive differences for the marks above the mean. Then (with the help of a book on Statistics!) I had the bright idea of getting the absolute value of the differences, adding these up and getting their mean. This would work, because absolute values are, of course, always positive.

So our problem now is to find the average of the absolute differences between marks and their mean.



Our solution involves

- finding the mean of the actual marks
- subtracting the mean from each individual mark and getting the absolute difference. (In our program we will use a COMAL function **ABS** to do this.)
- adding the absolute differences
- finding the mean of the absolute differences.

The left hand portion of the above diagram is the same as the diagram on page 60. This suggests that we can start writing our program by copying the appropriate statements from Program 34, in particular lines 70 to 150, thus:

```

0070 SUM:=0
0080 COUNT:=0
0090 INPUT "FIRST MARK ? "; MARK
0100 REPEAT
0110 SUM:=SUM+MARK
0120 COUNT:=COUNT+1
0130 INPUT "NEXT MARK ? "; MARK
0140 UNTIL MARK<0
0150 MEAN:=SUM/COUNT
  
```

Now we want to carry on and find the absolute value of the difference between each mark and the mean. But an awkward problem rears its ugly head! We no longer have each mark!

Look back at line 130. This statement is executed over and over again and as each mark is read in it replaces the previous value of *MARK*. So all we have when the loop finally terminates is the negative rogue value used to signal the end of the data. A fat lot of good that is!

So what can we do?

Well, we could type in all the values again by including another INPUT statement in a new REPEAT loop. But it is pretty tedious having to enter the numbers once without having to go through the whole process again. Is there another way?

We could use a different name for each mark, e.g. MARK 1, MARK 2, MARK 3, etc., so that consecutive values would not over-write each other by being placed in the same variable MARK. Of course we wouldn't use a loop then and our program would look something like this:

```
INPUT "FIRST MARK" : MARK 1  
SUM := SUM + MARK 1  
INPUT "NEXT MARK" : MARK 2  
SUM := SUM + MARK 2  
INPUT "NEXT MARK" : MARK 3  
SUM := SUM + MARK 3
```

and so on for as many marks as we want to process. And then when we had calculated the mean we would have to embark on another long piece of program, e.g.

```
SUM OF DIFFERENCE := ABS (MARK 1 - MEAN)  
SUM OF DIFFERENCES := SUM OF DIFFERENCES  
+ ABS (MARK 2 - MEAN)  
SUM OF DIFFERENCES := SUM OF DIFFERENCES  
+ ABS (MARK 3 - MEAN)
```

:

and so on.

But this solution is, if anything, worse than the previous one. The tedium has been shifted from the entry of the data to the actual program. The two big flaws are:

- (a) the program is long and clumsy
- (b) the loop structure which is basic to the solution is not being used.

What we need is some way of including a loop and of using the same variable name for all the marks, yet distinguishing between the different individual marks to prevent over-writing. A tall order! Yet it can be done quite simply by making use of the idea of a *subscripted or indexed variable*.

## The DIM statement

What is meant by a 'subscripted variable'? Consider a list of items and their prices given as follows:

1st item	£2.52
2nd item	£1.78
3rd item	£2.16
4th item	£1.99
5th item	£2.35
6th item	£1.67
7th item	£2.01
8th item	£2.05

Alternatively, the list could be written as follows:

item (1) £2.52  
item (2) £1.78  
item (3) £2.16  
.  
item (8) £2.05

or it could be written like this:

item<sub>1</sub> £2.52  
item<sub>2</sub> £1.78  
item<sub>3</sub> £2.16  
. .  
item<sub>8</sub> £2.05

In this last form, the numbers 1, 2, 3 etc. are written as subscripts (*sub* = under). This form is often used in mathematics, e.g. the terms of a sequence of numbers might be written  $t_1, t_2, t_3, \dots, t_n$ . The important thing to notice is that we use the same name (e.g. item) for all the items, but distinguish between the different ones by using different subscripts. The subscripts refer to the different positions of the items in the list. Thus we distinguish between different items, not by name, but by position in a list.

COMAL makes use of this idea of a list distinguished by subscripts. The subscripts are written in brackets as in the second form above,

e.g.      VALUES (7) := 13.8

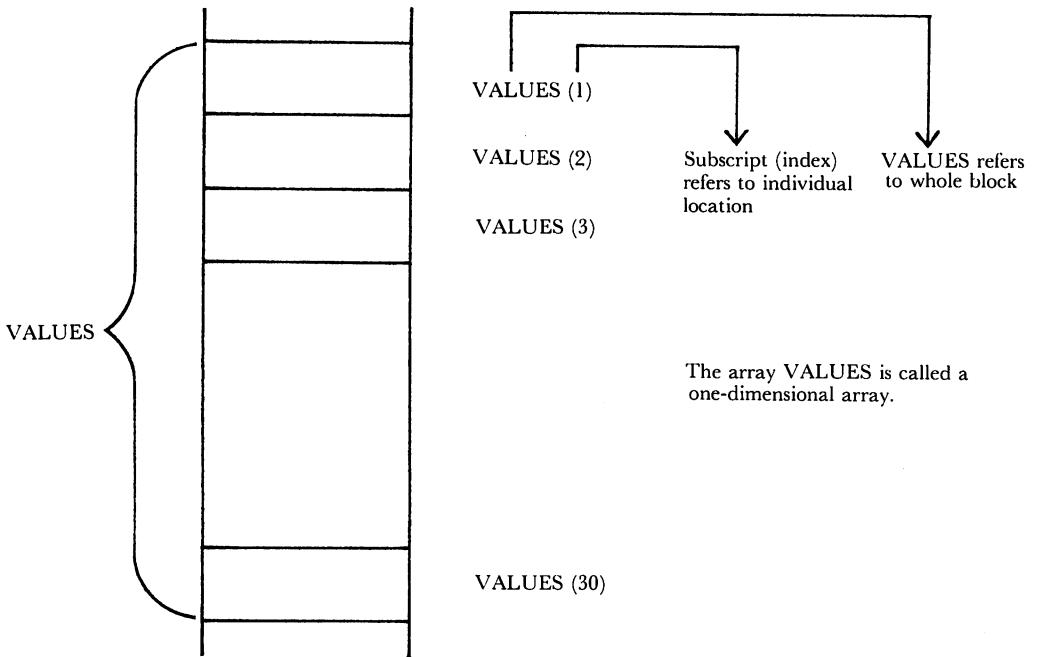
This has the effect of putting 13.8 into the 7th position of a list or array called VALUES.

Similarly LIST #(4) := 63 has the effect of placing 63 in the 4th position of an integer array called LIST#. Note the use of the usual # sign to indicate an integer variable.

As used in these two examples, VALUES and LIST # are often referred to as 'subscripted variables'.

One important point needs to be established before we can make effective use of subscripted variables in programs. Up to this, each time we used a variable name (with the exception of strings) we used it to refer to a single location in the memory of the computer, where this location held a single number, integer or decimal. A subscripted variable name is used to refer to a number of locations, sometimes called a block of locations. The computer must be told how many locations to reserve for that block. This is done by using the **DIM** statement in COMAL, e.g. DIM VALUES(30) causes 30 locations to be reserved in the computer's memory under the name

VALUES. The individual locations are referred to as VALUES (1), VALUES (2), VALUES (3), ... VALUES (30).



## On your Marks

Let us now make use of a one-dimensional array to resolve the marks dispute for our two favourite teachers!

```
0010 // PROGRAM 55
0020 //
0030 // COMELY KATE
0040 //
0050 // TO FIND MEAN ABSOLUTE DIFFERENCE
0060 //
0070 DIM MARKS(50)
0080 PRINT
0090 SUM:=0
0100 COUNT#:=1
0110 INPUT "FIRST MARK "; MARKS(COUNT#)
0120 REPEAT
0130   SUM:=SUM+MARKS(COUNT#)
0140   COUNT#:=COUNT#+1
0150   INPUT "NEXT MARK "; MARKS(COUNT#)
0160 UNTIL MARKS(COUNT#)<0
0170 TOTAL#:=COUNT#-1
0180 MEAN:=SUM/TOTAL#
0190 PRINT
```

```

0200 PRINT "MEAN = ",MEAN
0210 PRINT
0220 SUMOFDIFFERENCES:=0
0230 FOR COUNT#:=1 TO TOTAL# DO
0240 DIFFERENCE:=ABS(MARKS(COUNT#)-MEAN)
0250 SUMOFDIFFERENCES:=SUMOFDIFFERENCES+DIFFERENCE
0260 NEXT COUNT#
0270 MEANDIFFERENCE:=SUMOFDIFFERENCES/TOTAL#
0280 PRINT "THE MEAN ABSOLUTE DIFFERENCE IS ",MEANDIFFERENCE
0290 END

```

RUN

```

FIRST MARK 56
NEXT MARK 64
NEXT MARK 40
NEXT MARK 80
NEXT MARK 49
NEXT MARK 55
NEXT MARK 67
NEXT MARK 71
NEXT MARK 65
NEXT MARK 63
NEXT MARK 53
NEXT MARK 57
NEXT MARK -9

```

MEAN = 60

THE MEAN ABSOLUTE DIFFERENCE IS 8.333333

We have made a few changes to our input section.

First, we set aside an array called MARKS of size 50 to hold the marks. We do not have to use all 50 locations but of course we must not try to use more.

Second, we initialised COUNT # to 1 on line 100 so that it would point to the first position in the array MARKS when it was used in line 110. Thus on line 110, MARKS (COUNT#) means MARKS (1). Note that it is appropriate to use an integer variable for COUNT since it is acting as a subscript or index for the array MARKS. You might ask why not an integer array variable too, say MARKS#, since the marks are all whole numbers. Well, the marks need not be integers, and on another occasion we might like to use decimal values.

Third, on line 140 when we incremented COUNT# it moved one ahead of the actual number of marks entered. This means that at the end of the loop COUNT# is one more than the number of valid marks entered and so has to be reduced by one before finding the mean. This is done on line 170.

We use a FOR loop to find the mean of the differences since at this stage we know exactly how many iterations are required. The actual working of the loop should be quite clear. However just to check that everything is in order let us 'dry run' the program on a very small set of data, e.g. Find the mean and the mean absolute difference of the data 60, 50, 62, 56. Use -3 to terminate data.

## INPUT LOOP

LINE	1st ITERATION			2nd ITERATION			3rd ITERATION			4th ITERATION		
	COUNT#	MARK (COUNT#)	SUM									
130	1	60	60	2	50	110	3	62	172	4	56	228
140	2	?	60	3	?	110	4	?	172	5	?	228
150	2	50	60	3	62	110	4	56	172	5	-3	228

The question mark indicates that we don't know (and don't care) at that stage what that particular location in the array MARK holds.

On line 160 TOTAL # gets a value of 4, i.e. 5 - 1, and on line 170 the MEAN is calculated as 57, i.e. 228/4.

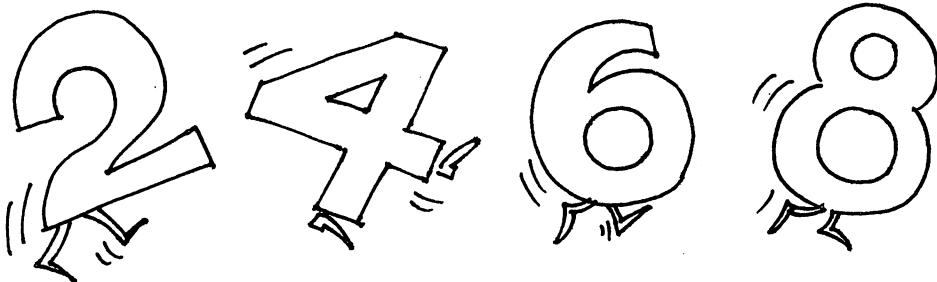
## DIFFERENCES LOOP

LINE	1st ITERATION				2nd ITERATION				3rd ITERATION				4th ITERATION			
	COUNT	MARK (COUNT#)	DIFF	SUM												
240	1	60	3	0	2	50	7	3	3	62	5	10	4	56	1	15
250	1	60	3	3	2	50	7	10	3	62	5	15	4	56	1	16
260	2	50	3	3	3	62	7	10	4	56	5	15	5	-3	1	16

MEAN DIFFERENCE is then calculated as  $16/4 = 4$  on line 270.

*Note* We are not obliged to use the same variable COUNT# for both our loops. Sometimes it is convenient to do so, sometimes not. Usually it is a matter of style. The best thing to do is whatever makes the program clearer and more readable. You may find it better to use a different variable. Do so if you wish.

## Skip along



In order to acquire confidence in handling arrays let us take two more simple examples. For our first example we will read in a list of numbers and print out every second one. Note that we cannot use the identifier LIST as a variable name since it means something else in COMAL. LIST is a reserved word, so we use the name LISST in the following programs.

```

0010 // PROGRAM 56
0020 //
0030 // COMELY KATE
0040 //
0050 // TO PRINT EVERY SECOND ELEMENT OF AN ARRAY
0060 //
0070 TAB:=2
0080 DIM LISST(20)
0090 FOR COUNT#:=1 TO 20 DO
0100 READ LISST(COUNT#)
0110 NEXT COUNT#
0120 //
0130 FOR COUNT#:=1 TO 20 STEP 2 DO
0140 PRINT LISST(COUNT#),
0150 NEXT COUNT#
0160 //
0170 DATA 98.6, 45.7, 67.9, 45.6, 34.9, 56.7, 54.8, 89.7, 99.9, 56.7, 45.9,
47.8, 67.6
0180 DATA 77.7, 68.9, 88.5, 99.6, 87.5, 77.5, 56.9
0190 END

```

RUN

```
98.6 67.9 34.9 54.8 99.9 45.9 67.6 68.9 99.6 77.5
```

## Back, Back I say

For our second example we will read in a list of numbers and print them out in reverse order.

```

0010 // PROGRAM 57
0020 //
0030 // COMELY KATE
0040 //
0050 // TO PRINT A LIST IN REVERSE ORDER
0060 //
0070 TAB:=2
0080 DIM LISST(20)
0090 FOR COUNT#:=1 TO 20 DO
0100 READ LISST(COUNT#)
0110 NEXT COUNT#
0120 //
0130 FOR COUNT#:=20 DOWNTO 1 DO
0140 PRINT LISST(COUNT#);
0150 NEXT COUNT#
0160 DATA 12, 54, 78, 67, 56, 45, 48, 98, 34, 34
0170 DATA 87, 76, 65, 84, 94, 89, 34, 56, 77, 88
0180 END

```

RUN

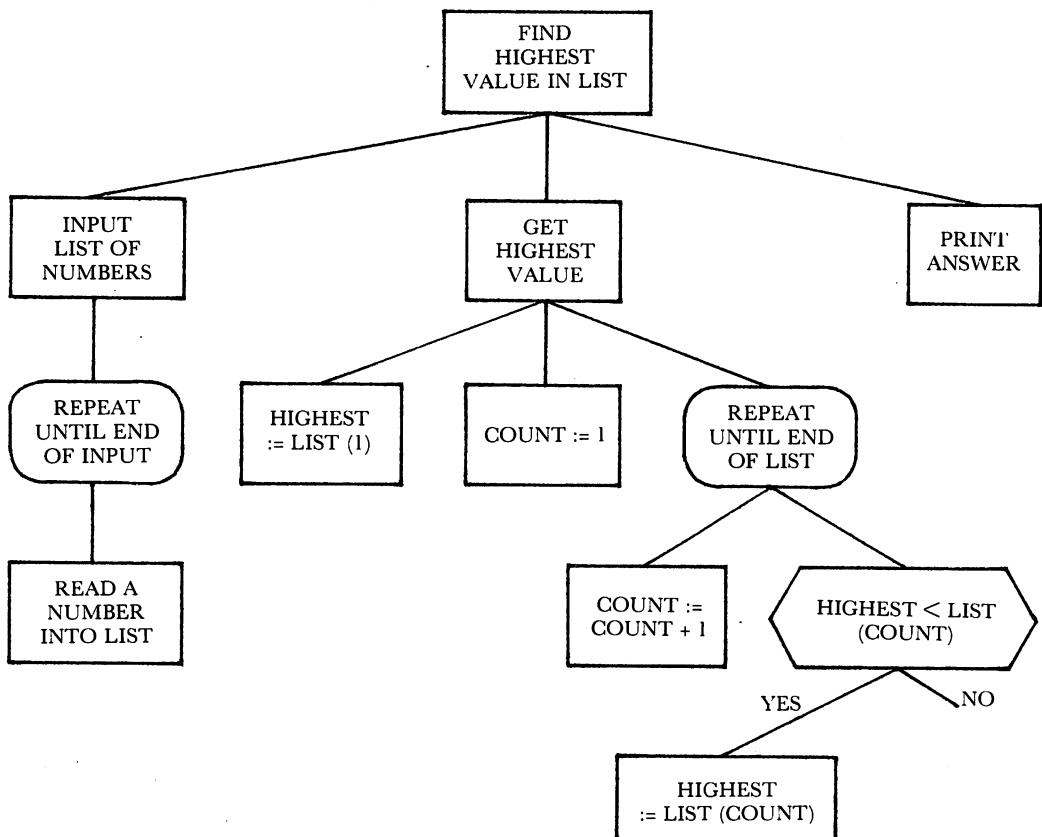
```
88 77 56 34 89 94 84 65 76 87 34 34 98 48 45 56 67 78 54 12
```

These are two very simple programs and you should have no trouble following them. We have used READ and DATA so that you can see the results quickly.

## 'Higher, still and Higher'

**Problem:** Read a list of numbers and find the highest value.

In determining which value is the largest we begin by assuming that the first value in the list is the largest and so we set HIGHEST = LIST (1).



We then search down through the list looking for a higher value. If we find one, the answer to the question posed in the selection box HIGHEST < LIST (COUNT) becomes YES and we reset HIGHEST to the value of this entry in the list, thus HIGHEST := LIST (COUNT).

Note that we use the word LIST and we do not bother to put a # after COUNT in the diagram because we are not concerned with the details of the COMAL program at this stage. Remember the structure diagram is to help us outline the solution to our problem and to break down our problem into manageable chunks. On the first level of the diagram above we simply spell out the three major tasks to be done. Then on the second and subsequent levels we break two of these tasks down further.

```

0010 // PROGRAM 58
0020 //
0030 // COMELY KATE
0040 //
0050 // TO FIND THE HIGHEST NUMBER IN A LIST
0060 //
0070 DIM LISST(20)
0080 //
0090 // FIRST SET UP THE LIST
0100 FOR COUNT#:=1 TO 20 DO
0110   READ LISST(COUNT#)
0120 NEXT COUNT#
0130 //
0140 // NOW SEARCH FOR HIGHEST VALUE
0150 //
0160 HIGHEST:=LISST(1)
0170 FOR COUNT#:=2 TO 20 DO
0180   IF HIGHEST<LISST(COUNT#) THEN HIGHEST:=LISST(COUNT#)
0190 NEXT COUNT#
0200 PRINT
0210 PRINT "THE HIGHEST NUMBER IN THE LIST IS ",HIGHEST
0220 DATA 12, 54, 78, 67, 56, 45, 48, 98, 34, 34
0230 DATA 87, 76, 65, 84, 94, 89, 34, 56, 77, 88
0240 END

```

RUN

THE HIGHEST NUMBER IN THE LIST IS 98

## Strings

No doubt you have now worked out why it was necessary to use DIM with every string we have used in our programs so far, e.g. DIM PLURAL\$ of 20. The DIM used with strings is the same DIM as we have been using in this chapter. A string is considered to be an array of characters with one character per location in the array.

Does this mean that we can pick out an individual character by specifying its position within the string? It certainly does.

Suppose, for example we have declared the variable NAME\$ as follows:

DIM NAME\$ OF 20

and have given NAME\$ the value JOE SOAP by the instruction NAME\$ := "JOE SOAP"

Then      NAME\$ (1) = J  
           NAME\$ (2) = O  
           NAME\$ (7) = A, etc.

We remind ourselves that although we have allowed for up to 20 characters in our string NAME\$ we do not actually have to *have* that many characters. In the previous example there are 8 characters in NAME\$ (don't forget the blank or space in the middle). The number of characters in a string is referred to as the length of the string.

COMAL provides us with a function called **LEN** which tells us how many characters there are in a string,

e.g.            **LEN (NAME\$) = 8**  
                **LEN ("HEY DIDDLE DIDDLE") = 17**

What would happen if we referred to NAME\$ (9) when the length of NAME\$ is only 8? The system would tell us we are making an error. If it occurred in a program the program would stop.

Let us now tackle a few simple examples of string manipulation.

## A Okay

Count the number of times the letter A appears in a string.

```
0010 // PROGRAM 59
0020 //
0030 // COMELY KATE
0040 //
0050 // TO COUNT THE NUMBER OF A'S IN A STRING
0060 //
0070 DIM STRING$ OF 50
0080 PRINT
0090 INPUT "ENTER ANY STRING  ": STRING$
0100 PRINT
0110 LENGTH:=LEN(STRING$)
0120 POSITION:=0
0130 NUMBEROFA$:=0
0140 WHILE POSITION<LENGTH DO
0150   POSITION:=POSITION+1
0160   IF STRING$(POSITION)="A" THEN NUMBEROFA$:=NUMBEROFA$+1
0170 ENDWHILE
0180 PRINT "NUMBER OF A'S = ",NUMBEROFA$
0190 END
```

RUN

ENTER ANY STRING     THE FAT CAT SAT ON THE MAT

NUMBER OF A'S =    4

The length of the string is calculated in line 110 and the number of A's is initialised to zero on line 130. Then we move through the string one character at a time and check to see if it is an A. If it is, we add 1 to the number of A's (see line 160). Note that because the IF statement occurs all on one line we do not need an ENDIF statement.

POSITION has been initialised to 0 on line 120 so that the first time it is incremented inside the WHILE loop (line 150) it becomes 1 and points to the first character in the string.

## Back to Front

Read in an arbitrary string and print it backwards.

```

0010 // PROGRAM 60
0020 //
0030 // COMELY KATE
0040 //
0050 // TO REVERSE A STRING
0060 //
0070 DIM STRING$ OF 50
0080 PRINT
0090 INPUT "ENTER ANY STRING  ": STRING$
0100 PRINT
0110 LENGTH:=LEN(STRING$)
0120 FOR POSITION:=LENGTH DOWNTO 1 DO
0130 PRINT STRING$(POSITION),
0140 NEXT POSITION
0150 END

```

RUN

ENTER ANY STRING THE CAT SAT ON THE MAT

TAM EHT NO TAS TAC EHT

The length of the string is calculated on line 110. This gives the position of the last character in the string. Therefore working from that position down to the first position and printing each character as we go, we have the string backwards. This is exactly what our FOR loop (lines 120 to 140) does.

## Back to Back

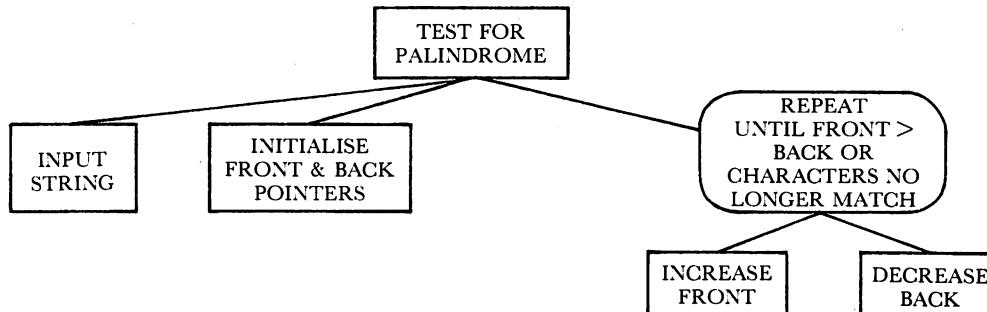
Read in an arbitrary string and determine if it is a palindrome.

A palindrome (as, of course, you know!) is a string which reads the same backwards as forwards.

Two counters are needed, FRONT to count forward through the string, and BACK to count backwards. FRONT is initialised to 0, BACK is initialised to the length of the string +1 (Why is this?).

FRONT is incremented by 1 and BACK is decremented by 1, until one or other of the following happens:

- (1) FRONT and BACK “cross over” each other, in which case the letters have been matching front and back and the string is a palindrome.
- (2) The selected letters don’t match, in which case we do not have a palindrome.



```

0010 // PROGRAM 61
0020 //
0030 // COMELY KATE
0040 //
0050 // TO FIND IF A STRING IS A PALINDROME
0060 //
0070 DIM STRING$ OF 50
0080 PRINT
0090 INPUT "ENTER ANY STRING  ": STRING$
0100 PRINT
0110 LENGTH#:=LEN(STRING$)
0120 FRONT#:=0
0130 BACK#:=LENGTH#+1
0140 MATCHING:=TRUE
0150 REPEAT
0160   FRONT#:=FRONT#+1
0170   BACK#:=BACK#-1
0180   IF STRING$(FRONT#)<>STRING$(BACK#) THEN MATCHING:=FALSE
0190 UNTIL FRONT#>BACK# OR NOT MATCHING
0200 PRINT
0210 IF MATCHING THEN
0220   PRINT STRING$,"  IS A PALINDROME"
0230 ELSE
0240   PRINT STRING$,"  IS NOT A PALINDROME"
0250 ENDIF
0260 END

```

RUN

ENTER ANY STRING THE FAT CAT

THE FAT CAT IS NOT A PALINDROME

RUN

ENTER ANY STRING ABLE WAS I ERE I SAW ELBA

ABLE WAS I ERE I SAW ELBA IS A PALINDROME

Because the length of the string, the front pointer and the back pointer are all whole numbers, we have used integer variables LENGTH#, FRONT# and BACK#.

However, the really important point to pay special attention to is the use of a new type of variable -

*a logical variable*

The logical variable in this program is MATCHING. A logical variable is one which has only two values of interest, TRUE and FALSE.

In the program MATCHING is given the value TRUE to start with (line 140) and it remains TRUE so long as the character in the FRONT position /in the string is equal to (i.e. matches) the character in the BACK position. If these characters do not match at any stage, MATCHING takes the value FALSE (see line 180). If this happens, the REPEAT loop terminates, since it will only continue until one or other of the conditions FRONT > BACK or NOT MATCHING becomes true.

*Note* Logical variables are often called Boolean variables in honour of George Boole whose work on logic we mentioned in the last chapter.

## Logical Operators

There are three logical operators - AND, OR, NOT.

Just as arithmetic operators are applied to numbers or expressions,

e.g.       $42.7 - 18.9$

$(4 - A) + (81 * B * C)$

so logical operators are applied to logical variables or conditions,

e.g.      LOST OR FOUND

$(A < B) \text{ AND } (B < C)$

In COMAL logical variables usually represent conditions which can be true or false. In order to use the logical operators AND, OR and NOT you must know the following:

For CONDITION 1 AND CONDITION 2 to be true, *both CONDITION 1 and CONDITION 2 must be true separately*, otherwise the combination is false.

For CONDITION 1 OR CONDITION 2 to be false, *both CONDITION 1 and CONDITION 2 must be false separately*, otherwise the combination is true.

NOT true is false.

NOT false is true.

## The logical variable EOD

When a program is using READ statements to take data from a DATA list, it is important for the program to 'know' when the end of the data is reached. In Programs 56, 57 and 58 there were exactly 20 items of data and simple FOR loops were used to read exactly the right amount. This of course obliged us to make sure we had at least 20 items in our DATA lines. (What would happen if we had more?) We could have used the idea of a rogue value as we did with INPUT in Program 55. For example we might have used a negative number as an indication that the end of data had been reached.

However, COMAL provides us with a very useful system variable - **EOD** (End of Data) - which allows us to test very easily for the end of data. EOD has the value FALSE as long as there are DATA items remaining to be read. When the last item has been read EOD is set to TRUE. Thus a data reading loop can be controlled by

WHILE NOT EOD DO

⋮

REPEAT

⋮

UNTIL EOD

or

Remember that EOD is automatically set in a program, depending on whether there are DATA items remaining to be read or not. You do not set EOD yourself.

You should rewrite Programs 56, 57 and 58 using loops controlled by EOD instead of FOR loops.

## Types of Variables

We have now met five types of variables in COMAL:

- (1) Ordinary numeric variables capable of holding whole numbers and decimal numbers, e.g. NUMBER := 4.78.
- (2) Integer numeric variables capable of holding whole numbers only, e.g. COUNT# := 17.
- (3) Number arrays, e.g. LISST (5) := 12.9 or VALUES (5) := 12.
- (4) Strings, e.g. NAME\$ := "DILLY DREAMER".
- (5) Logical variables, capable of holding the value TRUE or the value FALSE, e.g. GOOD := TRUE.

You may wonder why there is no way of distinguishing a logical variable from a numeric variable. Well, COMAL actually treats a logical variable like a numeric variable by taking the value 1 as equivalent to TRUE and the value 0 as being equivalent to FALSE. In fact, therefore, we could use the # sign in our logical variable names since these two values, 1 and 0, are integers, e.g. GOOD# := TRUE.

An even more interesting point is that *most COMAL systems take 0 to be FALSE and any other value to be TRUE.*

## Substrings

So far we have manipulated our strings one character at a time. Often, however, we need to refer to a substring of a string, i.e. a sequence of characters forming part of a string.

- e.g.      "SO" is a substring of "JOE SOAP"  
              "E" is a substring of "JOE SOAP"  
              "E SO" is a substring of "JOE SOAP"

How can we specify a part of a string? We simply use the starting position and finishing position of our substring as follows:

Suppose NAME\$ = "JOE SOAP"

then      NAME\$ (5,6) = "SO"  
              ↑↑  
              starting position ————— finishing position

NAME\$ (3,3) = "E"  
NAME\$ (3,6) = "E SO"

Let us write a very simple program to illustrate the use of substrings.

The aim of the program is to read in a person's name in the form christian name, surname and to print it in the order surname, christian name. For simplicity we assume that there are just 5 letters in each part of the name.

```
0010 // PROGRAM 62
0020 //
0030 // COMELY KATE
0040 //
0050 // TO REVERSE A NAME
0060 //
0070 DIM NAME$ OF 11, LEFTPART$ OF 5, RIGHTPART$ OF 5
0080 PRINT
0090 INPUT "ENTER NAME "; NAME$
0100 LEFTPART$:=NAME$(1,5)
0110 RIGHTPART$:=NAME$(7,11)
0120 PRINT
0130 PRINT RIGHTPART$+" "+LEFTPART$
0140 END
```

RUN

ENTER NAME BILLY BONES

BONES BILLY

RUN

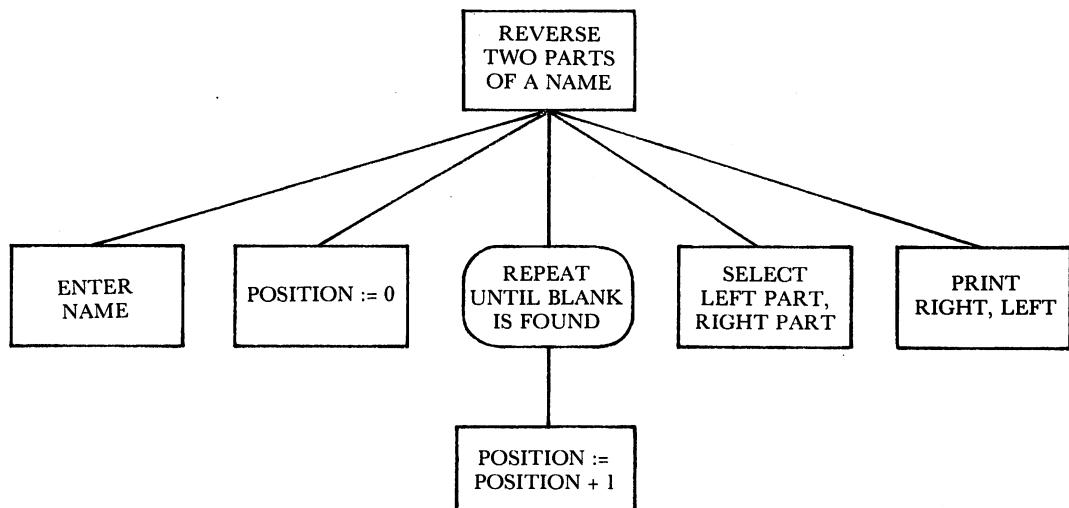
ENTER NAME BOBBY BRIGHT

BRIGH BOBBY

It is easy to see how the program works. The christian name is picked out and assigned to the variable LEFTPART\$ in line 100. The surname is picked out and assigned to RIGHTPART\$ in line 110. Then the RIGHTPART\$ is placed before the LEFTPART\$ and printed in line 130.

As you can see this program works properly only if there are exactly 5 letters in each part of the name. We could easily modify it to work on different lengths of christian name and surname by changing the number 5 on each of the lines 70 and 100 and the values 7 and 11 on line 110. There is still a difficulty, however. We need to know the number of letters in the christian name and surname beforehand. This is a bit too inflexible. We look for a better way.

Essentially what we have to do is to search for the space which separates the christian name from the surname. (Of course, if there is no space, the approach won't work! But would you be able to separate the two parts correctly if there were no space? For example, does JOERICSON mean JO ERICSON or JOE RICSON?) Once we have found the blank we take the left part up to the blank as the christian name and the right part from the blank to the end as the surname. All we have to do then is print in the order right part, left part.



```

0010 // PROGRAM 63
0020 //
0030 // COMELY KATE
0040 //
0050 // TO REVERSE A NAME
0060 //
0070 DIM NAME$ OF 20, LEFTPART$ OF 20, RIGHTPART$ OF 20
0080 PRINT
0090 INPUT "ENTER NAME ":" NAME$"
0100 POSITION:=0
0110 REPEAT
0120   POSITION:=POSITION+1
0130 UNTIL NAME$(POSITION)=""
0140 LEFTPART$:=NAME$(1,POSITION)
0150 LENGTH:=LEN(NAME$)
0160 RIGHTPART$:=NAME$(POSITION+1,LENGTH)
0170 PRINT
0180 PRINT RIGHTPART$+" "+LEFTPART$
0190 END
  
```

ENTER NAME DILLY DREAMER

DREAMER DILLY

RUN

ENTER NAME BOBBY BRIGHT

BRIGHT BOBBY

The REPEAT loop (lines 110 to 130) simply moves along the string one position at a time until the character found in that position is a space (line 130). Then LEFTPART\$ is assigned the value of the substring from position 1 up to the position of the blank. RIGHTPART\$ is given the substring from the position after the blank up to the end of the string (as given by length). Why is a blank printed between RIGHTPART\$ and LEFTPART\$ on line 180?

## **Summary**

One dimensional arrays are very convenient structures for representing collections of data.

Locations within an array may be identified by a subscript or index.

The size of an array is introduced by a DIM statement.

Substrings, including individual characters, may be selected from strings.

Logical variables help to make programs clearer and more readable.

Logical variables are operated on by the logical operators AND, OR, NOT.

Five types of variables have now been introduced.

### **COMAL KEYWORDS**

AND	AUTO	CASE	CAT	CLEAR	CON	DATA
DEL	DELETE	DIM	DIV	DO	EDIT	ELIF
END	ENDCASE	ENDIF	ENDWHILE	EOD		
FOR	IF...THEN...ELSE		INPUT	LEN	LIST	
LOAD	MOD	NEW	NEXT	NOT	OR	OTHERWISE
PRINT	READ	RENUM	REPEAT	RND		
ROUND	RUN	SAVE	STEP	STOP	TAB	
UNTIL	WHEN	WHILE				

## **QUESTIONS and EXERCISES**

1. What happens if DIM TABLE (40) is declared and only 30 locations are used?
2. If we had used READ and DATA instead of INPUT in our approach to Program 55 we could have used an easy method of restoring our data values after we had calculated the mean. We could have used the RESTORE command which resets the data pointer to the beginning of the data. Rewrite Program 55 using READ, DATA and RESTORE.
3. What would the output be if you issued the following commands in COMAL:  
(i) CAT   (ii) CATALOG   (iii) LIST P63.
4. Write a program to fill an array VALUES with numbers and then to print out the numbers in the odd positions, i.e. VALUES (1), VALUES (3), etc., followed by the numbers in the even positions, VALUES (2), VALUES (4), VALUES (6), etc.
5. Write a program to read in a list of numbers, double each number and print the new list.

6. Given that  $A = 5$ ,  $B = 8$ ,  $C = 2$ , determine whether the following are true or false:
  - (i)  $A < B$ ; (ii)  $B < C$ ; (iii)  $A \geq C$ ; (iv)  $(A < B)$  AND  $(B < C)$ ;
  - (v)  $(A < B)$  OR  $(B < C)$ ; (vi)  $(A < C)$  OR  $(\text{NOT}(B < C))$ ;
  - (vii)  $B < \text{LEN}(\text{"JOHNNIE"})$
7. Write programs in COMAL to read in a list of integers and
  - (a) count the number of odd entries
  - (b) count the number of even entries
  - (c) count the number of entries whose digits add up to 21
  - (d) count the number of entries in each of the following categories:
    - A: 85-100
    - B: 70-84
    - C: 55-69
    - D: 40-54
    - E: 25-39
    - F: 10-24
    - G: <10
8. Write a single COMAL program to perform all the tasks in Question 7.
9. Write programs to read in two lists of numbers and
  - (a) add corresponding elements in the lists to produce a new list
  - (b) multiply corresponding elements
  - (c) determine which of the two lists has the higher average
  - (d) determine how many pairs of corresponding entries are the same.
10. What is a string? What is a substring? Describe clearly, using examples, how to extract a substring from a string.
11. What is the difference between INPUT and "INPUT", between "457" and 457, between JOE and "JOE"?
12. How would you select the words ROPE and POOR from MICROPROCESSOR?
13. Change Program 62 to allow for different lengths of christian name and surname.
14. Write programs to read a string and
  - (1) print only the characters in the odd positions
  - (2) print only every third character
  - (3) count the number of E's
  - (4) count the number of times the sequence AB appears
  - (5) insert a space between successive characters.
15. Write a program to read a piece of text and replace each full stop by an asterisk.
16. Write a program to accept a string and rearrange the characters of the string in all possible ways, e.g. CAT, CTA, ACT, ATC, TAC, TCA.

# 11 Two-dimensional Arrays

## Across the board

Consider this picture. It shows the position, at some stage, of a game of 'noughts and crosses'. The game makes use of a grid or 2-dimensional array. Let us ask the question - how can we represent such an array in a computer program?

X	O	

The array is considered to consist of 3 rows and 3 columns, and we make use of this fact to describe the array. It is called a  $3 \times 3$  array.

Remember how we used a subscript or index to describe the position of an item in a list, e.g.  $L(5)$  indicates the 5th item in a list  $L$ . In the case of a grid we use two subscripts, one for rows and one for columns.

Suppose we call the array A. The position shown here is described as  $A(1,1)$  because it is in the first row and the first column.

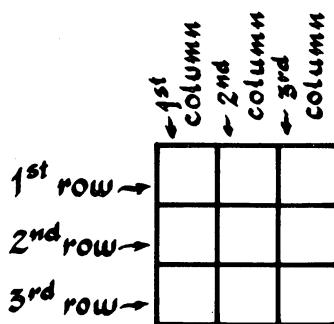
•		

The position shown here is described as  $A(2,3)$  because it is in the second row and the third column.

		•

Two important points are:

- (1) Rows run across.
- Columns run down.



- (2) The row number is always written first and the column number second. Thus,  $A(2,3)$  refers to the 2nd row, 3rd column and **not** the 2nd column, 3rd row.

Looking at the original picture of our game of noughts and crosses we note that there are X's in position A(2,2) and in position A(3,1) and that there are 0's in positions A(1,3) and A(3,2).

Arrays are used in many circumstances, e.g.

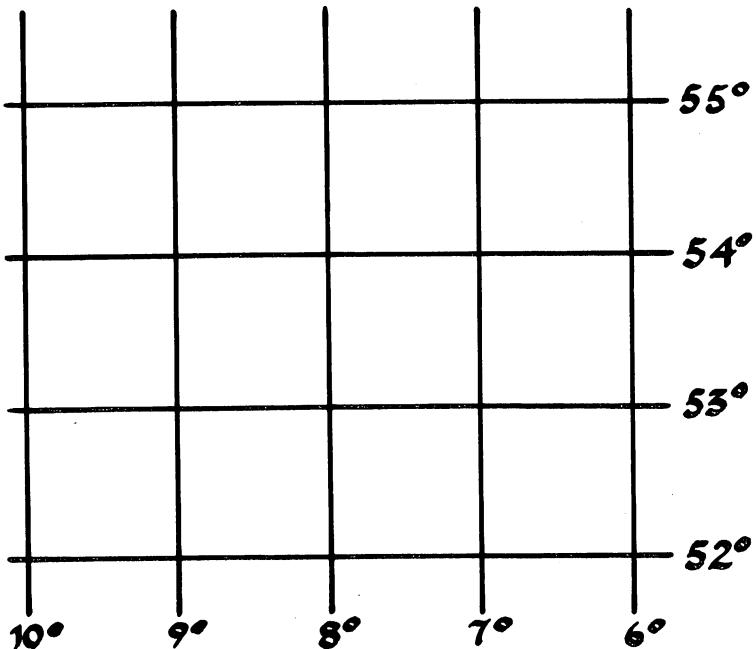
Addition table

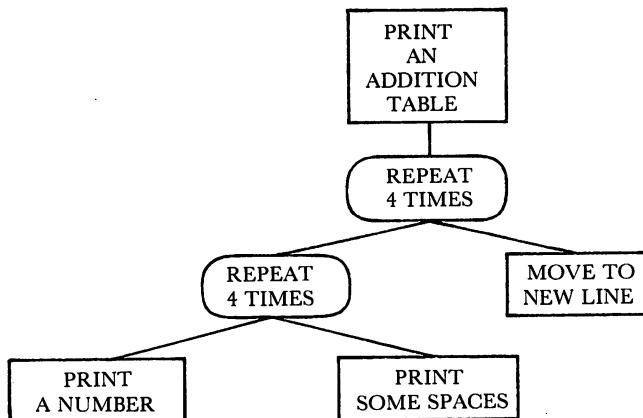
+	1	2	3	4
1	2	3	4	5
2	3	4	5	6
3	4	5	6	7
4	5	6	7	8

Calendar  
June 1981

	Sun	Mon	Tue	Wed	Thu	Fri	Sat
		1	2	3	4	5	6
	7	8	9	10	11	12	13
	14	15	16	17	18	19	20
	21	22	23	24	25	26	27
	28	29	30				

Map Grid





```

0010 // PROGRAM 64
0020 //
0030 // COMELY KATE
0040 //
0050 // TO PRINT AN ADDITION TABLE
0060 DIM TABLE(4,4)
0070 PRINT
0080 FOR ROW:=1 TO 4 DO
0090   FOR COLUMN:=1 TO 4 DO
0100     TABLE(ROW,COLUMN):=ROW+COLUMN
0110     PRINT TABLE(ROW,COLUMN);"    ";
0120   NEXT COLUMN
0130   PRINT
0140   PRINT
0150 NEXT ROW
0160 END

```

RUN

2	3	4	5
3	4	5	6
4	5	6	7
5	6	7	8

Here again we have the classic case of nested loops, the outer loop controlled by the variable ROW and the inner loop controlled by the variable COLUMN. The inner loop variable COLUMN goes through all its 4 values for each value of the outer loop variable ROW.

As usual it is essential that the inner loop be completely enclosed in the outer loop. Therefore NEXT COLUMN must come before NEXT ROW.

Note how the semicolon at the end of the instruction on line 110 causes successive

values to appear on the same line. A string consisting of a few spaces has been included in the print statement to separate numbers being output. (In what other manner could this be done?) When the end of the COLUMN loop has been reached, line 130 causes the existing line of print to be terminated, so that the next PRINT instruction will start a new line. In fact a blank line is printed next (see line 140) so that the following row of numbers will be neatly separated from the last one.

## Summing up again

The output from Program 64 is fairly primitive. It just shows the body of the addition table and omits the numbers across the top and down the left-hand side.

Let us now try to print the table exactly as shown here, including the + sign and the two lines.

```

0010 // PROGRAM 65
0020 //
0030 // COMELY KATE
0040 //
0050 // TO PRINT AN ADDITION TABLE
0060 DIM TABLE$(4,4)
0070 PRINT
0080 PRINT "+ !   ",
0090 FOR COUNT#:=1 TO 4 DO
0100 PRINT COUNT#,"     ",
0110 NEXT COUNT#
0115 PRINT
0130 PRINT "-----"
0140 FOR ROW#:=1 TO 4 DO
0150 PRINT ROW#," !   ",
0160 FOR COLUMN#:=1 TO 4 DO
0170   TABLE$(ROW#,COLUMN#):=ROW#+COLUMN#
0180   PRINT TABLE$(ROW#,COLUMN#);;"      ";
0190 NEXT COLUMN#
0200 PRINT
0210 PRINT
0220 NEXT ROW#
0230 END

```

RUN

+	1	2	3	4
1	2	3	4	5
2	3	4	5	6
3	4	5	6	7
4	5	6	7	8

+	1	2	3	4
1	2	3	4	5
2	3	4	5	6
3	4	5	6	7
4	5	6	7	8

Because all the values are whole numbers we have decided to use integer variables in this version. Let us now provide a commentary on the action.

## Commentary

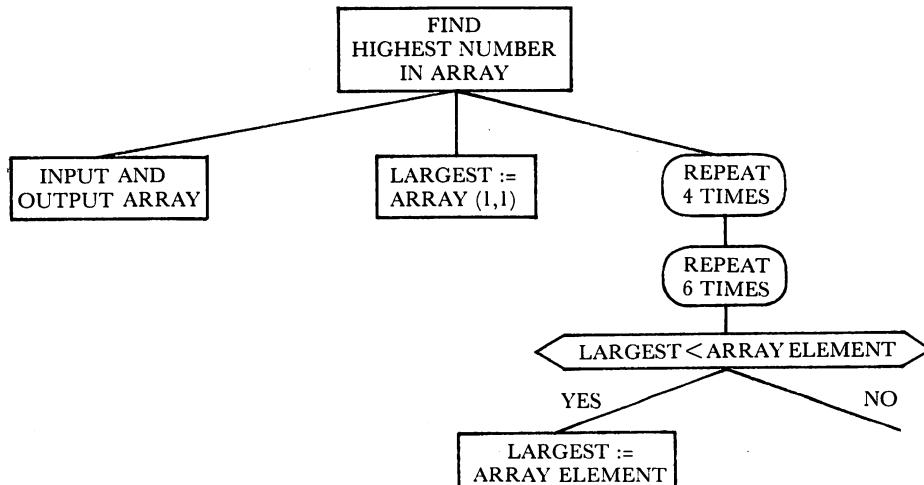
LINE	ACTION
80	causes the + sign and the beginning of the 'down line' to be printed.
90-110	cause the digits 1 to 4 to be printed across the top, with appropriate spaces between them.
115	terminates current print line.
130	prints a line under the top row of digits.
140	begins outer loop controlled by ROW#.
150	prints 'left hand' digit, followed by next part of down line, with suitable spacing.
160	begins inner loop controlled by COLUMN#.
170	calculates sum of 'left' and 'top' digits and puts result in array TABLE#.
180	prints result from array TABLE#, followed by some spaces.
190	increments COLUMN# and checks to see if inner loop is finished.
200	terminates current print line.
210	prints a blank line.
220	increments ROW# and checks to see if outer loop is finished.

You are encouraged to experiment with the above program, e.g. try the effect of omitting some of the semicolons, or line 150, or changing some of the lines.

## Scanning an Array

In order to become thoroughly familiar with the way in which the nested loop structure assists in dealing with arrays, let us write a program to create an array of numbers, and then scan the array to pick out the largest number.

Assume, to begin with, that the array is a  $4 \times 6$  array called simply ARRAY. The following structure diagram expresses the solution.



```

0010 // PROGRAM 66
0020 //
0030 // COMELY KATE
0040 //
0050 // TO FIND THE LARGEST ELEMENT IN AN ARRAY
0060 DIM ARRAY(4,6)
0070 PRINT
0080 // FIRST CREATE THE ARRAY
0090 //
0100 PRINT "THE GIVEN ARRAY IS :"
0110 PRINT
0120 FOR ROW#:=1 TO 4 DO
0130   FOR COLUMN#:=1 TO 6 DO
0140     READ ARRAY(ROW#,COLUMN#)
0150     PRINT ARRAY(ROW#,COLUMN#), "      ",
0160   NEXT COLUMN#
0170   PRINT
0180   PRINT
0190 NEXT ROW#
0200 //
0210 // NOW SEARCH FOR THE LARGEST
0220 //
0230 LARGEST:=ARRAY(1,1)
0240 FOR R#:=1 TO 4 DO
0250   FOR C#:=1 TO 6 DO
0260     IF LARGEST<ARRAY(R#,C#) THEN LARGEST:=ARRAY(R#,C#)
0270   NEXT C#
0280 NEXT R#
0290 PRINT
0300 PRINT "THE LARGEST ELEMENT IS ",LARGEST
0310 DATA 82, 76, 93, 13, 41, 12, 54, 19, 73, 89, 76, 45
0320 DATA 45, 56, 76, 88, 99, 34, 56, 34, 23, 78, 86, 97
0330 END

```

RUN

THE GIVEN ARRAY IS :

82	76	93	13	41	12
54	19	73	89	76	45
45	56	76	88	99	34
56	34	23	78	86	97

THE LARGEST ELEMENT IS 99

Lines 120 to 190 form a nested loop structure which reads in and prints the array.

Note that the array values do not have to be laid out in the DATA lines in the form of the array. It is the logic of the double loop structure which forms them into the required  $4 \times 6$  array.

The search for the largest value begins by taking the first entry, ARRAY (1,1), as the biggest and storing it in LARGEST. Then the array is scanned row by row, using the double loop structure contained in lines 240 to 280 to find a larger element if possible.

LARGEST is compared to each array element on line 260. If it is less then it is changed to the current array value. Otherwise we just pass on to the next element.

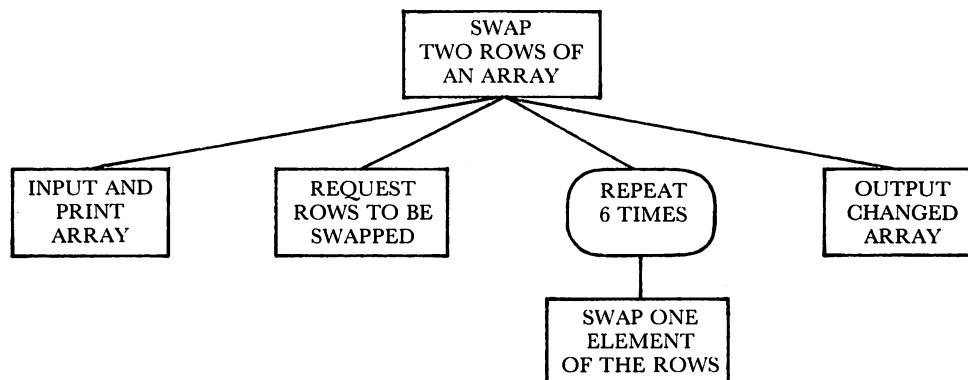
**Note** (1) We are not obliged to use the same variable names for the row and column subscripts in the first double loop as in the second. Thus we use ROW# and COLUMN# in the first, R# and C# in the second.

**Note** (2) We have used an 'ordinary' numeric identifier ARRAY (without the #) for the array to allow for the possibility of decimal values. But since the row and column indices must be integers, we decided to use integer variables for these.

## 'Swop Shop'!

**Problem:** Read in an array and swap two of its rows e.g.

82	76	93	13	14	12		12	13	18	12	90	16
54	19	72	18	73	12	Swop Row 1 and Row 4	54	19	72	18	73	12
31	32	46	40	89	95		31	32	46	40	89	95
12	13	18	12	90	16		82	76	93	13	41	12



```
0010 // PROGRAM 67
0020 //
0030 // COMELY KATE
0040 //
0050 // TO SWAP TWO ROWS OF AN ARRAY
0060 DIM ARRAY(4,6)
0070 PRINT
0080 // FIRST CREATE THE ARRAY
0090 //
0100 PRINT "THE GIVEN ARRAY IS :"
0110 PRINT
0120 FOR ROW#:=1 TO 4 DO
0130   FOR COLUMN#:=1 TO 6 DO
0140     READ ARRAY(ROW#,COLUMN#)
0150     PRINT ARRAY(ROW#,COLUMN#), "    ",
0160   NEXT COLUMN#
0170 PRINT
0180 PRINT
```

```

0190 NEXT ROW#
0200 //
0210 // ASK USER WHICH ROWS HE WISHES TO HAVE SWAPPED
0220 //
0230 INPUT "ENTER THE ROW NUMBERS OF THE ROWS YOU    WISH TO HAVE SWAPPED ";
ROW1#, ROW2#
0240 PRINT
0250 //
0260 // SWAP THE ROWS
0270 //
0280 FOR COLUMN#:=1 TO 6 DO
0290   TEMPORARY:=ARRAY(ROW1#,COLUMN#)
0300   ARRAY(ROW1#,COLUMN#):=ARRAY(ROW2#,COLUMN#)
0310   ARRAY(ROW2#,COLUMN#):=TEMPORARY
0320 NEXT COLUMN#
0330 //
0340 // NOW PRINT THE CHANGED ARRAY
0350 //
0360 PRINT
0370 PRINT "THE CHANGED ARRAY IS :"
0380 PRINT
0390 FOR R#:=1 TO 4 DO
0400   FOR C#:=1 TO 6 DO
0410     PRINT ARRAY(R#,C#),",      "
0420   NEXT C#
0430 PRINT
0440 PRINT
0450 NEXT R#
0460 DATA 82, 76, 93, 13, 41, 12, 54, 19, 73, 89, 76, 45
0470 DATA 45, 56, 76, 88, 99, 34, 56, 34, 23, 78, 86, 97
0480 END

```

RUN

THE GIVEN ARRAY IS :

82	76	93	13	41	12
54	19	73	89	76	45
45	56	76	88	99	34
56	34	23	78	86	97

ENTER THE ROW NUMBERS OF THE ROWS YOU WISH TO HAVE SWAPPED 1,4

THE CHANGED ARRAY IS :

56	34	23	78	86	97
54	19	73	89	76	45
45	56	76	88	99	34
82	76	93	13	41	12

Program 67 is the same as Program 66 up to line 190. Then the person using the program is asked to specify which rows he wants swapped.

The swapping-loop is contained in lines 280 to 320. Take careful note of how the swapping of each two elements is performed. It would not do to just write

```
ARRAY (ROW 1#, COLUMN#) := ARRAY (ROW 2#, COLUMN#)
ARRAY (ROW 2#, COLUMN#) := ARRAY (ROW 1#, COLUMN#)
```

Why not?

One of the elements (ARRAY(ROW 1#, COLUMN#)) in this case is stored temporarily in location TEMPORARY. Why? You should work out for yourself the answers to these two questions and you should carefully trace through the action of the swapping-loop.

### Note on Style

Program 67 may be used without alteration to change any two rows. This makes the program generally useful. If, instead of using ROW 1# and ROW 2# we had used fixed numbers, say 1 and 4 instead, then the program could only be used to swap rows 1 and 4 and would have to be changed if two other rows were to be swapped. It is good to use variables like this to make your programs more general.

In fact the program could be further improved and made still more general. How?

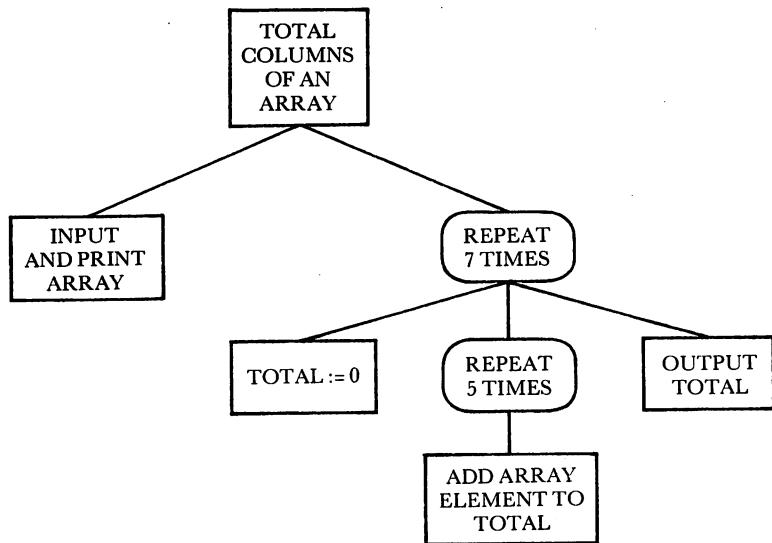
In the above program, since we had decided to use variables ROW 1# and ROW 2#, we went further and allowed the user to specify what the variables should be. This also is a useful feature to include in programs, namely to allow a user to interact with the program. The INPUT command, of course, has allowed us to do this again and again.

## Totting Up

Let us tackle one more little problem using a number array before we pass on to string arrays.

Read in a set of 35 numbers and form them into a  $5 \times 7$  array. Then find the sum of the numbers in each column. print the array and the column sums.

e.g.	48.6	50.1	42.8	60.3	73.2	45.8	50.0
	50.8	52.5	40.4	55.8	68.9	50.2	48.6
	61.7	62.6	48.9	58.9	80.4	53.5	51.6
	49.2	60.4	45.8	59.2	67.6	48.1	49.4
	42.6	62.6	46.3	59.5	68.9	52.6	45.6
TOTALS	252.9	288.2	224.2	293.7	359.0	250.2	245.2



```

0010 // PROGRAM 68
0020 //
0030 // COMELY KATE
0040 //
0050 // TO TOTAL COLUMNS OF AN ARRAY
0055 TAB:=4
0060 PRINT
0070 DIM ARRAY(5,7)
0080 //
0090 // READ IN THE NUMBERS
0100 //
0110 FOR R#:=1 TO 5 DO
0120   FOR C#:=1 TO 7 DO
0130     READ ARRAY(R#,C#)
0140     PRINT ARRAY(R#,C#),
0150   NEXT C#
0160   PRINT
0170 NEXT R#
0180 PRINT
0190 //
0200 // NOW ADD COLUMNS
0210 //
0220 FOR C#:=1 TO 7 DO
0230   TOTAL:=0
0240   FOR R#:=1 TO 5 DO
0250     TOTAL:=TOTAL+ARRAY(R#,C#)
0260   NEXT R#
0270   PRINT TOTAL,
0280 NEXT C#
0290 //
0300 DATA 48.6, 50.1, 42.8, 60.3, 74.2, 45.8, 50.0
0310 DATA 50.8, 52.5, 40.4, 55.8, 68.9, 50.2, 48.6
0320 DATA 61.7, 62.6, 48.9, 58.9, 80.4, 53.5, 51.6
0330 DATA 49.2, 60.4, 45.8, 59.2, 67.6, 48.1, 49.4
0340 DATA 42.6, 62.6, 46.3, 59.5, 68.9, 52.6, 45.6
0350 END
  
```

**RUN**

48.6	50.1	42.8	60.3	74.2	45.8	50
50.8	52.5	40.4	55.8	68.9	50.2	48.6
61.7	62.6	48.9	58.9	80.4	53.5	51.6
49.2	60.4	45.8	59.2	67.6	48.1	49.4
42.6	62.6	46.3	59.5	68.9	52.6	45.6
252.9	288.2	224.2	293.7	360	250.2	245.2

Once again the nested loop structure, elegantly expressed by COMAL FOR statements, appears twice in the program.

First, lines 110 to 170 cause the numbers to be read in, formed into a  $5 \times 7$  array, and printed out in this form. The comma at the end of line 140 keeps the numbers on the same line until C# goes from 1 to 7, i.e. a whole row is printed. The PRINT command on line 160 then causes that line to be terminated and a new line to be set up for the next row.

The second double loop runs from line 220 to 280. This time C# and R# are reversed because we work down along a column in each inner loop. For example, when C# = 1, R# goes from 1 to 5 and the five numbers in column 1 are added. The resulting TOTAL is printed on line 270 when the inner loop has done its job. Since we want the totals to appear across a line at the bottom of our array we use a comma at the end of line 270.

It is essential to initialise TOTAL to zero on line 230. If we omitted to do this we would get the wrong totals. Try it and see!

## String Arrays

We have seen that a string is essentially a one-dimensional array or row of characters. A list of strings is therefore a two-dimensional array where each row of the array forms a single string, e.g.

JOE SOAP
DILLY DREAMER
BILLY BONES
JOEY DE FENCE
:

The 1st row contains the string "JOE SOAP". The 3rd row contains "BILLY BONES" and so on.

How do we declare such an array in a program?  
Quite simply by using the DIM statement as follows:

DIM LISST\$ (50) OF 20

As usual the \$ sign at the end of an identifier indicates a string variable. In effect the declaration says that LISST\$ is to be capable of holding up to 50 entries, where each entry is to be a string of up to 20 symbols.

How do we use such an array? For example, how would we put DILLY DREAMER into the 2nd position?

Again the answer is simple -

LISST\$(2) := "DILLY DREAMER"

And if we wished to put the list entry "JOEY DE FENCE" into a variable called SHADY\$, we would say

SHADY\$ := LISST\$ (4)

Notice that LISST\$ (4) refers to a whole string here, not merely the 4th character in a string.

How would we refer to the 4th character of this array element?

By using LISST\$ (4,4).

What about the 7th character? Use LISST\$ (4,7). Note that the first digit refers to the string itself, i.e. the 4th entry in the list of strings, while the second digit refers to the character within the string, i.e the 7th character.

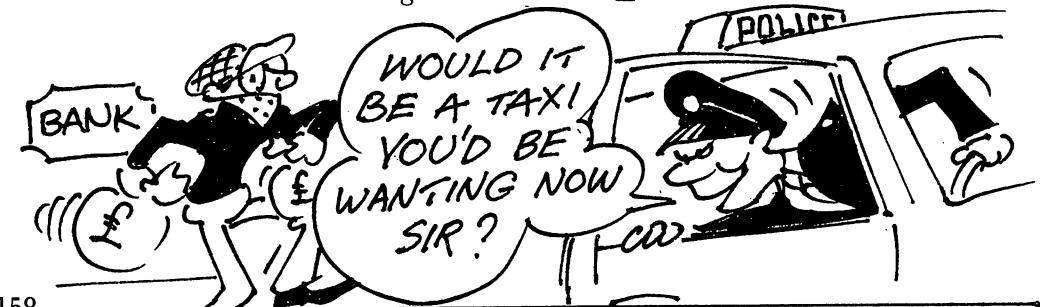
Now what about substrings? How would we refer to the substring DE within the string JOEY DE FENCE? We would use

LISST\$ (4, 6, 7)

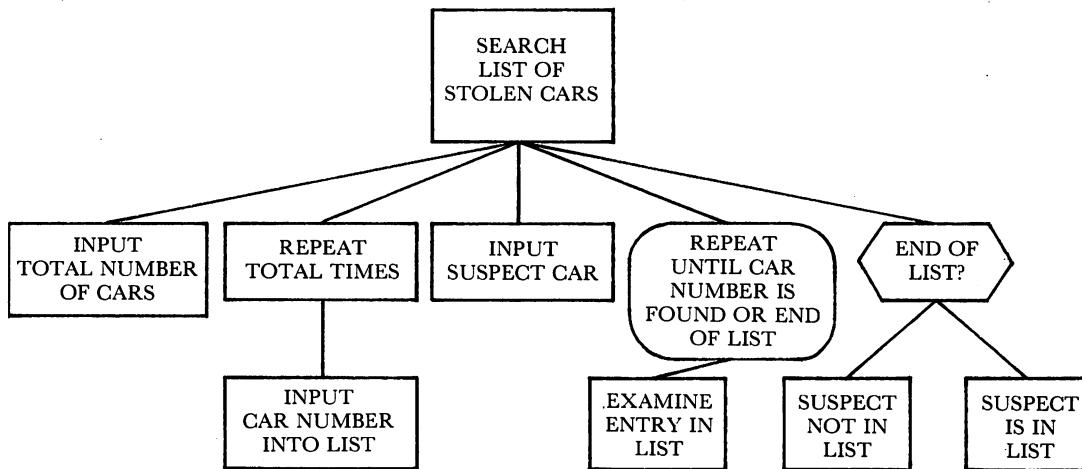
i.e. the substring from the 6th to the 7th symbol inclusive within the string which is the 4th entry in the array LISST\$.

## The Arm of the Law

We live in an imperfect world. Cars are sometimes stolen! People don't like it when their cars are stolen. The police don't like it when people don't like it when their cars are stolen. We must do something about it. We will!



We will write a program to set up a list of stolen cars and then to search this list if a suspected car is reported. The list will actually contain car numbers. Since these are in fact strings, not numbers, we will use a list of strings.



```

0010 // PROGRAM 69
0020 //
0030 // COMELY KATE
0040 //
0050 // TO DEAL WITH A LIST OF STOLEN CARS
0060 //
0070 DIM STOLENCARS$(50) OF 6, SUSPECT$ OF 6
0080 //
0090 // FIRST CONSTRUCT A LIST OF STOLEN CAR NUMBERS
0100 //
0110 READ TOTAL
0120 FOR COUNT:=1 TO TOTAL DO
0130   READ STOLENCARS$(COUNT)
0140 NEXT COUNT
0150 //
0160 // NOW SEARCH LIST FOR SUSPECT CAR
0170 //
0180 PRINT
0190 INPUT "ENTER SUSPECT CAR NUMBER  ": SUSPECT$
0200 PRINT
0210 POSITION:=1
0220 REPEAT
0230   POSITION:=POSITION+1
0240 UNTIL POSITION>TOTAL OR STOLENCARS$(POSITION)=SUSPECT$
0250 IF POSITION>TOTAL THEN
0260   PRINT SUSPECT$,"DOES NOT APPEAR IN LIST OF STOLEN CARS"
0270 ELSE
0280   PRINT SUSPECT$,"  IS IN LIST OF STOLEN CARS"
0290 ENDIF
0300 DATA 15
0310 DATA "5160NI", "AZH457", "AZH241", "JZH333", "LYI405", "ZMI205", "HMI987"
0320 DATA "KNI254", "5120NI", "ZNI123", "ZNI587", "LAI257", "OZH589", "KZH258",
     "875ENI"
0330 END
  
```

```
RUN  
ENTER SUSPECT CAR NUMBER      5120NI  
5120NI      IS IN LIST OF STOLEN CARS  
RUN  
ENTER SUSPECT CAR NUMBER      455AZH  
455AZH      DOES NOT APPEAR IN LIST OF STOLEN CARS
```

We have used READ and DATA to construct our list of stolen cars. Input would be far too slow in practice. Our main objective is to be in a position to check through the list quickly when a suspected stolen car is spotted. It would be hopeless having to enter the list, car number by car number, and then search it. It would be far, far quicker to scan through the list by eye, without the aid of the computer, even if the list contained several hundred entries.

However, if we have the data prepared beforehand in DATA statements, and use READ instead of INPUT, we will speed up enormously the job of creating the list. Of course it is still tedious typing in the contents of all the data statements, but once they are in they are there for good. The real point is that when the program is run the list is created almost instantaneously and is not slowed down by our typing speed.

Note how simple our search loop is (lines 220 to 240). All the loop does is keep moving through the list one position at a time until one or other of the terminating conditions is met, i.e.

(a) we have moved past the end of the list —

POSITION > TOTAL

or

(b) we have encountered the suspect car —

STOLENCAR\$ (POSITION) = SUSPECT\$

When the loop has terminated we check to see if in fact we have gone through the list without finding the car number we were looking for. If we have (line 250) then we print a message to the effect that the car does not appear in the list of stolen cars. Otherwise (see lines 270, 280) we announce that the car *does* appear on the list.

### Summary

Two-dimensional arrays are required to represent many different types of information.

Two indices or subscripts are used to identify an entry in a 2-dimensional array.

An array of strings is essentially a two-dimensional array.

A substring of an array element may be easily extracted by using appropriate subscripts.

## QUESTIONS and EXERCISES

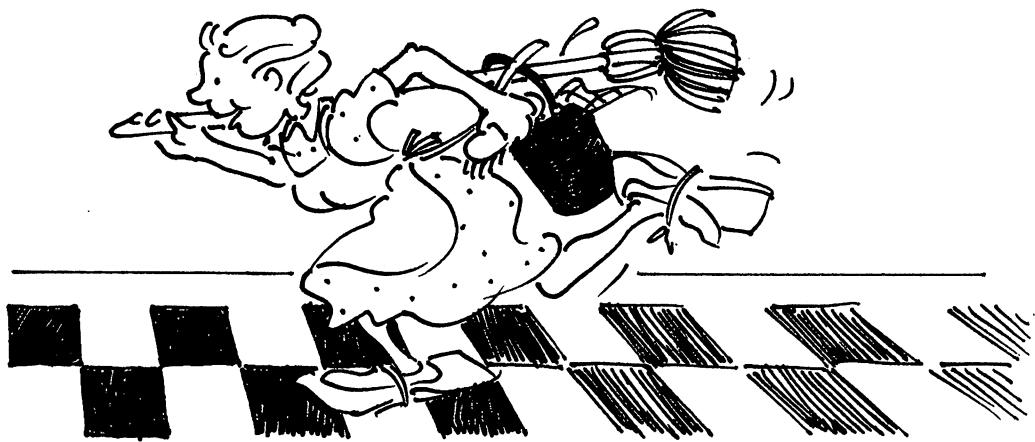
1. The two-dimensional array TABLE is shown in the diagram. What are the dimensions of TABLE? Write down the TABLE positions of each of the entries 1 to 6.

TABLE

	1				
			2		
		3			6
4					
				5	

2. Write a program to print a  $5 \times 5$  multiplication table.
3. Write programs to read in a 2-dimensional array of numbers and
- (i) print them column by column
  - (ii) print them row by row in reverse order
  - (iii) find the smallest element
  - (iv) transpose the array, i.e. change columns into rows and rows into columns
  - (v) find the entries which are the smallest in a row but the largest in a column, if any.
4. Write a program to swap any two columns of a given matrix (i.e. array).
5. Write a program to create a square array which has zero in every position except the leading diagonal positions, which are to have 1, as shown. This is usually called an *identity or unit matrix*.
- |   |   |   |   |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 |
6. Write down the DIM statements required to declare
- (a) a  $10 \times 16$  matrix of integers
  - (b) a list of 50 strings each of possible length 15 symbols.
7. Write a program to read in a list of names and search for a given name.
8. Write a program to read in a list of names and to count all those
- (a) whose christian name is JOHN
  - (b) whose surname is JONES.
9. Write a program to print out a calendar for 1983.

# 12 Functions



## Getting the job done

The word **function** is very often associated with job, or work, or task. When we say 'the cleaner's *function* is to keep the school clean', we usually mean his *job* is to keep the school clean. Similarly in programming we have seen that instructions have their own jobs to do: PRINT has a certain job to do; INPUT has a certain job to do; +, -, \*, etc. have their own jobs to do.

Think carefully for a moment about the effect of an operator like +. For example, consider the statement

$$2 + 5 = 7$$

We may consider this as expressing the fact that the operator + transforms the pair of numbers 2, 5 into the single number 7. In general, + transforms pairs of numbers into single numbers. We can express this neatly, using our fundamental INPUT - PROCESS - OUTPUT description:

INPUT	PROCESS	OUTPUT
2,5	+	7
13,9	+	22
any pair	+	single number

Summing up, then, we can say that when + gets to work on a pair of numbers it produces a single number, just as when our busy harassed cleaner gets to work on a dirty school he transforms it into a shining palace for student princes!

$$\begin{array}{c} +(2,5) \xrightarrow{\text{produces}} 7 \\ \text{cleaner (dirty school)} \xrightarrow{\text{produces}} \text{clean school} \end{array}$$

Similarly  $*(2,5) \xrightarrow{\text{produces}} 10$ , which says that ‘multiply’ working on the pair 2,5 produces the number 10.

Now for a little excitement! In computing we often stand the word ‘function’ on its head. We call + itself a function.

Thus, instead of saying that the function of + is to transform a pair of numbers into a single number, we say that + *is a function which transforms pairs into single numbers*.

Similarly \*, -, / are functions which transform pairs of numbers into single numbers, according to different rules in each case, e.g.

$$\begin{array}{l} -(7,2) \rightarrow 5 \\ *(7,2) \rightarrow 14 \\ /(7,2) \rightarrow 3.5 \end{array}$$

The numbers 7 and 2 are called the **arguments**, or **parameters**, of the function in each case and the numbers 5, 14 and 3.5 are the **values of the functions** -, \*, / respectively.

## Functioning at a higher level

It is time now to meet some new functions in COMAL and to renew acquaintance with some we have already met. We don’t *usually* call +, -, \*, etc. functions, but the ones we are going to discuss in this chapter are *always* called functions. The first one we will consider is the TAB function.

We have already seen that TAB can be used to establish the width of a print-zone for items separated by commas. In fact the TAB function can be used very flexibly to control the format of print statements, e.g.

PRINT TAB (15), “\*”

causes 15 spaces or blanks to be printed, followed immediately by an asterisk.

Similarly

PRINT TAB (22), “HI THERE”

causes 22 spaces to be printed, followed by the message HI THERE.

Consider the following little program:

```
0010 // PROGRAM 70
0020 //
0030 // COMELY KATE
0040 //
0050 // TO ILLUSTRATE USE OF TAB
0060 //
0070 PRINT TAB(10),"HELLO"
0080 PRINT TAB(6),"HELLO"
0090 PRINT TAB(2),"HELLO"
0100 END
```

RUN

```
        HELLO
    HELLO
HELLO
```

As you can see the TAB function saves us the trouble of including exactly the correct number of spaces between quote marks, or using a loop to achieve the same effect. It allows us to be more adventurous and yet more economical in creating output.

It is worth while examining a few more programs to illustrate the TAB function.

We will use the command CLEAR to clear the screen for each of our picture-drawing programs.

**Example 1** Draw this square in the middle of the screen. We will assume that the screen is 40 characters wide, i.e. that you can print a maximum of 40 characters across it.

```
*****  
*   *  
*   *  
*   *  
*   *  
*   *  
*****
```

```
0010 // PROGRAM 71
0020 //
0030 // COMELY KATE
0040 //
0050 // TO PRINT A SQUARE
0060 //
0065 CLEAR
0070 // PRINT TOP LINE
0080 PRINT TAB(15),"*****"
0090 //
0100 // PRINT NEXT 8 LINES
0110 //
0120 FOR LINE#:1 TO 8 DO
0130   PRINT TAB(15),"*",TAB(24),"*"
0140 NEXT LINE#
0150 //
0160 // PRINT BOTTOM LINE
0170 PRINT TAB(15),"*****"
0180 END
```

RUN

```
*****  
* *  
* *  
* *  
* *  
* *  
* *  
* *  
* *  
* *  
*****
```

Line 80 causes the printer to move in 15 spaces and then print a string consisting of 10 asterisks. Line 130 causes the printer to move in 15 spaces, print an asterisk, then move along another 8 spaces ( $15 + 1 + 8 = 24$ ) and print another asterisk. This is done 8 times within the FOR loop.

Line 170 is exactly the same as line 80 and performs exactly the same action. You should work out carefully why these instructions perform the required tasks. You should also experiment with the program to reduce or enlarge the square.

**Example 2** Write a program to draw the following pattern on the screen

```
-- -- -- -- --  
-- -- -- -- --  
-- -- -- -- --  
etc.
```

```
0010 // PROGRAM 72  
0020 //  
0030 // COMELY KATE  
0040 //  
0050 // TO DRAW A SIMPLE PATTERN  
0060 //  
0065 CLEAR  
0070 FOR LINE#:=1 TO 10 DO  
0080 PRINT TAB(LINE#), "----"  
0090 NEXT LINE#  
0100 END
```

RUN

```
-----  
-----  
-----  
-----  
-----  
-----  
-----  
-----  
-----  
-----
```

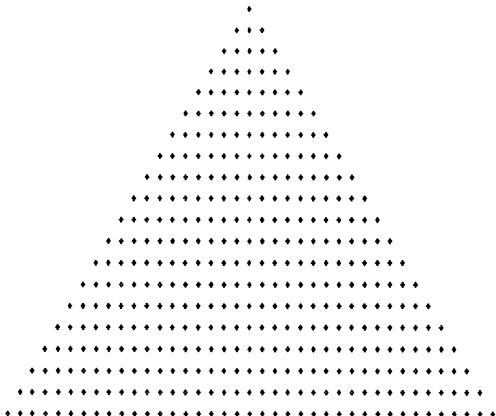
Note that the argument of the TAB function is a variable, namely LINE#. When LINE# = 1, the printer moves in one space, when LINE# = 2, the printer moves in 2 spaces, etc. Thus we get the skewed pattern shown.

**Example 3** Draw a triangle with its apex in the centre of the screen.



```
0010 // PROGRAM 73
0020 //
0030 // COMELY KATE
0040 //
0050 // TO DRAW A TRIANGLE
0060 //
0065 CLEAR
0070 FOR LINE#:=1 TO 20 DO
0080   PRINT TAB(21-LINE#),
0090   FOR DOT#:=1 TO 2*LINE#-1 DO
0100     PRINT ",";
0110   NEXT DOT#
0120   PRINT
0130 NEXT LINE#
0140 END
```

RUN



Note that the argument of the TAB function, 21-LINE#, decreases as LINE# increases so that the screen cursor moves in fewer spaces as we move down the screen. This is exactly what is needed.

Also notice how neatly we get an increasing number of dots on successive lines. We use  $2 * \text{LINE\#} - 1$  to give us this number. When  $\text{LINE\#} = 1$ ,  $2 * \text{LINE\#} - 1 = 1$ , so one dot will be printed on the top line. When  $\text{LINE\#} = 2$ ,  $2 * \text{LINE\#} - 1 = 3$ , so 3 dots will be printed on the second line, and so on.

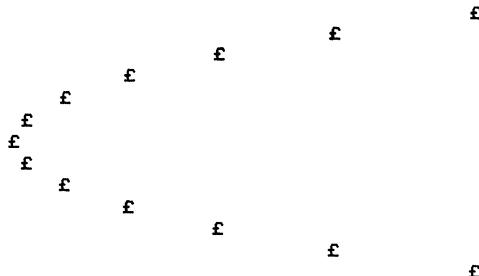
The last two examples show how variable arguments in the TAB function allow variable spacing to be achieved. Interesting results can be produced in this way. For example, what would you call the shape produced on the screen by the following program?

```

0010 // PROGRAM 74
0020 //
0030 // COMELY KATE
0040 //
0050 // TO DRAW A SIMPLE CURVE
0060 //
0065 CLEAR
0070 FOR X:=-6 TO 6 DO
0080 PRINT TAB(X*X+2), "£"
0090 NEXT X
0100 END

```

RUN



## SIN

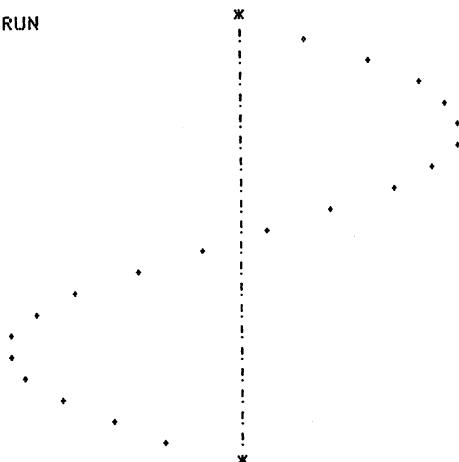
We can draw further interesting graphs by making use of some of the mathematical functions. Consider the function **SIN** (short for SINE). The SIN function takes an angle expressed in radian measure as its argument and produces a number between -1 and +1 as a result. The values of the SIN function for different angles may be found in mathematical tables. The COMAL SIN function also generates these values.

Let us draw a graph of the SIN function:

```

0010 // PROGRAM 75
0020 //
0030 // COMELY KATE
0040 //
0050 // TO DRAW A SINE GRAPH
0060 //
0065 CLEAR
0070 FOR RADIAN$:=0 TO 6.5 STEP 0.3 DO
0080 POSITION:=INT(18*SIN(RADIAN$))
0090 IF POSITION=0 THEN
0100 PRINT TAB(19), "*"
0110 ELIF POSITION<0 THEN
0120 PRINT TAB(19+POSITION), ".", TAB(19), "!"
0130 ELSE
0140 PRINT TAB(19), "!", TAB(19+POSITION), "."
0150 ENDIF
0160 NEXT RADIAN$
0170 END

```



The variable RADIANS is used to hold the value of the angle in radians. It is allowed to run from 0 to 6.5 in steps of 0.3 (line 70). This is equivalent to going from 0 degrees to about 360 degrees in steps of about 20 degrees.

Since SIN (RADIANS) ranges from -1 to +1, we multiply it by 18 to expand the range to -18 to +18. This gives us a range of 36, which is close to the screen width of 40. The idea is to have 0 in the centre of the screen, negative values to the left and positive values to the right.

Line 80 calculates the integer part of 18 SIN (RADIANS). The COMAL function INT is used for this. INT merely chops off the decimal part of a positive number, or rounds down in the case of a negative number. In effect then, the largest integer less than or equal to the argument is calculated,

e.g.	INT (4.78) = 4
	INT (-3.29) = -3
	INT (-5.87) = -6
	INT (6.0) = 6

The variable POSITION is used to hold the result of INT (18 \* SIN(RADIANS)). If POSITION = 0 then line 100 causes a \* to be printed in the 20th position across a line (first move in 19 spaces with TAB (19)).

If POSITION is negative the cursor is moved in 19 + POSITION spaces, then a dot is printed, then the cursor is moved to position 20 where a stroke to form part of a centre line is printed. Note carefully that since POSITION is negative 19 + POSITION is less than 19 and therefore left of centre.

If POSITION is neither zero nor negative, then line 140 will be executed. This causes the cursor to move in 19 spaces, print a stroke and then move in a further POSITION spaces before finally printing a dot.

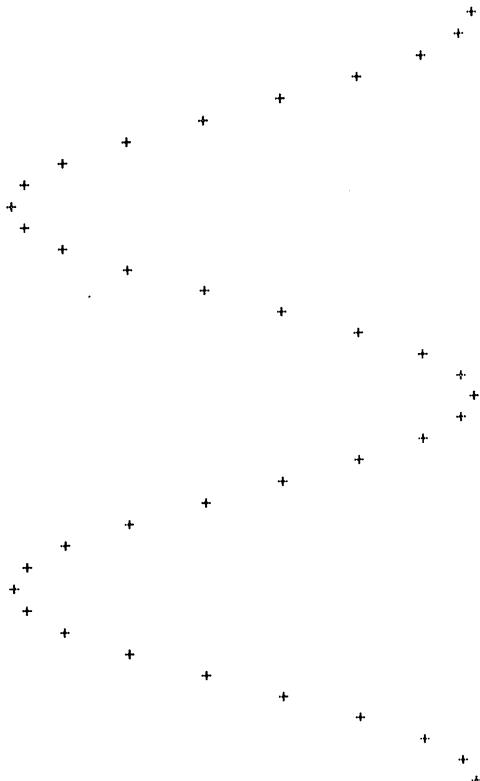
You must think carefully about how these values of POSITION cause a dot to be printed in just the right places to draw a picture of the SIN function.

## Portrait of cousin COS

How about a graph of the function **COS** (short for COSINE) now?

```
0010 // PROGRAM 76
0020 //
0030 // COMELY KATE
0040 //
0050 // TO DRAW A GRAPH OF COSINE
0060 //
0065 CLEAR :
0070 //
0080 PI:=3.14159
0090 CONVERTFACTOR:=PI/180
0100 CLEAR
0110 FOR DEGREES:=0 TO 720 STEP 20 DO
0120   ANGLE:=DEGREES*CONVERTFACTOR
0130   COSVAL:=COS(ANGLE)
0140   POSITION:=20+18*COSVAL
0150   PRINT TAB(POSITION),"@"
0160 NEXT DEGREES
0170 END
```

RUN

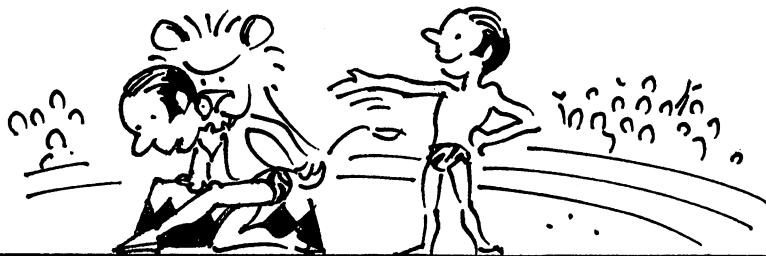


This time we choose to express our FOR loop in terms of degrees rather than radians. We go from 0 to 720 in steps of 20 degrees (line 110). However, the COS function must have its argument in radians, so we must convert from degrees to radians. This is done by multiplying the value of DEGREES by  $\pi$  and dividing by 180. This conversion factor has been calculated and assigned to CONVERTFACTOR in line 90 so that all we have to do is multiply RADIANS by CONVERTFACTOR in line 120 before getting the COS in line 130. This is more efficient than writing

$$\text{ANGLE} := \text{RADIAN} * \text{PI}/180$$

within the loop. Why is this? Because if we wrote line 120 this way the program would have to calculate  $3.14159/180$  each time through the loop, a total of 37 times in all. By evaluating the expression before the loop starts, the calculation is done once only.

Exactly the same considerations apply to the spacing value for the TAB function as applied to the SIN graph, except that we have not drawn a line through the centre here.



## Double Act

Let us now write a program to superimpose the SIN and COS graphs. The main problem here is to work out whether to print the SIN or the COS value first. If the value SINVAL of SIN is less than the value COSVAL of COS then the SIN is printed to the left of COS (see line 220 below). If COSVAL is less, then COS is printed to the left of SIN (line 240). To distinguish between the two graphs we print a dot for SIN and a + for COS.

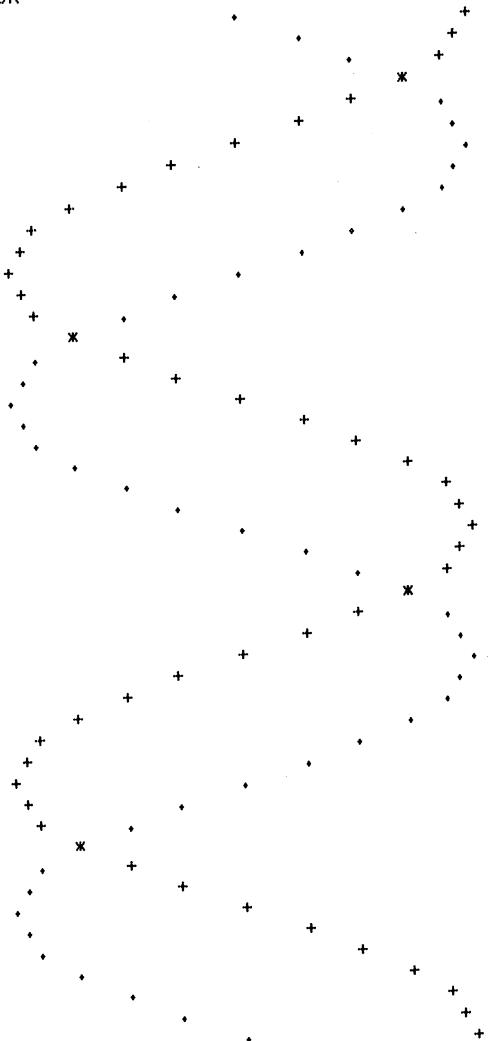
```

0010 // PROGRAM 77
0020 //
0030 // COMELY KATE
0040 //
0050 // TO DRAW A GRAPH OF SINE + COSINE
0060 // -----
0070 //

0080 PI:=3.14159
0090 CONVERTFACTOR:=PI/180
0100 VERYSMALL:=0.001
0110 CLEAR
0120 MIDDLE:=20
0130 FOR DEGREES:=0 TO 720 STEP 15 DO
0140 ANGLE:=DEGREES*CONVERTFACTOR
  
```

```
0150 SINVAL:=SIN(ANGLE)
0160 COSVAL:=COS(ANGLE)
0170 POSITION1:=MIDDLE+18*SINVAL
0180 POSITION2:=MIDDLE+18*COSVAL
0190 IF ABS(POSITION1-POSITION2)<VERYSMALL THEN
0200   PRINT TAB(POSITION1),"*"
0210 ELSEIF POSITION1<POSITION2 THEN
0220   PRINT TAB(POSITION1),".",TAB(POSITION2),"+"
0230 ELSE
0240   PRINT TAB(POSITION2),"+",TAB(POSITION1),"."
0250 ENDIF
0260 NEXT DEGREES
0270 END
```

RUN



You are probably puzzled by line 190. Its job is really to test whether the values of SIN and COS are equal. Why didn't you just write IF POSITION 1 = POSITION 2, I hear you say!

Well the reason is a common-sense down-to-earth one. The values of SIN and COS and therefore of POSITION 1 and POSITION 2 are real numbers. Two real values are very seldom exactly equal. Consider for example two pages of a book. They are considered to be the same size but it is very unlikely that they are. Anyhow it is impossible to measure their lengths and widths absolutely exactly. So we are content to accept two lengths as being equal if there is an immeasurably small difference between them. Sometimes we don't require the difference to be immeasurable - just small will do. In practice we usually measure only to a certain number of decimal places, say 2 or 3. If two numbers agree to within a certain number of decimal places, we say they are equal. This is the stance we adopt in the above program. If POSITION 1 and POSITION 2 agree to within three decimal places we accept them as equal. Thus we define a variable called **VERYSMALL** equal to 0.001.

Since we don't know which is greater, POSITION 1 or POSITION 2, we take the absolute value of their difference on line 190 before testing to see if it is less than **VERYSMALL**. The COMAL function **ABS** is used to get the absolute value.

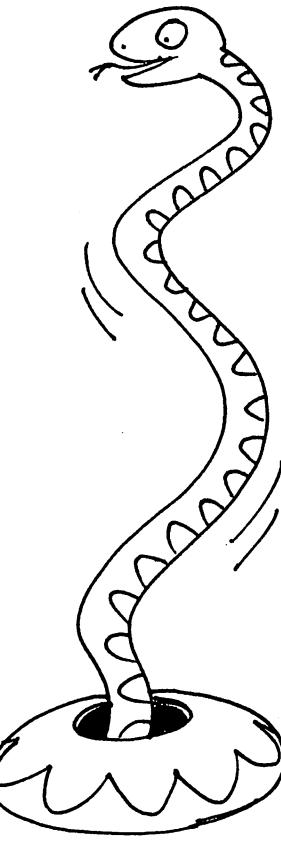
## The Exploding Exponential

As another example of a mathematical graph let us draw the graph of the exponential function **EXP**. EXP(V) calculates the value of  $e^v$  where  $e$  is the base of natural logarithms, e.g. EXP(2) = 7.389.

The values of this function become very big very rapidly, so we will just take a small range: 0 to 3.6.

```
0010 // PROGRAM 7B
0020 //
0030 // COMELY KATE
0040 //
0050 // TO DRAW A GRAPH OF THE EXPONENTIAL FUNCTION
0060 //
0070 CLEAR
0080 FOR X:=0 TO 3.6 STEP 0.4 DO
0090   POSITION:=EXP(X)
0100   PRINT TAB(POSITION), "*"
0110   PRINT
0120 NEXT X
0130 END
```

```
RUN
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
```



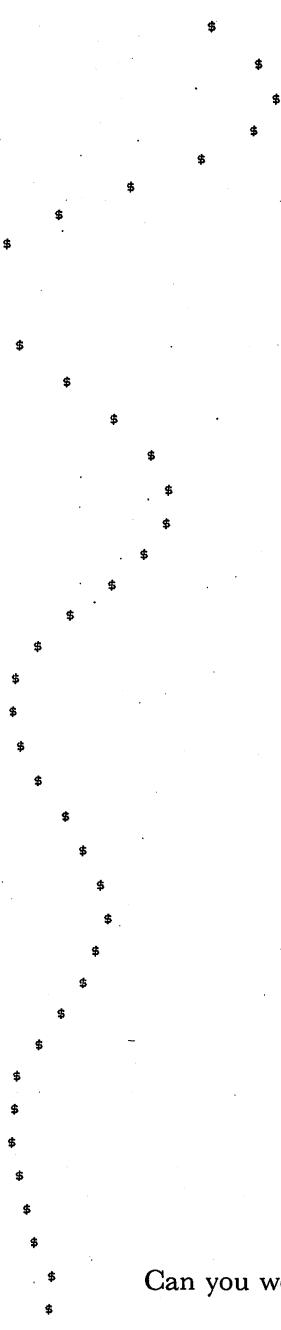
## Taming the Exponential



An interesting graph can be formed by combining the SIN and the EXP functions, as in the following program:

```
0010 // PROGRAM 79
0020 //
0030 // COMELY KATE
0040 //
0050 // TO DRAW A DAMPED OSCILLATION
0060 // -----
0070 //
0080 CLEAR
0090 FOR X:=0 TO 20 STEP 0.5 DO
0100 POSITION:=20+19*EXP(-0.1*X)*SIN(X)
0110 PRINT TAB(POSITION), "$"
0120 PRINT
0130 NEXT X
0140 END
```

RUN



Can you work out for yourself why the shape is as shown?

## Other COMAL functions

COMAL provides a number of other functions besides the ones we have mentioned so far. A list of most of these is given in **Appendix 1**.

### Random Road

Perhaps the most interesting of all the functions available in COMAL is the random number function **RND**. We have used this function already. You will remember that RND on its own produces an unpredictable decimal number in the range 0 ... 1 and that RND(A,B) where A and B are integers produces an unpredictable integer value in the range A ... B inclusive.

The following program displays some numbers of both types.

```
0010 // PROGRAM 80
0020 //
0030 // COMELY KATE
0040 //
0050 // TO DISPLAY SOME RANDOM NUMBERS
0060 //
0070 PRINT "THE FOLLOWING ARE RANDOM NUMBERS IN THE RANGE 0 TO 1"
0080 PRINT
0090 FOR COUNT:=1 TO 10 DO
0100 PRINT RND;
0110 NEXT COUNT
0120 PRINT
0130 PRINT "THE FOLLOWING ARE INTEGER RANDOM NUMBERS IN THE RANGE 1 TO 6"
0140 PRINT
0150 FOR COUNT:=1 TO 20 DO
0160 PRINT RND(1,6);
0170 NEXT COUNT
0180 END

RUN

THE FOLLOWING ARE RANDOM NUMBERS IN THE RANGE 0 TO 1
0.6328993 0.1005926 0.8480765 0.1897004 0.2412089 0.769335 0.7224148 0.9500588
0.0569355 0.9256001
THE FOLLOWING ARE INTEGER RANDOM NUMBERS IN THE RANGE 1 TO 6
4 6 4 5 6 1 1 2 6 3 6 6 2 5 3 4 5 6 1 4
```

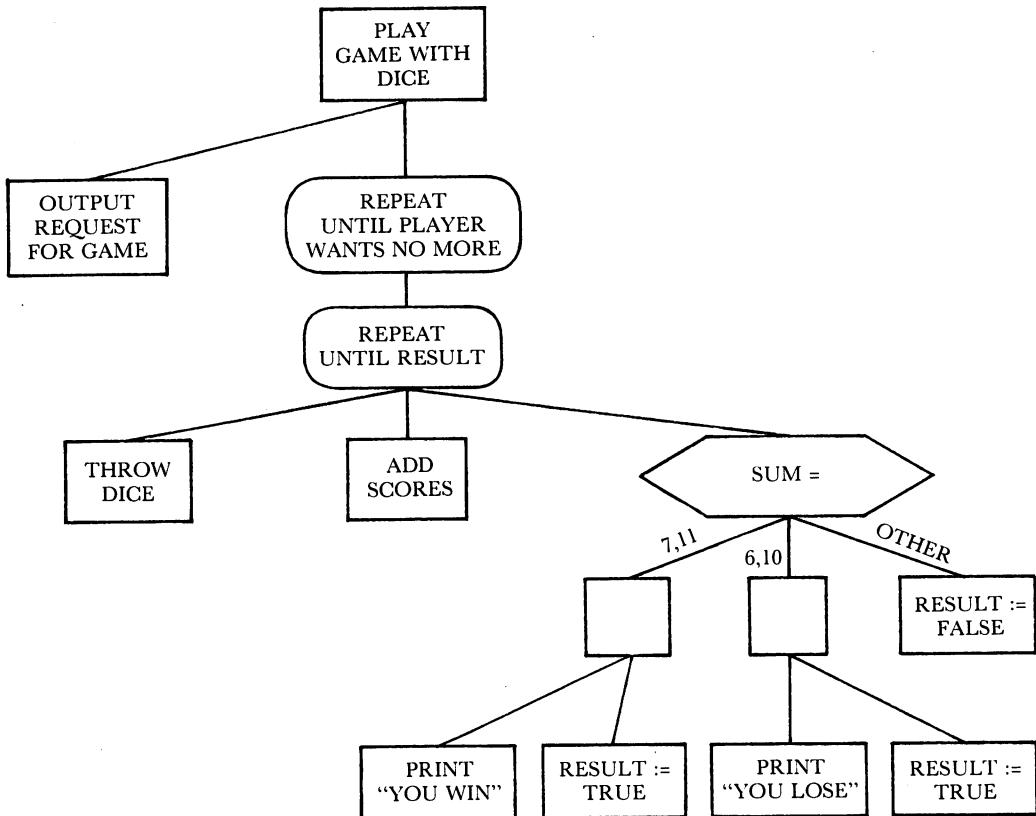
Consider the results in the range 1 to 6. Do you think the results are similar to what they would be if you tossed a real die? Roll a die twenty times and compare the results.

How would you like to roll a die 100 times and examine the results? Or perhaps you would like to throw it 1000 times. Somewhat tedious, eh? The beauty of the above program is that by simply changing the number 20 in line 150, we can simulate as many throws of the die as we wish.

Let us now use our RND function to program the computer to play a simple game.

## Playing the Game

The game is as follows: Two dice are rolled. If the numbers shown add up to 7 or 11, then you win. If the numbers add up to 6 or 10, the computer wins. Otherwise the dice are rolled again.



```
0010 // PROGRAM 81
0020 //
0030 // COMELY KATE
0040 //
0050 // TO PLAY A SIMPLE GAME
0060 //
0070 DIM REPLY$ OF 3
0080 INPUT "WOULD YOU LIKE TO PLAY A GAME WITH DICE ? ":"REPLY$"
0090 PRINT
0100 WHILE REPLY$=="YES" DO
0110 //
0120 // ROLL THE DICE
0130 //
0140 REPEAT
```

```
0150  NUMBER1#:=RND(1,6)
0160  NUMBER2#:=RND(1,6)
0170  PRINT "THE NUMBERS ROLLED ARE ",NUMBER1#," AND ",NUMBER2#
0180  PRINT
0190  PRINT
0200  //
0210  // REPORT RESULT
0220  //
0230  SUM#=NUMBER1#+NUMBER2#
0240  IF SUM#=7 OR SUM#=11 THEN
0250    PRINT "YOU WIN -- WELL DONE"
0260    RESULT#=TRUE
0270  ELIF SUM#=6 OR SUM#=10 THEN
0280    PRINT "GREAT SCOT!! I WIN HA! HA!"
0290    RESULT#=TRUE
0300  ELSE
0310    RESULT#=FALSE
0320  ENDIF
0330 UNTIL RESULT
0340 PRINT
0350 PRINT
0360 INPUT "WOULD YOU LIKE TO PLAY AGAIN ": REPLY$
0370 ENDWHILE
0380 END
```

RUN

WOULD YOU LIKE TO PLAY A GAME WITH DICE ? YES

THE NUMBERS ROLLED ARE 4 AND 1

THE NUMBERS ROLLED ARE 6 AND 2

THE NUMBERS ROLLED ARE 2 AND 5

YOU WIN -- WELL DONE

WOULD YOU LIKE TO PLAY AGAIN YES  
THE NUMBERS ROLLED ARE 5 AND 6

YOU WIN -- WELL DONE

WOULD YOU LIKE TO PLAY AGAIN YES  
THE NUMBERS ROLLED ARE 1 AND 6

YOU WIN -- WELL DONE

WOULD YOU LIKE TO PLAY AGAIN YES  
THE NUMBERS ROLLED ARE 4 AND 6

GREAT SCOT!! I WIN HA! HA!

WOULD YOU LIKE TO PLAY AGAIN NO

### *Commentary*

LINE	ACTION
80	The user is asked if he would like to play.
90	A blank line is printed.
100	If the reply was YES the WHILE loop is entered and a game is played. Note that if the reply was NO the whole WHILE loop - lines 100 to 370 - is skipped and the program ends. The reason we use a WHILE loop rather than REPEAT is to allow for this possibility.
140	An inner loop is entered. The purpose of this loop is to keep throwing two dice until a score of 6, 7, 10 or 11 is achieved. The loop must be done at least once and so REPEAT is appropriate.
150, 160	Two random numbers are computed to correspond to the scores on the two dice.
170	These values are printed.
180, 190	Two blank lines are printed in order to space the output clearly.
230	The sum of the scores is calculated.
240	SUM# is checked to see if it is 7 or 11. If it is ...
250	A suitable message is printed and ...
260	RESULT is set to TRUE to indicate that the inner loop should be terminated.
270	If the sum was not 7 or 11, it is now checked to see if it is 6 or 10. If it is ...
280	A suitable message is printed and ...
290	RESULT is set to TRUE.
300	If none of the above possibilities has arisen ...
310	RESULT is set to FALSE.
320	The IF statement is closed.
330	If RESULT is TRUE the REPEAT loop terminates. If RESULT is FALSE we return to line 150 to perform the REPEAT loop once more.
340, 350	Two more blank lines to separate output.
360	User is asked to play again.
370	WHILE condition is tested. If REPLY\$ is YES the whole process is repeated from line 140.
380	Otherwise the program is terminated.

## Teaching the Computer to teach



The random number function can be used to advantage in situations other than game-playing. We will use it in the construction of a simple program to test students' arithmetic knowledge.

First, however, consider the following rudimentary program.

```
0010 // PROGRAM 82
0020 //
0030 // COMELY KATE
0040 //
0050 // ADDITION QUESTIONS
0060 //
0070 PRINT
0080 PRINT "HERE IS A SIMPLE QUESTION ON ADDITION"
0090 PRINT
0100 INPUT "17 + 13 = "; ANSWER
0110 PRINT
0120 IF ANSWER=30 THEN
0130 PRINT "CORRECT - VERY GOOD"
0140 ELSE
0150 PRINT "SORRY - WRONG ANSWER!"
0160 ENDIF
0170 END
```

RUN

HERE IS A SIMPLE QUESTION ON ADDITION

17 + 13 = 30

CORRECT - VERY GOOD

RUN

HERE IS A SIMPLE QUESTION ON ADDITION

17 + 13 = 27

SORRY - WRONG ANSWER!

The program puts a simple ‘sum’ on the screen, asks for an answer, checks the answer and then prints an appropriate message. However, several improvements could be made:

- (1) If the wrong answer is given, then the correct answer could be printed, along with an appropriate message, e.g.

PRINT “SORRY WRONG ANSWER. THE CORRECT ANSWER IS 30”

- (2) More than one attempt at the answer could be allowed. After a certain number of wrong attempts (e.g. 3) have been given, then the right answer could be printed.
- (3) Instead of using the fixed numbers 17 and 13, we could use variables FIRST and SECOND, say, so that we can vary the sum given, e.g.

PRINT FIRST, “+”, SECOND, “=”,

Notice that we do not put FIRST and SECOND in inverted commas. We want the *numbers* represented by FIRST and SECOND printed, not the *words* FIRST and SECOND. Of course we would have to give FIRST and SECOND values before the PRINT command is executed,

e.g.

FIRST := 17  
SECOND := 13

But, you probably say, isn’t that the same thing as using constants 17 and 13 in the PRINT statement, because every time the program is run FIRST will be given the value 17 and SECOND will be given the value 13, and we will have the same sum as before.

This is where our random number function comes in!

Instead of just setting FIRST and SECOND to fixed values, we use the random number function to give them different values from one run to the next,

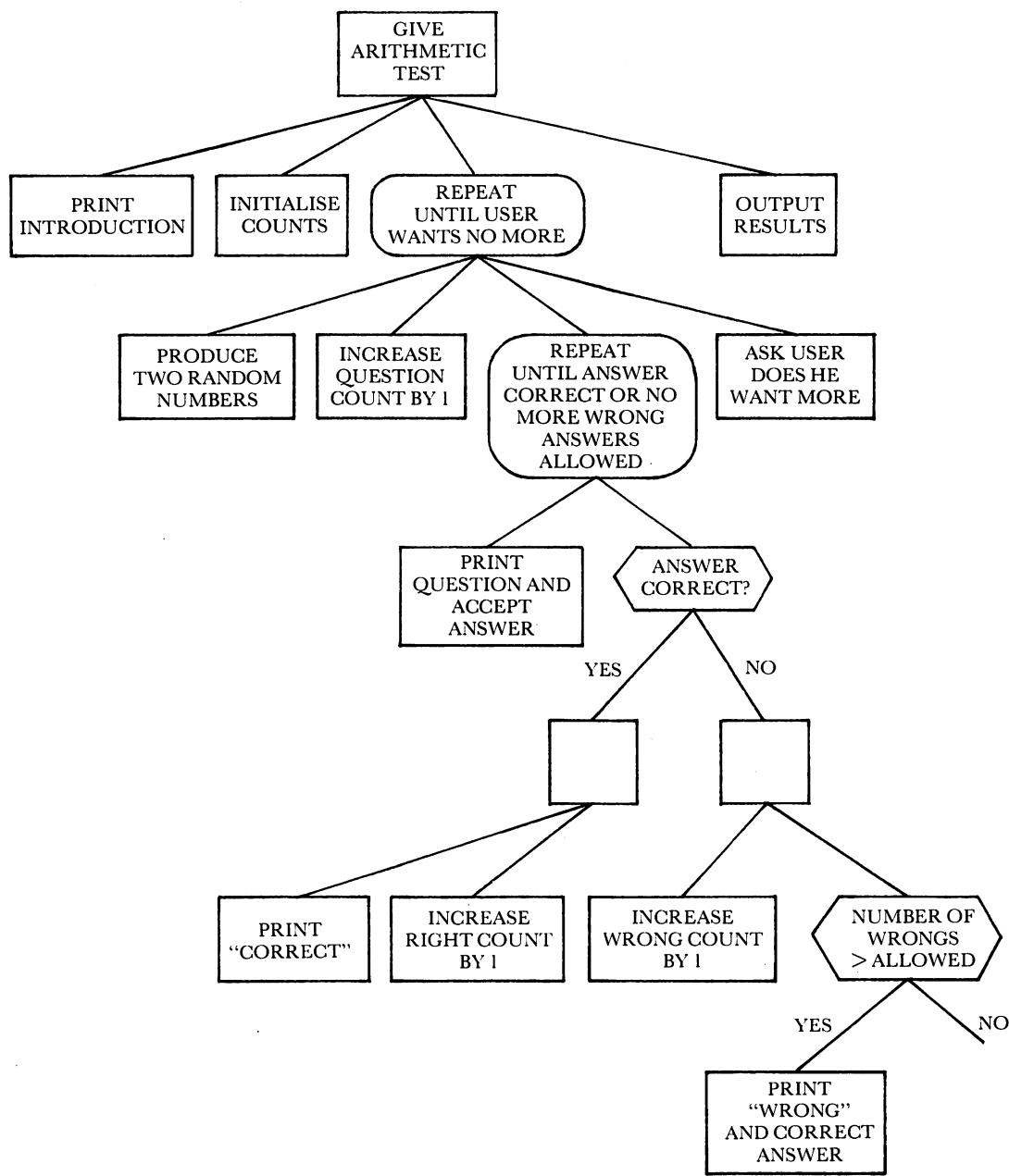
e.g.

FIRST := RND(1,100)  
SECOND := RND(1,100)

These assignment statements will give FIRST and SECOND two unpredictable values between 1 and 100.

- (4) Now that we have a method of varying the question asked, it would be a bit of a nuisance to have to keep asking the program to start again by typing RUN. It would be better to have the program continue automatically with more questions. So why not do it politely, by asking the user does he want to continue, e.g. INPUT “WOULD YOU LIKE ANOTHER QUESTION?”: REPLY\$
- (5) We should keep a count of right and wrong answers, so that we can let the student know his score at the end.

Let us put all these ideas together to produce the following algorithm:



```

0010 // PROGRAM B3
0020 //
0030 // COMELY KATE
0040 //
0050 // ADDITION QUESTIONS
0060 //
0070 DIM REPLY$ OF 3
0080 ALLOWEDWRONG:=3
0090 HIGHESTNUMBER:=100
0100 PRINT
0110 PRINT "HERE ARE SOME SIMPLE QUESTIONS ON           ADDITION"
0120 PRINT
0130 TOTALQUESTIONS:=0
0140 NUMBERRIGHT:=0
0150 REPEAT
0160   FIRST:=RND(1,HIGHESTNUMBER)
0170   SECOND:=RND(1,HIGHESTNUMBER)
0180   NUMBERWRONG:=0
0190   TOTALQUESTIONS:=TOTALQUESTIONS+1
0200   REPEAT
0210     PRINT FIRST," + ",SECOND," = ",
0220     INPUT ANSWER
0230     PRINT
0240     IF ANSWER=FIRST+SECOND THEN
0250       PRINT "CORRECT - VERY GOOD"
0260       NUMBERRIGHT:=NUMBERRIGHT+1
0270       RIGHT:=TRUE
0280     ELSE
0290       PRINT "SORRY - WRONG ANSWER!"
0300       NUMBERWRONG:=NUMBERWRONG+1
0310       PRINT
0320       RIGHT:=FALSE
0330       IF NUMBERWRONG<ALLOWEDWRONG THEN PRINT "TRY AGAIN"
0340     ENDIF
0350   UNTIL RIGHT OR NUMBERWRONG=ALLOWEDWRONG
0360   IF NUMBERWRONG=ALLOWEDWRONG THEN
0370     PRINT "CORRECT ANSWER IS ",FIRST+SECOND
0380     PRINT
0390   ENDIF
0400   PRINT
0410   INPUT "WOULD YOU LIKE ANOTHER QUESTION ? ": REPLY$
0420 UNTIL REPLY$="NO"
0430 PRINT
0440 PRINT
0450 PRINT
0460 PRINT "YOU SCORED ",NUMBERRIGHT," OUT OF ",TOTALQUESTIONS
0470 PRINT
0480 PRINT "BYE - AND GOOD LUCK!"
0490 END

```

RUN

HERE ARE SOME SIMPLE QUESTIONS ON ADDITION

64 + 11 = ? 75

CORRECT - VERY GOOD

WOULD YOU LIKE ANOTHER QUESTION ? YES  
 85 + 19 = ? 104

CORRECT - VERY GOOD

WOULD YOU LIKE ANOTHER QUESTION ? YES  
25 + 77 = ? 102

CORRECT - VERY GOOD

WOULD YOU LIKE ANOTHER QUESTION ? YES  
73 + 96 = ? 169

CORRECT - VERY GOOD

WOULD YOU LIKE ANOTHER QUESTION ? YES  
6 + 93 = ? 100

SORRY - WRONG ANSWER!

TRY AGAIN

6 + 93 = ? 101

SORRY - WRONG ANSWER!

TRY AGAIN

6 + 93 = ? 102

SORRY - WRONG ANSWER!

CORRECT ANSWER IS 99

WOULD YOU LIKE ANOTHER QUESTION ? NO

YOU SCORED 4 OUT OF 5

BYE - AND GOOD LUCK!

You should be able to follow the logic of the program from the preceding discussion.  
A few points, however, are worth making:

- (1) We have set HIGHESTNUMBER to 100 on line 90, so that by simply changing this line the range of numbers used in the questions may be changed. This is better than using a fixed number 100 in lines 160 and 170. It is always better to use a variable name in such cases instead of a fixed constant. It makes the program easier to read and easier to change.
- (2) We choose only to count the right answers using the variable NUMBERRIGHT which is increased by 1 for each correct answer.
- (3) TOTALQUESTIONS is used to count the total number of questions asked.
- (4) NUMBERWRONG is used to count the number of wrong attempts at any individual question. If this number reaches 3 no more attempts are allowed, but the correct answer is given. Note that again the constant 3 is not used on line 360. The variable name ALLOWEDWRONG is used instead because it is more flexible as outlined above in note (1). ALLOWEDWRONG has been set to 3 on line 80. By changing this line the number of wrong answers allowed may easily be changed.

## Do it Yourself

In addition to the functions provided automatically, the so-called inbuilt functions, COMAL allows us to define our own functions. Let us study how this is done by means of a few simple examples.

```
0010 // PROGRAM 84
0020 //
0030 // COMELY KATE
0040 //
0050 // FUNCTION EXAMPLE(1)
0060 //
0070 DEF FNSUM(X)
0080   FNSUM:=X+12
0090 ENDDEF FNSUM
0100 //
0110 PRINT FNSUM(5)
0120 END
```

RUN

17

On lines 70 to 90 we define our very own brand new function. There are three stages in the definition.

- (1) the function heading which *always* starts with the word **DEF** (short for definition). Then follows the name of the function being defined which *always* starts with the letters **FN**. In this case the name is FNSUM. Finally in brackets comes the argument or arguments to be used by the function. This is similar to the argument or arguments used in the inbuilt functions, e.g. SIN(X) or RND(A,B).
- (2) the statements which actually define the function. In this case there is just one statement

FNSUM := X + 12

If there is more than one statement then at least one of them must assign a value to the variable which is the function name. In this case clearly the value X + 12 is assigned to the function name FNSUM.

- (3) the closing of the function definition which always consists of the word **ENDDEF** followed by the name of the function.

Once a function has been defined it may be used in exactly the same way as the pre-defined or inbuilt functions such as SIN, or COS, or RND, etc. are used. In the above program we simply print the value of the function applied to the argument 5, thus PRINT FNSUM (5). Using a function is often referred to as *calling* the function.

Remember these two important points:

- (1) a function is defined once.
- (2) a function which has been defined may be called many times.

## Another Example

```
0010 // PROGRAM 85
0020 //
0030 // COMELY KATE
0040 //
0050 // FUNCTION EXAMPLE(2)
0060 //
0070 DEF FNMYSTERY(A)
0080   FNMYSTERY:=A*A
0090 ENDEF FNMYSTERY
0100 //
0110 PRINT FNMYSTERY(12)
0120 END
```

What does this function do? What answer will be printed? Type in the program, run it and see.

The functions which COMAL automatically provides are called **pre-defined functions**. The functions we define ourselves are called **user-defined functions**. We can use the pre-defined functions in defining our own functions, as shown in the following program:

```
0010 // PROGRAM 86
0020 //
0030 // COMELY KATE
0040 //
0050 // FUNCTION EXAMPLE(3)
0060 //
0070 DEF FNTRIG(ANGLE)
0080   FNTRIG:=SIN(ANGLE)+COS(ANGLE)
0090 ENDEF FNTRIG
0100 //
0110 PRINT FNTRIG(1)
0120 END
```

RUN

1.381773

We can define more than one function in a program:

```
0010 // PROGRAM 87
0020 //
0030 // COMELY KATE
0040 //
0050 // TO SHOW MORE THAN ONE FUNCTION
0060 //
0070 DEF FNSQUARE(X)
0080   FNSQUARE:=X*X
0090 ENDEF FNSQUARE
0100 //
0110 DEF FNCUBE(Y)
0120   FNCUBE:=Y*Y*Y
0130 ENDEF FNCUBE
0140 //
0150 FOR NUMBER:=1 TO 10 DO
0160   PRINT NUMBER,"      ",FNSQUARE(NUMBER),"      ",FNCUBE(NUMBER)
0170 NEXT NUMBER
0180 END
```

RUN

1	1	1
2	4	8
3	9	27
4	16	64
5	25	125
6	36	216
7	49	343
8	64	512
9	81	729
10	100	1000

An important point to note is that we do not have to use the same identifier for the parameter of a function when calling the function as when defining it. Here we use NUMBER when calling and X or Y when defining. This is extremely useful. It means that when we are defining a function we can go ahead and use whatever variable name seems appropriate without having to worry about what parameter name will be used when the function is called. It means too that the function may be called at different times with different parameter names.

The parameter used in defining a function is often called a **dummy argument** or **formal parameter**. The parameter used in calling the function is often called an **actual parameter**.

Note that of course we do not have to use the same formal parameter name in defining each of our functions. We have used X in one above, Y in the other.

Another important point to note is that *a function does nothing until it is called*. Defining a function does not cause the action described to take place. The function is effectively dormant until it is called. If it is never called then its action never takes place.

We have seen that we can use pre-defined COMAL functions in defining our own functions. We can also use our own functions in defining other functions, as shown in the following example:

```
0010 // PROGRAM 88
0020 //
0030 // COMELY KATE
0040 //
0050 // FUNCTION USING FUNCTION
0060 //
0070 DEF FNSQUARE(X)
0080   FNSQUARE:=XXX
0090 ENDDDEF FNSQUARE
0100 //
0110 DEF FNF(NUM)
0120   FNF:=FNSQUARE(NUM)+FNSQUARE(NUM)
0130 ENDDDEF FNF
0140 //
0150 PRINT FNF(5)
0160 END
```

RUN

50

## Functions and Arithmetic

User-defined functions can be very useful in writing programs to solve simple problems in arithmetic.

**Example** Find the area of a circle of given radius.

Remember the formula for the area of a circle?  $A = \pi r^2$ .

```
0010 // PROGRAM 89
0020 //
0030 // COMELY KATE
0040 //
0050 // TO FIND THE AREA OF A CIRCLE
0060 //
0070 DEF FNAREA(RADIUS)
0080 PI:=22//7
0090 FNAREA:=PI*RADIUS*RADIUS
0100 ENDDEF FNAREA
0110 //
0120 PRINT
0130 INPUT "RADIUS OF CIRCLE ? ": R
0140 PRINT
0150 PRINT "AREA OF CIRCLE OF RADIUS ",R," = ",FNAREA(R)
0160 END
```

RUN

```
RADIUS OF CIRCLE ? 12.7
AREA OF CIRCLE OF RADIUS 12.7 = 506.9114
```

Here we have used the variable name RADIUS in defining the function and the variable name R in using the function.

**Example** Find the curved surface area and volume of a cylinder of given height  $h$  and base radius  $r$ .

The formulae required for the problem (in case you've forgotten!) are:

$$\text{Curved Surface Area} = 2\pi rh$$

$$\text{Volume} = \pi r^2 h$$

```
0010 // PROGRAM 90
0020 //
0030 // COMELY KATE
0040 //
0050 // TO FIND THE CURVED SURFACE AREA AND THE VOLUME OF A CYLINDER
0060 //
```

```

0070 DEF FNSURFACEAREA(R, H)
0080   GLOBAL PI
0090   FNSURFACEAREA:=2*PI*R*H
0100 ENDDEF FNSURFACEAREA
0110 //
0120 DEF FNVOLUME(R, H)
0130   GLOBAL PI
0140   FNVOLUME:=PI*R*R*H
0150 ENDDEF FNVOLUME
0160 //
0170 PI:=3.14
0180 PRINT
0190 PRINT "ENTER RADIUS AND HEIGHT OF CYLINDER"
0200 PRINT
0210 INPUT "RADIUS =  ": RADIUS .
0220 PRINT
0230 INPUT "HEIGHT =  ": HEIGHT
0240 PRINT
0250 PRINT
0260 PRINT "THE CURVED SURFACE AREA =  ",FNSURFACEAREA(RADIUS,HEIGHT)
0270 PRINT
0280 PRINT "THE VOLUME =  ",FNVOLUME(RADIUS,HEIGHT)
0290 END

```

RUN

ENTER RADIUS AND HEIGHT OF CYLINDER

RADIUS = 10

HEIGHT = 15

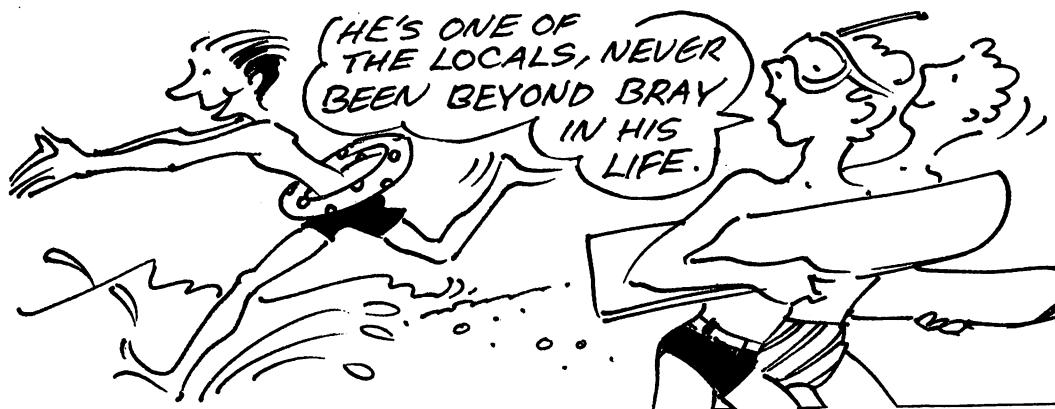
THE CURVED SURFACE AREA = 942

THE VOLUME = 4710

Two functions FNSURFACEAREA and FNVOLUME are defined to correspond to the two formulae above. The action of the program should be fairly clear. There are, however, two points of significance to be discussed:

- (1) For the first time in our own functions we have used more than one parameter in defining a function - thus FNSURFACEAREA (R,H) and FNVOLUME(R,H). When calling these functions we are of course obliged to use two parameters also. The parameters used when calling a function must always match those used in defining the function. Here, for example, the value of the actual parameter RADIUS on line 260 is passed to the formal parameter R in the function FNSURFACEAREA and the value of HEIGHT is passed to H. The first actual parameter is always matched with the first formal parameter, the second actual with the second formal, etc.
- (2) A new term **GLOBAL** has been used on lines 80 and 130. We must now discuss this term.

## LOCAL and GLOBAL variables



The parameters used in defining a function are purely local to the function. What does that mean? It means that their values are known and used only within the function itself and are not known and cannot be used outside the function. Take the function FNVOLUME above. The values of R and H are known only within FNVOLUME itself. If we included a line 285 PRINT R,H say, the computer would object when it came to this line and print the message

### UNDEFINED VARIABLE OR FUNCTION VALUE

pointing accusingly at line 285 as it did so. We could however print R and H *within* the function body.

Similarly the values of RADIUS and HEIGHT are known outside the function, but *not within the function itself*. If we tried for example to print them by including a line 125 PRINT RADIUS, HEIGHT the computer would give us the same error message as above and point accusingly at line 125.

'Hold on!' I hear you say. 'I thought you said that the values of RADIUS and HEIGHT were passed into the function to the variables R and H.' So they are, but within the function the program knows only of R and H and does not recognise RADIUS and HEIGHT. Effectively the function is CLOSED to all variables outside the function. This is the normal state of affairs. However, it can be modified by making use of GLOBAL.

Variables defined outside a function may be made known inside the function by declaring them to be GLOBAL inside the function. Thus in our previous program we need to use the value of PI, defined in the main program on line 170, inside both our functions. Therefore we include the declaration GLOBAL PI inside both functions (lines 80 and 130).

Similarly all variables defined in a main program (except the formal parameters) may be made available inside functions by declaring them to be GLOBAL. The actual parameters are known through their representatives, the formal parameters.

## Roots again

Find the roots of the equation  $ax^2 + bx + c = 0$ . No doubt you will remember that equations like this can be solved by means of the formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

It is worth reminding ourselves that we cannot simply ask the computer to solve a quadratic equation. We must give it precise instructions as to what to do and therefore we need a formula such as the above. The formula essentially provides us with an algorithm for solving all quadratic equations.

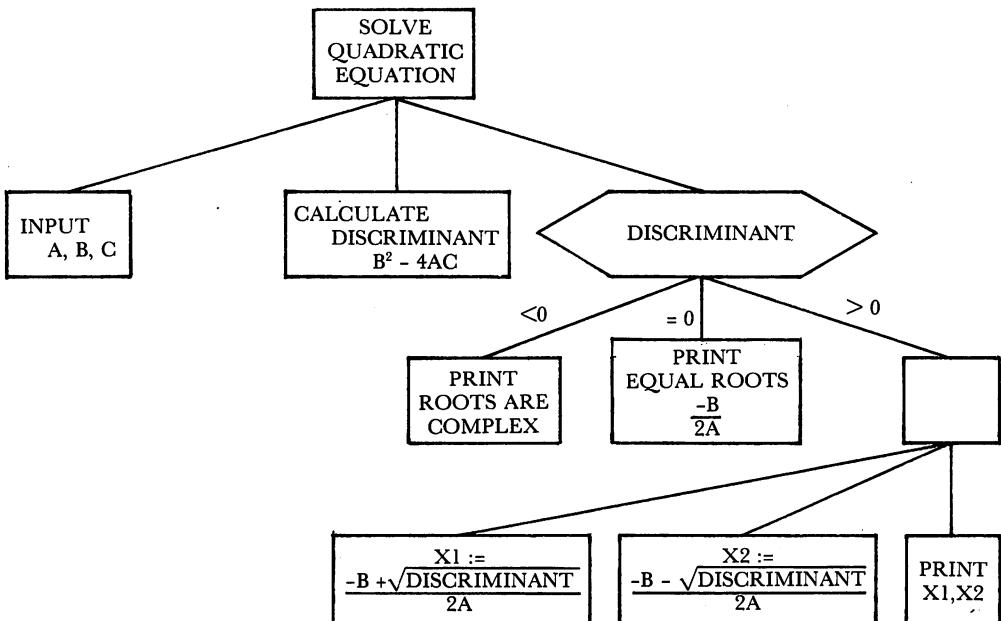
If we examine the formula, we notice that the expression

$$b^2 - 4ac$$

plays an important role. The expression dictates what kind of roots we get. Three cases can be distinguished:

- (1) If  $b^2 - 4ac > 0$  we get two distinct real roots for the equation.
- (2) If  $b^2 - 4ac = 0$  the two roots are the same.
- (3) If  $b^2 - 4ac < 0$  we get no real roots at all. The roots are said to be *complex* in this case.

Our program should make allowance for all three cases and be able to cope correctly with whichever case arises in a particular example.



One further point should be noted before we write the program. The important quantities in the expression  $ax^2 + bx + c$  are  $a$ ,  $b$  and  $c$ . The solution depends entirely on their values. Therefore these are the data values which must be given to the program. We choose to use INPUT to get them (line 120 below), so that the user may use the program to solve different equations quite easily.

```

0010 // PROGRAM 91
0020 //
0030 // COMELY KATE
0040 //
0050 // TO SOLVE A QUADRATIC EQUATION
0060 //
0070 DEF FNDISC(A, B, C)
0080   FNDISC:=(B*B-4*A*C)
0090 ENDEF FNDISC
0100 //
0110 PRINT
0120 INPUT "ENTER COEFFICIENTS OF EQUATION A,B,C  ": A, B, C
0130 PRINT
0140 DISCRIMINANT:=FNDISC(A,B,C)
0150 IF DISCRIMINANT<0 THEN
0160   PRINT "THE ROOTS ARE COMPLEX"
0170 ELIF DISCRIMINANT=0 THEN
0180   PRINT "THE ROOTS ARE EQUAL. THEY ARE BOTH = ",(-B)/(2*A)
0190 ELSE
0200   X1:=(-B+SQR(Discriminant))/(2*A)
0210   X2:=(-B-SQR(Discriminant))/(2*A)
0220   PRINT "THE ROOTS ARE ",X1," AND ",X2
0230 ENDIF
0240 END

RUN

ENTER COEFFICIENTS OF EQUATION A,B,C  1,-5,6
THE ROOTS ARE 3 AND 2

RUN

ENTER COEFFICIENTS OF EQUATION A,B,C  1,-6,9
THE ROOTS ARE EQUAL. THEY ARE BOTH = 3

RUN

ENTER COEFFICIENTS OF EQUATION A,B,C  1,3,9
THE ROOTS ARE COMPLEX

```

You should type in the program yourself and run it on several different equations, to make sure it does cope with all cases.

We have used a user-defined function to evaluate the important expression  $b^2 - 4ac$ .

## Recursion

We have seen that a function can call another function – see Program 86 where a user-defined function **FNTRIG** calls the system functions SIN and COS, and Program 88 where a user-defined function **FNF** calls another user-defined function **FNSQUARE**. The question now is – can a function call itself? And the answer is YES!

Consider the following simple program which calculates powers of 2.

```
0010 // PROGRAM 92
0020 //
0030 // COMELY KATE
0040 //
0050 // TO CALCULATE POWERS OF 2
0060 //
0070 DEF FNPOWER(N)
0080 IF N=0 THEN
0090   FNPOWER:=1
0100 ELSE
0110   FNPOWER:=2*FNPOWER(N-1)
0120 ENDIF
0130 ENDDDEF FNPOWER
0140 //
0150 PRINT
0160 INPUT "WHAT POWER OF 2 WOULD YOU LIKE CALCULATED ? "; P
0170 PRINT
0180 ANSWER:=FNPOWER(P)
0190 PRINT "2 TO THE POWER OF ",P," = ",ANSWER
0200 END
RUN
WHAT POWER OF 2 WOULD YOU LIKE CALCULATED ? 8
2 TO THE POWER OF 8 = 256
```

The important line here is line 110, where on the right hand side we apply the function **FNPOWER** to the argument N-1. But this line occurs *within the definition of the function FNPOWER* so that effectively **FNPOWER is calling itself**.

*A function which calls itself is said to  
be a recursive function.*

Let us examine carefully how recursion works by following the progress of the program as it struggles to calculate  $2^4$ , i.e assume that 4 is input for P on line 160.

Line 180 calls the function FNPOWER (4) and the following actions take place:

ANSWER

$$\begin{aligned} &= \underline{\underline{\text{FNPOWER (4)}}} \\ &= \underline{\underline{2 * \text{FNPOWER (3)}}} \dots \text{line 110} \\ &= 2 * \underline{\underline{2 * \text{FNPOWER (2)}}} \dots \text{line 110} \\ &\doteq 2 * 2 * \underline{\underline{2 * \text{FNPOWER (1)}}} \dots \text{line 110} \\ &= 2 * 2 * 2 * \underline{\underline{2 * \text{FNPOWER (0)}}} \dots \text{line 110} \\ &= 2 * 2 * 2 * \underline{\underline{2 * 1}} \dots \text{line 90} \\ &= 2 * 2 * 2 * \underline{\underline{2 * 2}} \\ &= 2 * 2 * \underline{\underline{4}} \\ &= 2 * \underline{\underline{8}} \\ &= 16 \end{aligned}$$

Note that each function call breaks down into a product of 2 and the answer to a function call on a smaller argument, until eventually we get down to FNPOWER(0). FNPOWER(0) is evaluated to 1 and ends the recursion. In effect, the function for each argument is obliged to wait until the result of the function for the next lower argument is available, except for FNPOWER(0) which is immediately assigned to 1. We have tried to show this waiting effect above by calculating the result at the end in small stages working from right to left.

Recursion consists of two fundamental components:

- (1) A base part which establishes a fixed point for the definition. In the above example the base part is  $2^0 = 1$ .
- (2) A recursive part where a function is defined in terms of a simpler version of itself. In the above example the recursive part is  $2^n = 2 * 2^{n-1}$  where  $2^{n-1}$  requires slightly less calculation than  $2^n$ .

## Fibonacci Numbers

As another example of recursion let us consider the famous Fibonacci sequence of numbers which is defined as follows:

$$\text{Base Part} \quad \begin{cases} f(0) = 1 \\ f(1) = 1 \end{cases}$$

$$\text{Recursive Part} \quad \begin{cases} f(n) = f(n - 1) + f(n - 2) \end{cases}$$

The value of the Fibonacci function for any argument  $n$  is calculated by adding together the values of the function for the arguments  $n - 1$  and  $n - 2$ , except for the arguments 0 and 1 where the value is 1 in each case.

To see how the definition works let us evaluate  $f(5)$ .

$$\begin{aligned} f(5) &= f(4) + f(3) \\ &= \underbrace{f(3)}_{f(2) + f(1)} + \underbrace{f(2)}_{f(1) + f(0)} + \underbrace{f(2)}_{f(1) + f(0)} + f(1) \\ &= \underbrace{f(1)}_{1} + \underbrace{f(0)}_{1} + f(1) + f(1) + f(0) + f(1) + f(1) + f(0) + f(1) \\ &= 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 \\ &= 8 \end{aligned}$$

A program to calculate Fibonacci values is very simple.

```
0010 // PROGRAM 93
0020 //
0030 // COMELY KATE
0040 //
0050 // TO CALCULATE FIBONACCI NUMBERS
0060 //
0070 DEF FNFn(N)
0080 IF N=0 OR N=1 THEN
0090   FNFn:=1
0100 ELSE
0110   FNFn:=FNFn(N-1)+FNFn(N-2)
0120 ENDIF
0130 ENDDFN FNFn
0140 //
0150 PRINT
0160 INPUT "WHAT FIBONACCI NUMBER WOULD YOU LIKE ? ": NUMBER
0170 PRINT
0180 PRINT "FIBONNACI()",NUMBER," = ",FNFn(NUMBER)
0190 END
```

RUN

WHAT FIBONACCI NUMBER WOULD YOU LIKE ? 10

FIBONNACI(10) = 89

RUN

WHAT FIBONACCI NUMBER WOULD YOU LIKE ? 16

FIBONNACI(16) = 1597

Although our recursive function to calculate Fibonacci numbers is simple and straightforward it is not the best way to do the job. It is much slower than a program which uses a loop to keep adding successive terms of the sequence. Recursive programs often tend to be slower than iterative programs which perform the same job. However, recursion has a certain elegance and expresses in a very clear way the technique of breaking a problem down into simpler problems. If recursion is clearer than iteration for a given problem then it is generally better to use it. Remember that clarity and simplicity are primary virtues in constructing programs.

## Summary

COMAL provides a range of pre-defined functions.

A function accepts arguments or parameters and produces a result.

In addition to system functions a user may define his own functions.

A function is defined through the format

Function Heading

Function Body

Function End

A function is called by using the function name applied to actual parameters.

Formal parameters occur in a function definition; actual parameters occur in a function call.

A function is normally closed to the environment outside the function. All variables used within a function definition are local to the function unless specified as GLOBAL.

Recursion allows a function to be defined in terms of itself.

Recursive functions are often simple, clear and elegant.

### COMAL KEYWORDS

ABS	AND	AUTO	CASE	CAT	CLEAR	CON
COS	DATA	DEF	DEL	DELETE	DIM	DIV
DO	EDIT	ELIF	END	ENDCASE	ENDDEF	
ENDIF	ENDWHILE	EOD	EXEC	EXP	FN	
FOR	GLOBAL	IF...THEN...ELSE		INPUT	INT	
LEN	LIST	LOAD	MOD	NEW	NEXT	
NOT	OR	OTHERWISE	PRINT	READ	RENUM	
REPEAT	RND	ROUND	RUN	SAVE	SIN	
STEP	STOP	TAB	UNTIL	WHEN	WHILE	

### QUESTIONS and EXERCISES

1. What is
  - (a) a pre-defined function?
  - (b) a user-defined function?
  - (c) an argument or parameter?
  - (d) a formal parameter?
  - (e) an actual parameter?
2. Why are functions useful? When should a recursive function be used?
3. Explain what each of the following functions does:  
TAB            ROUND            SIN            COS            RND
4. Rewrite Program 71 using loops to get the top and bottom lines of asterisks printed.
5. Change Program 71 to make the square (a) smaller, (b) bigger. See if you can make it alternate between small and big. See if you can make the size vary randomly. (Hint: use CLEAR & RND.)

6. Using the TAB function, draw,  
 (i) an upside-down triangle,  
 (ii) this shape



(iii) the letter Z to fill the screen.

7. Draw graphs of the following:

- (1)  $\cos(x)$
- (2)  $\sin(x) + 2 \cos(x)$
- (3)  $\tan(x)$
- (4)  $\exp(x)$
- (5)  $\log(x)$
- (6)  $\text{ABS}(\sin(x))$
- (7)  $\sin^2(x) + \cos^2(x)$
- (8)  $A * \sin(x) + B * \cos(x)$   
for given values of A and B

8. Write a program to print random integers between 1 and 52 and to associate such values with the cards of a standard pack of playing cards.

9. Invent a game using dice and write a program to play the game.

10. Write function definitions for each of the following:

- (i)  $x^2 + x^3$
- (ii)  $ax^3 + bx^2 + cx + d$
- (iii)  $1/x + 1/x^2 + 1/x^3$
- (iv)  $\sin(ax + c)$
- (v)  $x^2 + y^2$
- (vi)  $a \sin x + b \cos x$

11. What happens in Program 91 if  $A = 0$ ? Change the program to take account of this. Also allow the program to continue automatically processing more equations until requested to stop.

12. Write a program to draw random patterns on the screen.

13. You probably noticed with Program 81 that if the dice had to be rolled again several times, the messages flashed up along the screen rather too quickly. Can you think of a way of slowing it down? Modify the program to include your idea.

14. Write recursive functions to

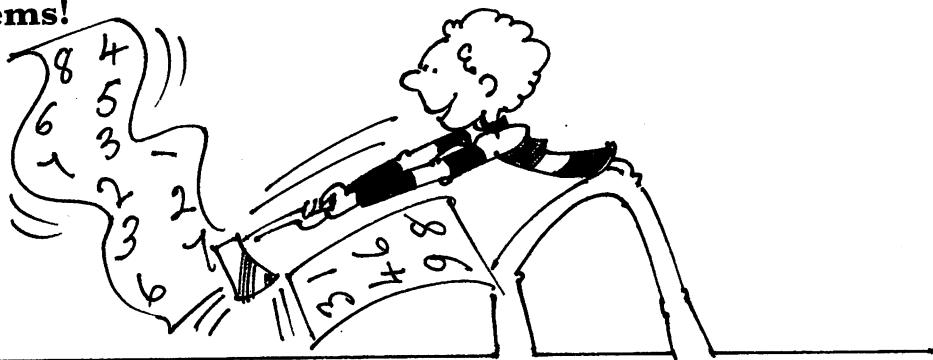
- (a) evaluate  $3^n$  for given  $n$
- (b) add two positive integers
- (c) multiply two positive integers
- (d) evaluate  $n!$  for given  $n$

15. Write programs to draw graphs of

- (a) a radio-active decay process
- (b) a bouncing ball.

# 13 Procedures

## Problems!



The art of problem solving lies in breaking problems down into smaller problems and if necessary breaking these into still smaller problems and so on until eventually the problems are small enough to be manageable. This applies whether the problem we start with is as large as producing a government budget or as small as adding some numbers. Leaving aside the trivial (!) problem of the country's finances, consider for a moment the task of adding

$$\begin{array}{r} 5763 \\ 4298 \\ + 8165 \\ \hline \end{array}$$

We do not try to add the three numbers in one gulp. We start with just the right-most digits. Even here we don't add all three digits at once. We just add 5 and 8 getting 13 and then add 3 to get 16. We then write down 6 and carry 1 to the next place.

The important point is that we reduce our addition problem to the much easier task of adding just two digits. This is manageable because we have learned our addition tables, i.e. we have learned how to add just two digits at a time.

The technique of reducing problems to simpler problems is so universal that we may state for ourselves the blunt rule -

*Never try to solve a large problem.*

Since programs represent solutions to problems, what we have said above applies to the construction of programs also. Of course we are already aware of this, because our structure diagrams constitute a method of resolving a problem solution into simpler and simpler stages.

The solution to a problem, then, often consists of a combination of solutions to the sub-problems into which the original problem may be decomposed. So too a program should consist of a combination of *sub-programs* which express the different stages of the

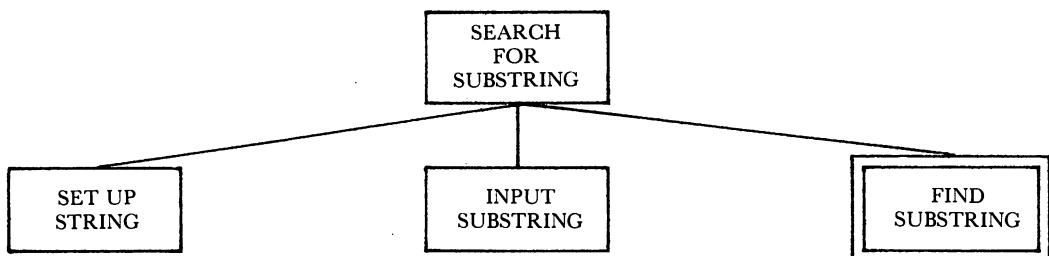
solution. Functions are one method of expressing these sub-programs. In this chapter we turn to another similar method, namely the construction and use of procedures.

### *Procedures are programs*

They are often called sub-programs because they form part of a larger program. Let us see what is involved by considering a very simple example.

## **Stringing along**

Given a string find if a certain substring occurs within it, e.g. the substring CAT occurs within the string THE CAT SAT CONTENTEDLY ON THE MAT, whereas the substring FAT does not occur.



```
0080 STRING$!=="THE CAT SAT CONTENTEDLY ON THE MAT"  
0090 INPUT "WHAT PATTERN DO YOU WISH TO SEARCH FOR ? "; SUBSTRING$  
0100 EXEC FINDSUBSTRING
```

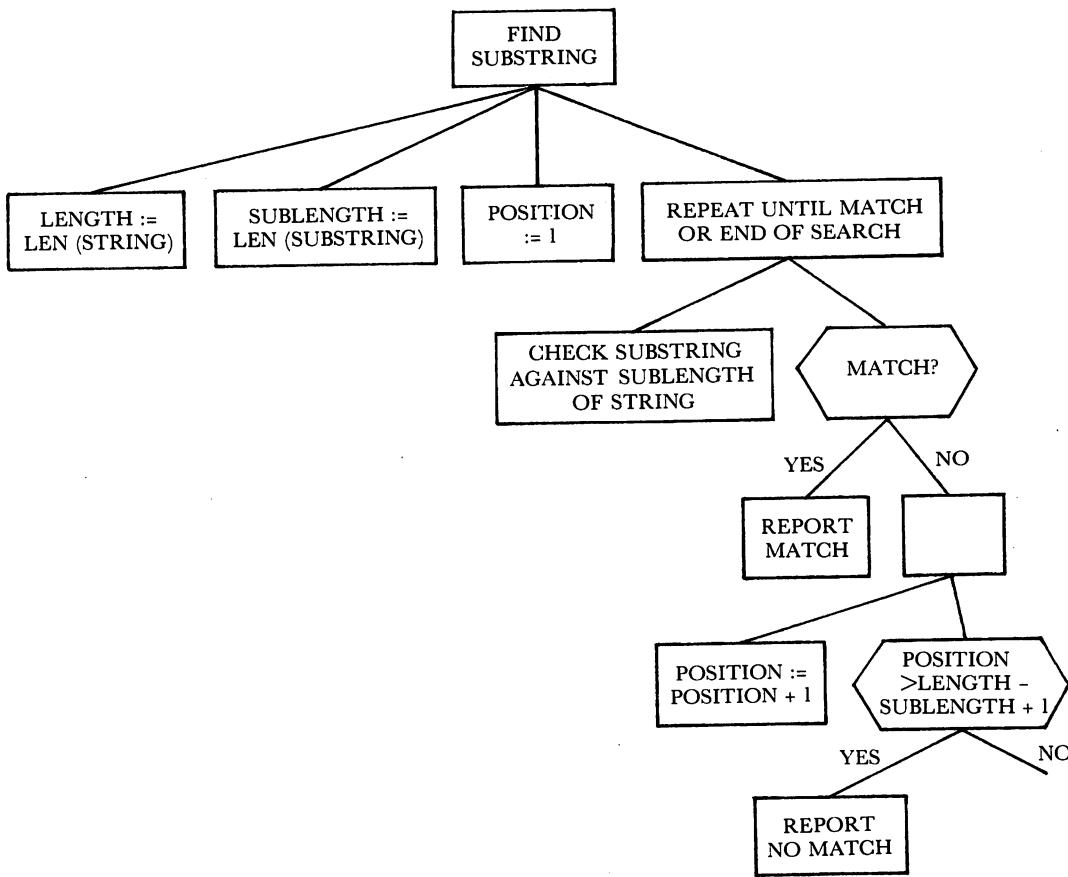
Our program follows directly from the first level of refinement of our structure diagram and apart from the usual opening comments and the dimension declarations, consists of just three statements:

- Line 80      sets up the string in which searching will take place.
- Line 90      prompts for the substring to be searched for.
- Line 100     calls for a sub-program named FINDSUBSTRING to be executed  
(EXEC is short for EXECUTE)

We have surrounded the task FIND SUBSTRING by a double rectangle in our diagram to indicate that further consideration will have to be given to it. Similarly our program is not complete yet and would not in fact work as it stands. We need to spell out the instructions of the sub-program FINDSUBSTRING before it can be executed.

So what have we gained?

Well, we have indicated in broad outline what the main tasks are. We can concentrate in peace on the details of each task now in the true spirit of problem solving, by gradually reducing the problem to smaller more manageable components. Of course two of the tasks (expressed by lines 80 and 90) need no further refinement. Therefore we need only to attend to the procedure for finding a substring.



The method is fairly straightforward. Starting in position 1 of the string we keep matching the substring against a piece of string of the same length until either we have found an exact match or we have gone too near the end of the string for there to be a match,

e.g. suppose we were looking for GOOD in the string MY GOOD FRIEND

	MY	GOOD	F R I E N D
1st Try	GOOD		
2nd Try	GOOD		
3rd Try	GOOD		
4th Try	GOOD		

Match!

If we were looking for COMPANIONS the following would be the sequence.

MY    GOOD    F R I E N D	<u>  </u> : : : : : : : 	
1st Try	COMPANIONS	
2nd Try	COMPANIONS	
3rd Try	COMPANIONS	
4th Try	COMPANIONS	
5th Try	COMPANIONS	
6th Try	would bring COMPANIONS too far to the right for any match to be possible. Note that POSITION (of start of COMPANIONS) would be 6 which is greater than LENGTH - SUBLLENGTH + 1 (= 14 - 10 + 1).	

Let us now write a program to express this substring search.

```
0130 PROC FINDSUBSTRING
0140   LENGTH:=LEN(STRING$)
0150   SUBLLENGTH:=LEN(SUBSTRING$)
0160   POSITION:=1
0170   MATCH:=FALSE
0180   REPEAT
0190     IF SUBSTRING$=STRING$(POSITION,POSITION+SUBLLENGTH-1) THEN
0200       MATCH:=TRUE
0210       PRINT SUBSTRING$," FOUND IN POSITION ",POSITION
0220     ELSE
0230       POSITION:=POSITION+1
0240       IF POSITION>LENGTH-SUBLLENGTH+1 THEN PRINT "NO MATCH"
0250     ENDIF
0260   UNTIL MATCH OR POSITION>LENGTH-SUBLLENGTH+1
0270 ENDPROC FINDSUBSTRING
```

The only points which may not be clear at first sight are

- (1) why POSITION + SUBLLENGTH - 1 is used on line 190 instead of just POSITION + SUBLLENGTH.
- (2) Why LENGTH - SUBLLENGTH + 1 is used instead of LENGTH - SUBLLENGTH on line 240.

We leave you, dear reader, to work out the reasons in this case for yourself!

Now how do we put these last two programs together to form one complete task. The answer is by including the second program as a procedure in the first program as follows:

```
0010 // PROGRAM 94
0020 //
0030 // COMELY KATE
0040 //
0050 // TO FIND A SUBSTRING OF A STRING
0060 //
0070 DIM STRING$ OF 50, SUBSTRING$ OF 50
0080 STRING$!=="THE CAT SAT CONTENTEDLY ON THE MAT"
0090 INPUT "WHAT PATTERN DO YOU WISH TO SEARCH FOR ? "; SUBSTRING$
0100 EXEC FINDSUBSTRING
```

```

0110 END
0120 //
0130 PROC FINDSUBSTRING
0140   LENGTH:=LEN(STRING$)
0150   SUBLLENGTH:=LEN(SUBSTRING$)
0160   POSITION:=1
0170   MATCH:=FALSE
0180   REPEAT
0190     IF SUBSTRING$=STRING$(POSITION,POSITION+SUBLLENGTH-1) THEN
0200       MATCH:=TRUE
0210       PRINT SUBSTRING$," FOUND IN POSITION ",POSITION
0220     ELSE
0230       POSITION:=POSITION+1
0240       IF POSITION>LENGTH-SUBLLENGTH+1 THEN PRINT "NO MATCH"
0250     ENDIF
0260   UNTIL MATCH OR POSITION>LENGTH-SUBLLENGTH+1
0270 ENDPROC FINDSUBSTRING

```

RUN

WHAT PATTERN DO YOU WISH TO SEARCH FOR ? CAT  
 CAT FOUND IN POSITION 5

RUN

WHAT PATTERN DO YOU WISH TO SEARCH FOR ? KITTEN  
 NO MATCH

The important points to note are:

- (1) The procedure begins with a heading

### PROC FINDSUBSTRING

The word **PROC** (short for PROCEDURE) must always be used in a procedure heading. FINDSUBSTRING is the name of the procedure and is of our own choosing. We can use any name we wish here.

- (2) When we call the procedure using the instruction EXEC we must use exactly the same name as when we define the procedure, thus EXEC FINDSUBSTRING. In some COMAL systems EXEC may be omitted. Line 100 will then read simply FINDSUBSTRING.
- (3) The procedure ends with the closing line ENDPROC FINDSUBSTRING. Again **ENDPROC** must always be used and FINDSUBSTRING must match the name in the procedure heading.
- (4) The statements between PROC and ENDPROC define the meaning and action of the procedure. Note that as with functions there are two important stages in the use of procedures:
  - (i) Procedure definition (lines 130 to 270 above)
  - (ii) Procedure call (line 100).

- (5) The definition of the procedure may occur after the END of the main program. In fact the definition could occur anywhere in the text of the program since the procedure is not activated until it is called. Thus, no matter where the statements were placed, they would not be executed until an EXEC statement is encountered.

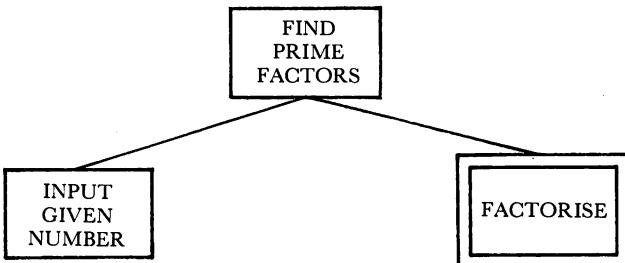
Finally note that the main program is very short whereas the procedure is quite long.

## **Figuring the Factors**

As another illustration of the simple use of procedures let us write a program to work out the prime factors of a given number. You remember that factors of a number are numbers which divide exactly into the given number and that prime factors are those factors which cannot be further factorised, e.g. 6 is a factor of 12, but it is not a prime factor.

The prime factors of 12 are 2, 2 and 3.

We write down 2 twice because it appears twice in the full factorisation of 12, i.e  $12 = 2 \times 2 \times 3$ .

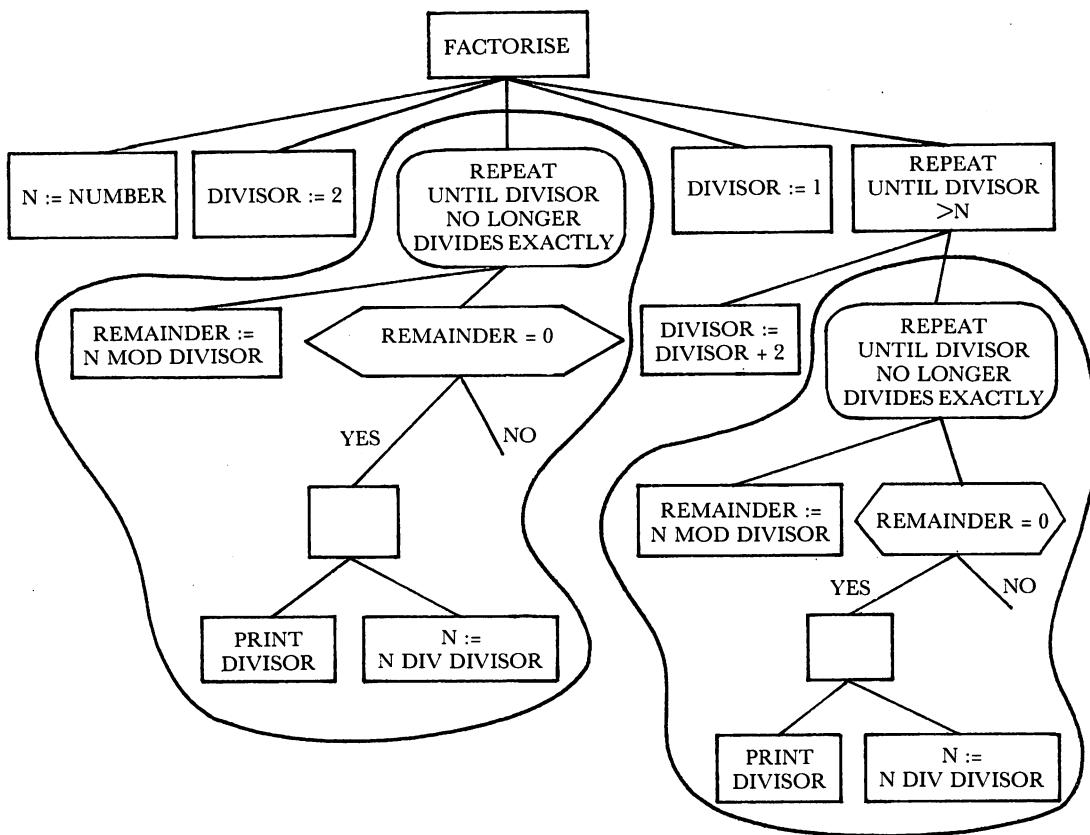


We leave the elaboration of FACTORISE for the moment and write our main program as follows (omitting the opening remarks):

```

70 INPUT "GIVEN NUMBER": NUMBER
80 EXEC FACTORISE
90 END
  
```

Now we proceed to determine the instructions of the procedure FACTORISE. To find the factors of a number we will start with 2 and see if it divides in exactly. If it does we will output 2 as one of our factors and reduce the original number by dividing it by 2. Then we will try 2 again. Following 2 we will try 3 and then 5 and so on if necessary until we have all the factors. You will notice that this is very like the way we determined whether a number was prime in Program 54. In fact the only real differences are that here we keep going when we find a factor, in order to see if there are more and we keep a list of the factors that we find by printing them out.



We must allow for the fact that the same factor may occur several times. This is done by repeatedly trying to divide by the same number until it no longer divides exactly – see the encircled sections of the diagram.

e.g.  $360 = 2 \times 2 \times 2 \times 3 \times 3 \times 5$ .

Therefore we must divide by 2 three times and by 3 twice.

Since 2 is the only even prime factor we treat it as a special case before going on to work through the odd numbers. This means that the above solution shows two parts which are exactly the same. You may not like this method and you are cordially invited to try a different approach. While you are at it you should notice that simply moving through all the odd numbers by starting with a divisor of 3 and then increasing it by 2 each time is a bit of a waste since many of the odd numbers are not prime, e.g. 9. Of course 9 will already have been taken care of by dividing by 3 twice. Again you are invited to do better. I will simply quote the ancient impatient phrase '*quod scripsi scripsi*' and pass on to the full program for factorising! The actual factorising part is included as a procedure.

```

0010 // PROGRAM 95
0020 //
0030 // COMELY KATE
0040 //
0050 // TO FACTORISE A NUMBER
0060 //
0070 INPUT "GIVEN NUMBER ? ": NUMBER
0080 EXEC FACTORISE
0090 END
0100 //
0110 PROC FACTORISE
0120 N:=NUMBER
0130 DIVISOR:=2
0140 PRINT "THE PRIME FACTORS OF ",N," ARE"
0150 PRINT
0160 REPEAT
0170   REMAINDER:=N MOD DIVISOR
0180   IF REMAINDER=0 THEN
0190     PRINT DIVISOR;
0200     N:=N DIV DIVISOR
0210   ENDIF
0220 UNTIL REMAINDER<>0
0230 DIVISOR:=1
0240 REPEAT
0250   DIVISOR:=DIVISOR+2
0260   IF DIVISOR<=N THEN
0270     REPEAT
0280       REMAINDER:=N MOD DIVISOR
0290       IF REMAINDER=0 THEN
0300         PRINT DIVISOR;
0310         N:=N DIV DIVISOR
0320       ENDIF
0330     UNTIL REMAINDER<>0
0340   ENDIF
0350 UNTIL DIVISOR>N
0360 ENDPROC FACTORISE

```

RUN

GIVEN NUMBER ? 17  
 THE PRIME FACTORS OF 17 ARE

17

RUN

GIVEN NUMBER ? 360  
 THE PRIME FACTORS OF 360 ARE

2 2 2 3 3 5

Notice once again that the main program is very short and that all the work takes place within the procedure.

## Parameters

As we have already stressed, procedures are in fact programs and could in many cases stand as full programs in their own right independently of any other program. However, the procedures we have displayed in our last two examples could not quite

do this - they both needed information from the main program in order to carry out their work. Consider FACTORISE for example. It needed to know the NUMBER it was being asked to factorise. Thus in order for line 120 to have any meaning NUMBER had to have a value. Similarly the procedure FINDSUBSTRING needed to know the values of STRING\$ and SUBSTRING\$ in order to do its work in Program 94.

Because a calling program must often pass information to a procedure, it is convenient to establish an explicit mechanism for passing this information. This is done by making use of parameters.

We have already seen parameters at work with functions. Consider for example SIN(X). X is a parameter. It tells the SIN function what angle it must calculate the sine of. Similarly RND(1,6) uses two parameters, 1 and 6, to specify the range of random numbers to be computed by the function.

Parameters may be used with procedures in the same way as they were used with functions, i.e. to pass information from the main program to the procedure. Such parameters are called **value parameters**. Let us illustrate by rewriting Program 95 using a value parameter for our procedure FACTORISE:

```
0010 // PROGRAM 96
0020 //
0030 // COMELY KATE
0040 //
0050 // TO FACTORISE A NUMBER
0060 //
0070 INPUT "GIVEN NUMBER ? ": NUMBER
0080 EXEC FACTORISE(NUMBER)
0090 END
0100 //
0110 PROC FACTORISE(N)
0120   DIVISOR:=2
0130   PRINT "THE PRIME FACTORS OF ",N," ARE"
0140   PRINT
0150   REPEAT
0160     REMAINDER:=N MOD DIVISOR
0170     IF REMAINDER=0 THEN
0180       PRINT DIVISOR;
0190       N:=N DIV DIVISOR
0200     ENDIF
0210   UNTIL REMAINDER<>0
0220   DIVISOR:=1
0230   REPEAT
0240     DIVISOR:=DIVISOR+2
0250     IF DIVISOR<=N THEN
0260       REPEAT
0270         REMAINDER:=N MOD DIVISOR
0280         IF REMAINDER=0 THEN
0290           PRINT DIVISOR;
0300           N:=N DIV DIVISOR
0310         ENDIF
0320       UNTIL REMAINDER<>0
0330     ENDIF
0340   UNTIL DIVISOR>N
0350 ENDPROC FACTORISE
```

```

RUN
GIVEN NUMBER ? 17
THE PRIME FACTORS OF 17 ARE
17

RUN
GIVEN NUMBER ? 360
THE PRIME FACTORS OF 360 ARE
2 2 2 3 3 5

```

Program 96 is almost exactly the same as Program 95. Notice, however, that we pass the value of NUMBER to the procedure FACTORISE by using EXEC FACTORISE (NUMBER) in line 80. Note too that the name of the parameter in the calling statement (line 80) does not have to be the same as the name of the parameter in the procedure heading. This has the advantage that the procedure can be written completely independently of whatever program may call it without worrying about what names will be used for the various values.

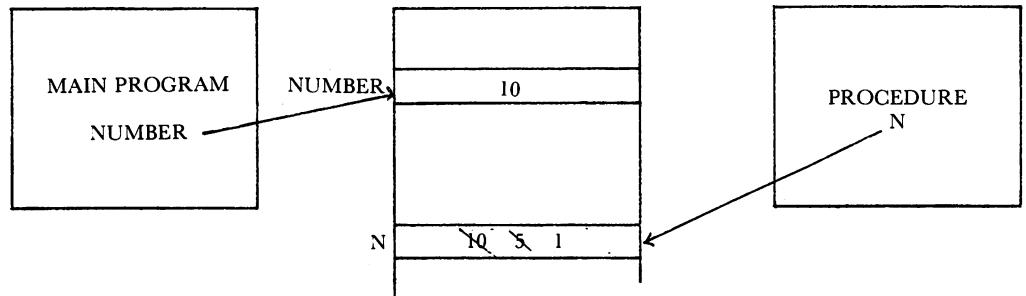
To be clear on the role that a value parameter plays, let us consider what happens when NUMBER has the value 10 and the statement EXEC FACTORISE (NUMBER) is executed.

The value 10 is passed to N because N corresponds to NUMBER and takes its place in the procedure. Note that we do not have the statement N := NUMBER in Program 96 (we had it in Program 95), since N automatically receives its value when the procedure is called. Thus N starts with a value of 10. N then changes to 5 when it is divided by 2, and finally to 1 when it is divided by 5. So when the procedure finishes, N has a value of 1. *But NUMBER still retains its value of 10.*

Just as with functions, the parameter in the calling statement is known as an **actual parameter** since it contains the actual value to be used by the procedure which is being called.

The parameter in the procedure heading is known as a **formal parameter** since it serves as a dummy variable without a value until the procedure is called.

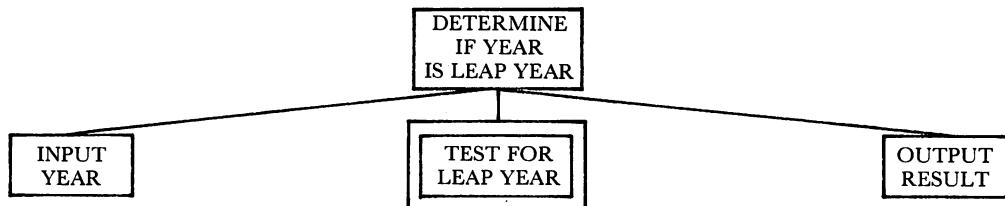
When the procedure is called, a copy of the actual parameter's value is placed in the formal parameter variable and it is this copy which is manipulated during the course of the procedure. Since it is the copy which is used within the procedure, the actual value back in the main program is protected from unwanted alteration.



## Second Call

Has this question occurred to you - can a procedure call another procedure? The answer is YES! An example will show how.

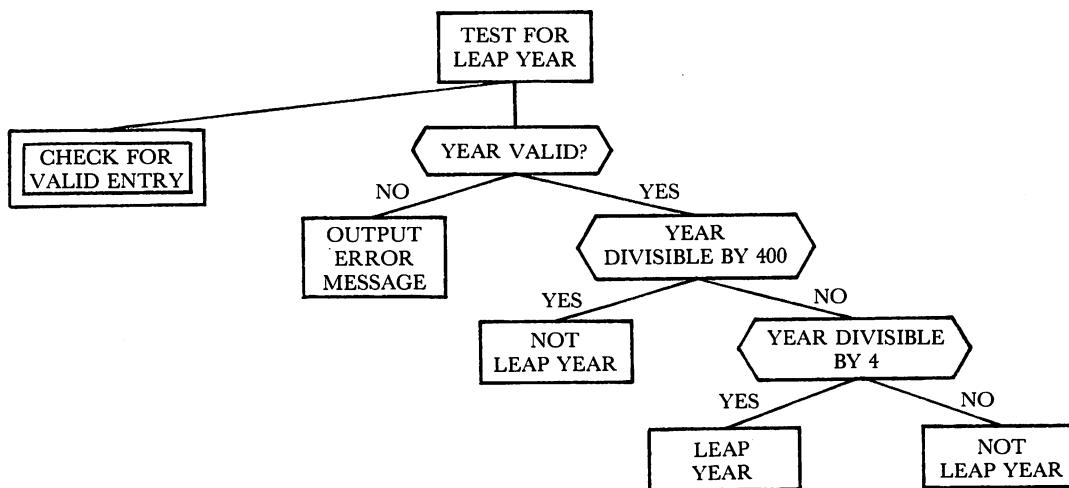
**Problem** Determine if a given year is a leap year.



gives the program

```
80 INPUT "ENTER YEAR": YEAR$  
90 EXEC TESTLEAPYEAR  
100 END
```

Let us now spell out the details of the TEST FOR LEAP YEAR:



The first box on level 1 of our diagram indicates a need for further elaboration. The idea is that before we proceed to test the actual year to see if it really is a leap year we decide to make sure the user has not made a mistake when entering the digits of the year. For example he might have entered 19.40 or I940 or some other sequence of symbols which would not be taken as an integer.

No doubt you are thinking ‘isn’t it about time we decided to check for invalid input?’ Wasn’t there many a program where some rubbish could have been entered to throw the program into complete disarray? You are perfectly right. The only excuse I can offer is that I didn’t want to complicate things too much as we went along. You will of course now go back and improve each program by including suitable checks!

Well now that we have decided to check the input how do we do it? It is no good saying something like

```
INPUT YEAR#
IF YEAR# <> INTEGER THEN etc.
```

Firstly the poor dumb computer does not know what the word INTEGER means. Secondly and even more critically it would kick up a fuss immediately after the INPUT statement was executed if we tried to enter something that wasn’t an integer. It would probably print an error message – ERROR IN NUMBER, or TOO MUCH INPUT, or something similar.

So what can we do?

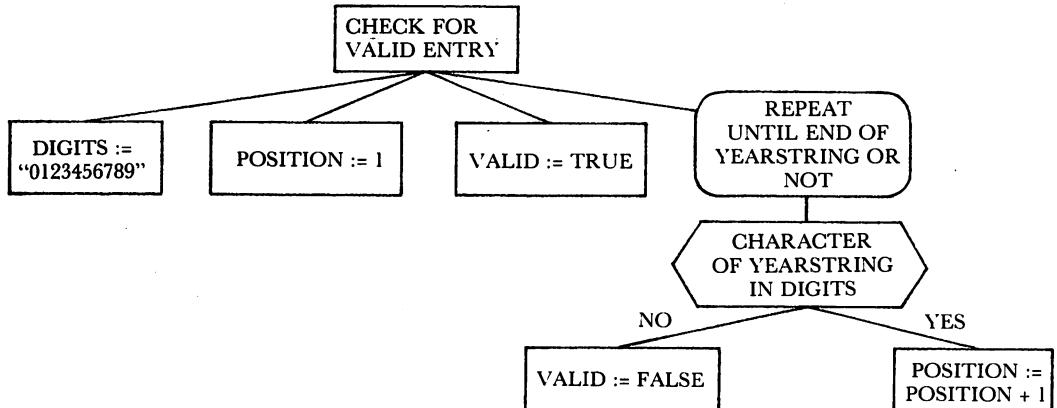
The answer is that we will have to take in the integer representing YEAR in the form of a string. A string is the most general form of input since it can contain any symbol. When the program has accepted the string it can then check to see if each character is in fact a digit. If it isn’t, an error is reported. If it is, the program can go ahead and test to see if the number formed by the digits is a leap year. To do this of course the string must be converted to a number. Luckily there is a handy COMAL function to do this for us, namely **VAL**

e.g.

$\text{VAL} ("1940") = 1940$

↑                                   ↑  
This is a string.                 This is a number.

Let us present the checking process in the form of a structure diagram.



We have used a cunning trick to check each character in our yearstring. Rather than asking if the character is 0, then if it is 1, then if it is 2, etc., we collect all the digits together in one string called DIGITS and then simply ask if each character of yearstring is in the string DIGITS. Again COMAL has a handy function to allow us to do this directly, as we see in the program below:

```
0010 // PROGRAM 97
0020 //
0030 // COMELY KATE
0040 //
0050 // TO TEST FOR LEAP YEAR
0060 //
0070 DIM YEAR$ OF 10
0080 PRINT
0090 INPUT "ENTER YEAR ": YEAR$
0100 EXEC TESTLEAPYEAR
0110 END
0120 //
0130 PROC TESTLEAPYEAR
0140   EXEC CHECKVALIDENTRY
0150   PRINT
0160   IF NOT VALID THEN
0170     PRINT "INVALID ENTRY"
0180   ELSE
0190     YEARVAL:=VAL(YEAR$)
0200     IF YEARVAL MOD 400=0 OR YEARVAL MOD 4<>0 THEN
0210       PRINT YEARVAL," IS NOT A LEAP YEAR"
0220     ELSE
0230       PRINT YEARVAL," IS A LEAP YEAR"
0240     ENDIF
0250   ENDIF
0260 ENDPROC TESTLEAPYEAR
0270 //
0280 PROC CHECKVALIDENTRY
0290   DIM DIGITS$ OF 10
0300   DIGITS$:="0123456789"
0310   POSITION:=1
0320   VALID:=TRUE
0330   REPEAT
0340     IF YEAR$(POSITION) IN DIGITS$ THEN
0350       POSITION:=POSITION+1
0360     ELSE
0370       VALID:=FALSE
0380     ENDIF
0390   UNTIL POSITION>LEN(YEAR$) OR NOT VALID
0400 ENDPROC CHECKVALIDENTRY

RUN
ENTER YEAR 1980
1980 IS A LEAP YEAR

RUN
ENTER YEAR 2000
2000 IS NOT A LEAP YEAR

RUN
ENTER YEAR 1983
1983 IS NOT A LEAP YEAR
```

The first point to note is that the procedure CHECKVALIDENTRY is called from within the procedure TESTLEAPYEAR. This is a perfectly normal and acceptable thing to do. Any procedure can call another procedure if it requires.

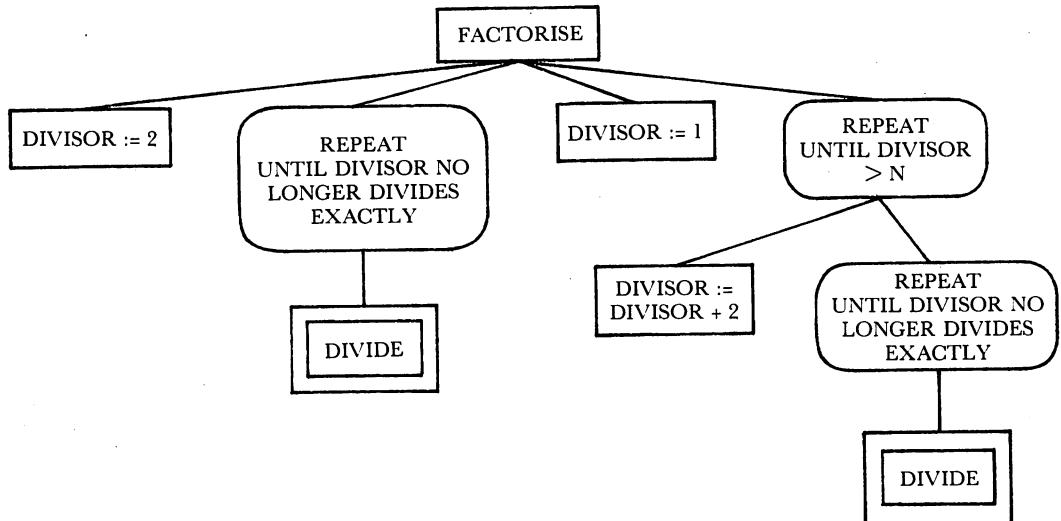
The second thing to examine is line 340. The COMAL function IN has been used to check if the single character YEAR\$ (POSITION) occurs in the string DIGITS\$. The function IN checks to see if a string is a substring of another string,

e.g.      "CAT" IN "THE FAT CAT" returns TRUE  
              "MAT" IN "THE FAT CAT" returns FALSE

Do I hear you saying (perhaps a little peeishly) - why didn't you tell us this before inflicting Program 94 on us? In fact why did you bother writing it at all when it would have been far handier to use the function IN? Ah well, haven't we learned something? Don't we know now how the function IN might work?

## Tidying up

We have already hinted that it seems a bit untidy to have two identical but separate portions of an algorithm in connection with our structure diagram on page 203. Let us tidy this up and at the same time show that just as a calling program needs to pass information to a procedure, so does a procedure often need to pass information back to a calling program.



We have already elaborated DIVIDE since it is intended to be the same as the encircled portions of the diagram on page 203.

```

0010 // PROGRAM 98
0020 //
0030 // COMELY KATE
0040 //
0050 // TO FACTORISE A NUMBER
0060 //
0070 INPUT "GIVEN NUMBER ? ": NUMBER
0080 EXEC FACTORISE(NUMBER)
0090 END
0100 //
0110 PROC FACTORISE(N)
0120 DIVISOR:=2
0130 REMAINDER:=0
0140 PRINT "THE PRIME FACTORS OF ",N," ARE"
0150 PRINT
0160 REPEAT
0170 EXEC DIVIDE(2,N,REMAINDER)
0180 UNTIL REMAINDER<>0
0190 DIVISOR:=1
0200 REPEAT
0210 DIVISOR:=DIVISOR+2
0220 IF DIVISOR<=N THEN
0230 REPEAT
0240 EXEC DIVIDE(DIVISOR,N,REMAINDER)
0250 UNTIL REMAINDER<>0
0260 ENDIF
0270 UNTIL DIVISOR>N
0280 ENDPROC FACTORISE
0290 //
0300 PROC DIVIDE(A, REF B, REF R)
0310 R:=B MOD A
0320 IF R=0 THEN
0330 PRINT A;
0340 B:=B DIV A
0350 ENDIF
0360 ENDPROC DIVIDE

```

RUN

```

GIVEN NUMBER ? 17
THE PRIME FACTORS OF 17 ARE

```

17

RUN

```

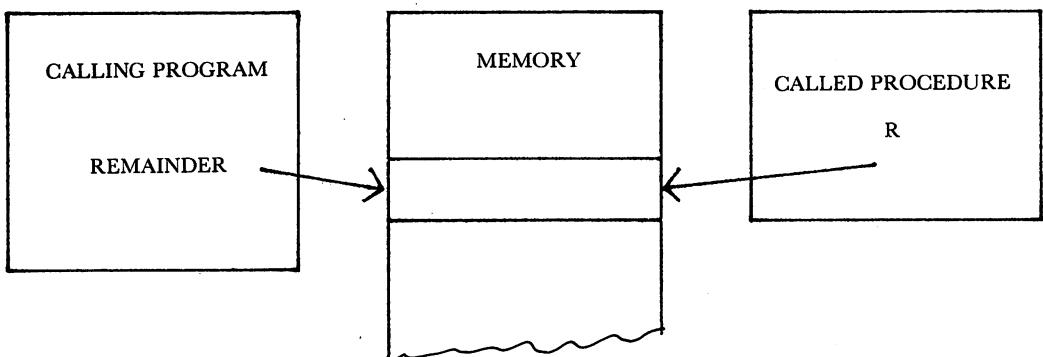
GIVEN NUMBER ? 360
THE PRIME FACTORS OF 360 ARE

```

2 2 2 3 3 5

Note very carefully that we have preceded two of the formal parameters in the procedure heading on line 300 by the word **REF** (short for REFERENCE). This informs the COMAL system that these two parameters, B and R, are to be used as alternative names to refer to the actual parameters N and REMAINDER respectively. In this case no copies of the actual parameter values will be made. B will refer to the same location in memory as N does and R will refer to the same location as

**REMAINDER**. This means that, for example, whatever value R acquires during the execution of the procedure DIVIDE will be automatically passed back to REMAINDER in the calling program.



## Value and Reference Parameters

We can now distinguish between two types of parameters, **call by value parameters** and **call by reference parameters**.

Value parameters are used to accept values from a calling program for use by a procedure. The called procedure makes a copy of the value passed and the parameter within the procedure is purely local to the procedure and independent of its counterpart in the calling program. The actual parameter in the calling program is not affected in any way by changes that may occur to the corresponding formal parameter within the procedure.

Reference parameters set up an intimate connection between actual and formal variables. The formal parameter is merely another name for the actual parameter and no copy is made by the procedure of the actual parameter's value. Both formal and actual parameters point to the same location in memory. A reference parameter must be preceded by REF in the procedure heading. Reference parameters are used to return values to the calling program. Like value parameters, reference parameters are local to the procedure in which they occur and cease to exist when the procedure is finished.

The following two simple programs illustrate the difference between value and reference parameters:

```
0010 // PROGRAM 99
0020 //
0030 // COMELY KATE
0040 //
0050 // TO ILLUSTRATE VALUE PARAMETER
0060 //
0070 PAR:=5
0080 PRINT "VALUE OF PAR BEFORE PROCEDURE IS ",PAR
0090 EXEC VALPROC(PAR)
0100 PRINT
```

```
0110 PRINT "VALUE OF PAR AFTER PROCEDURE IS FINISHED IS ",PAR  
0120 END  
0130 //  
0140 PROC VALPROC(PAR)  
0150 PAR:=13  
0160 ENDPROC VALPROC
```

RUN

VALUE OF PAR BEFORE PROCEDURE IS CALLED IS 5

VALUE OF PAR AFTER PROCEDURE IS FINISHED IS 5

```
0010 // PROGRAM 100  
0020 //  
0030 // COMELY KATE  
0040 //  
0050 // TO ILLUSTRATE REFERENCE PARAMETER  
0060 //  
0070 PAR:=5  
0080 PRINT "VALUE OF PAR BEFORE PROCEDURE IS CALLED IS ",PAR  
0090 EXEC REFFPROC(PAR)  
0100 PRINT  
0110 PRINT "VALUE OF PAR AFTER PROCEDURE IS FINISHED IS ",PAR  
0120 END  
0130 //  
0140 PROC REFFPROC(REF V)  
0150 V:=13  
0160 ENDPROC REFFPROC
```

RUN

VALUE OF PAR BEFORE PROCEDURE IS CALLED IS 5

VALUE OF PAR AFTER PROCEDURE IS FINISHED IS 13

## Closed Procedures

The procedures we have used so far in this chapter are known as open procedures. This means that, apart from the parameters, all other variables used in the procedures are equally accessible to the procedure itself and to the main program and all other procedures. For example the variable DIVISOR used in the procedure FACTORISE in Program 98 could have been used by the main program if it wanted it. In fact it was this *global* nature of the variable VALID which allowed it to be used by the procedure TESTLEAPYEAR after it had been determined by the procedure CHECKVALIDENTRY in Program 97.

Procedures are normally open. This contrasts with functions, which are closed. Procedures can however be closed. All we need to do is to place the word CLOSED in the procedure heading,

e.g.           PROC DOSOMETHING (A,B) CLOSED.

When a procedure is closed all the variables used by the procedure are local to the procedure and are not known outside the procedure. (This does not affect the action of any REF parameters which may be used). Even if variables of the same name are used inside and outside the procedure they do not interfere with each other. They might as well have completely different names for all they know of each other.

Again let us take two simple examples to illustrate the difference between open and closed procedures.

```
0010 // PROGRAM 101
0020 //
0030 // COMELY KATE
0040 //
0050 // TO ILLUSTRATE ORDINARY OPEN PROCEDURE
0060 //
0070 VAR:=5
0080 EXEC OPENPROC
0090 PRINT
0100 PRINT "VALUE OF VAR OUTSIDE PROCEDURE IS ",VAR
0110 END
0120 //
0130 PROC OPENPROC
0140 VAR:=17
0150 PRINT "VALUE OF VAR INSIDE PROCEDURE IS ",VAR
0160 ENDPROC OPENPROC
```

```
RUN
VALUE OF VAR INSIDE PROCEDURE IS 17
VALUE OF VAR OUTSIDE PROCEDURE IS 17
```

```
0010 // PROGRAM 102
0020 //
0030 // COMELY KATE
0040 //
0050 // TO ILLUSTRATE CLOSED PROCEDURE
0060 //
0070 VAR:=5
0080 EXEC CLOSEDPROC
0090 PRINT
0100 PRINT "VALUE OF VAR OUTSIDE PROCEDURE IS ",VAR
0110 END
0120 //
0130 PROC CLOSEDPROC CLOSED
0140 VAR:=17
0150 PRINT "VALUE OF VAR INSIDE PROCEDURE IS ",VAR
0160 ENDPROC CLOSEDPROC
```

```
RUN
VALUE OF VAR INSIDE PROCEDURE IS 17
VALUE OF VAR OUTSIDE PROCEDURE IS 5
```

In Program 101 notice that although VAR has been set to 5 on line 70, when it is printed on line 100 it has the value of 17 which it acquired inside the procedure. On the other hand, in Program 102 VAR keeps its value of 5 throughout the main

program, in spite of the fact that inside the procedure a variable called VAR was given the value 17. The reason of course is that the identifier VAR within the procedure is entirely separate from the identifier VAR within the main program. This is because the procedure is closed.

## Hi again!

Let us return to Program 58 (page 137). There we found the highest number in a set of numbers. We will now rewrite this program using a procedure to perform the task of extracting the highest number. You are invited to return to pages 136 to study once again the structure diagram and revise the method of solution. We will proceed directly with the program.

```
0010 // PROGRAM 103
0020 //
0030 // COMELY KATE
0040 //
0050 // TO FIND THE HIGHEST NUMBER IN A LIST
0060 //
0070 DIM LISST(20)
0080 //
0090 // FIRST SET UP THE LIST
0100 FOR COUNT#:=1 TO 20 DO
0110   READ LISST(COUNT#)
0120 NEXT COUNT#
0130 //
0140 // NOW SEARCH FOR HIGHEST VALUE
0150 //
0160 NUMBER:=0
0170 EXEC GETHIGHEST(NUMBER,LISST)
0180 PRINT
0190 PRINT "THE HIGHEST NUMBER IN THE LIST IS ",NUMBER
0195 //
0200 DATA 12, 54, 78, 67, 56, 45, 48, 98, 34, 34
0210 DATA 87, 76, 65, 84, 94, 89, 34, 56, 77, 88
0220 END
0230 //
0240 PROC GETHIGHEST(REF HIGHEST, REF LISST())
0250   HIGHEST:=LISST(1)
0260   FOR COUNT#:=2 TO 20 DO
0270     IF HIGHEST<LISST(COUNT#) THEN HIGHEST:=LISST(COUNT#)
0280   NEXT COUNT#
0290 ENDPROC GETHIGHEST
```

RUN

THE HIGHEST NUMBER IN THE LIST IS 98

Program 103 is very similar to Program 58, with the loop which searches for the highest value (lines 160 to 190 in Program 58) replaced by the procedure GETHIGHEST in Program 103. One thing may puzzle you. It is clear that HIGHEST must be a 'call by reference' parameter, since it must return the highest value in the list to the corresponding actual parameter NUMBER for printing in the

main program. But why is LISST a REF parameter? After all, LISST is not changed in any way and it is just the values in the LISST that we wish to consult in the procedure. So why is LISST not just a 'call by value' parameter?

The answer is simple. COMAL insists on arrays being 'call by reference' parameters. So we have no choice. But why does COMAL do this?

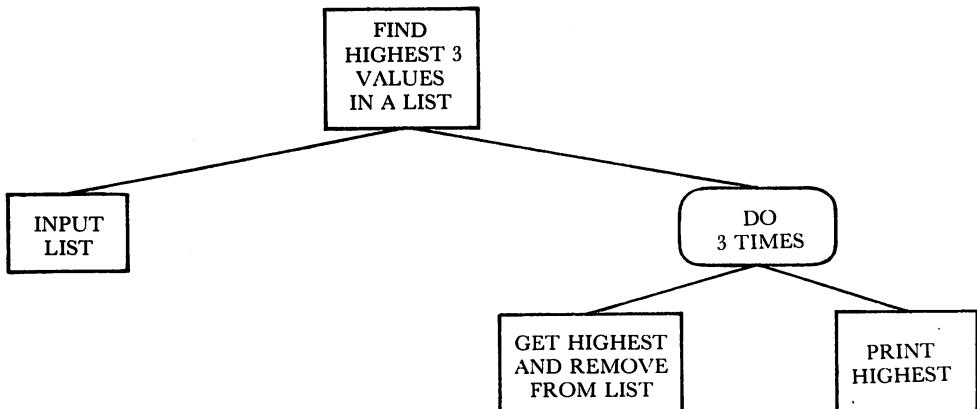
Well, you will remember that copies of value parameters are made by procedures, whereas no copies are made of reference parameters. Because it would often be expensive (both in time and memory space) to make copies of arrays, COMAL insists on the passing of whole arrays by reference so that no copy need be made.

One other important point to observe - the two brackets after LISST in the procedure heading (line 240). These indicate that LISST is a one-dimensional array. Note that the brackets must be empty. We do not repeat the 20 we used to dimension the array on line 70. If we were passing a 2-dimensional array as a parameter, two brackets with a single comma inside would be used — thus (,).

## U3

Suppose we are asked to find the highest *three* numbers in a list. Program 103 can easily be adapted to do this. However, it is not sufficient to set up a loop which executes the procedure GETHIGHEST three times. We would just get the highest number three times, not the highest three numbers.

What we need to do is to cross out the highest number after we have found it and printed it. This means that it would no longer be in contention the next time round. So the second time in searching for the highest value we would pick up the original second highest. It in turn would be crossed out and the third time round we would pick up the third highest.



```

0010 // PROGRAM 104
0020 //
0030 // COMELY KATE
0040 //
0050 // TO FIND THE THREE HIGHEST NUMBERS IN A LIST
0060 //
0070 DIM LISST(20)
0080 //
0090 // FIRST SET UP THE LIST
0100 FOR COUNT#:=1 TO 20 DO
0110   READ LISST(COUNT#)
0120 NEXT COUNT#
0130 //
0140 // NOW SEARCH FOR HIGHEST VALUE
0150 //
0160 NUMBER:=0
0170 FOR COUNT#:=1 TO 3 DO
0180   EXEC GETHIGHEST(NUMBER,LISST)
0190   PRINT
0200   PRINT "THE HIGHEST NUMBER IN THE LIST IS ",NUMBER
0210 NEXT COUNT#
0220 DATA 12, 54, 78, 67, 56, 45, 48, 98, 34, 34
0230 DATA 87, 76, 65, 84, 94, 89, 34, 56, 77, 88
0240 END
0250 //
0260 PROC GETHIGHEST(REF HIGHEST, REF LISST()) CLOSED
0270   HIGHEST:=LISST(1)
0280   POSITION#:=1
0290   FOR COUNT#:=2 TO 20 DO
0300     IF HIGHEST<LISST(COUNT#) THEN
0310       HIGHEST:=LISST(COUNT#)
0320       POSITION#:=COUNT#
0330     ENDIF
0340   NEXT COUNT#
0350   LISST(POSITION#):=-999999,0
0360 ENDPROC GETHIGHEST

```

RUN

```

THE HIGHEST NUMBER IN THE LIST IS  98
THE HIGHEST NUMBER IN THE LIST IS  94
THE HIGHEST NUMBER IN THE LIST IS  89

```

Two important points to be noted:

- (1) The 'crossing out' of the highest value found is done by replacing it in the list by the 'very' negative value -999999. This number is unlikely to come into contention for next highest in any realistic example. Note that, in order to perform this replacement, the position in which the highest value is found must be recorded. Can you see for yourself how the variable POSITION# does this?
- (2) The procedure GETHIGHEST is closed. The main reason for this is to prevent a clash between the variable COUNT# used inside the procedure and the variable COUNT# used in the main program in the loop from line 170 to 210. You should type in the program and omit the word CLOSED and see what happens. Can you figure out why it happens?

We could have used two different variable names of course to avoid a clash. But COUNT# is such a natural word to use and it was a nice opportunity to use a closed procedure.

## Sorting

**Problem:** Read in a list of examination marks and sort them into descending order.

This problem essentially continues the theme of the last two examples. We can use the same strategy as in the last example and achieve our objective by making a few minor changes.

```
0010 // PROGRAM 105
0020 //
0030 // COMELY KATE
0040 //
0050 // TO SORT A LIST OF NUMBERS INTO DESCENDING ORDER
0060 //
0070 READ N#
0080 DIM LISST(N#), SORTEDLIST(N#)
0090 //
0100 // FIRST SET UP THE LIST
0110 //
0120 PRINT
0130 PRINT "THE ORIGINAL LIST IS :"
0140 PRINT
0150 FOR COUNT#:=1 TO N# DO
0160   READ LISST(COUNT#)
0170   PRINT LISST(COUNT#);
0180 NEXT COUNT#
0190 //
0200 // FILL SORTED LIST WITH DUMMY VALUES
0210 //
0220 FOR COUNT#:=1 TO N# DO
0230   SORTEDLIST(COUNT#):=0
0240 NEXT COUNT#
0250 //
0260 // NOW SORT THE LIST
0270 //
0280 EXEC SORT(LISST,N#,SORTEDLIST)
0290 //
0300 // PRINT THE SORTED LIST
0310 //
0320 PRINT
0330 PRINT
0340 PRINT
0350 PRINT "THE SORTED LIST IS :"
0360 PRINT
0370 FOR COUNT#:=1 TO N# DO
0380   PRINT SORTEDLIST(COUNT#);
0390 NEXT COUNT#
0400 //
0410 DATA 20
0420 DATA 12, 54, 78, 67, 56, 45, 48, 98, 34, 34
0430 DATA 87, 76, 65, 84, 94, 89, 34, 56, 77, 88
0440 END
0450 //
0460 PROC SORT(REF LISST(), N#, REF SORTEDLIST())
0470   NUMBER:=0
0480   FOR P#:=1 TO N# DO
0490     EXEC GETHIGHEST(NUMBER,LISST,N#)
0500     SORTEDLIST(P#):=NUMBER
0510   NEXT P#
0520 ENDPROC SORT
```

```

0530 //
0540 PROC GETHIGHEST(REF HIGHEST, REF LISST(), N#) CLOSED
0550   HIGHEST:=LISST(1)
0560   POSITION#:=1
0570   FOR COUNT#:=2 TO N# DO
0580     IF HIGHEST<LISST(COUNT#) THEN
0590       HIGHEST:=LISST(COUNT#)
0600       POSITION#:=COUNT#
0610     ENDIF
0620   NEXT COUNT#
0630   LISST(POSITION#):=-999999.0
0640 ENDPROC GETHIGHEST

```

RUN

THE ORIGINAL LIST IS :

12 54 78 67 56 45 48 98 34 34 87 76 65 84 94 89 34 56 77 88

THE SORTED LIST IS :

98 94 89 88 87 84 78 77 76 67 65 56 56 54 48 45 34 34 12

Points to observe:

- (1) Rather than declaring the arrays LISST and SORTEDLIST to be of a fixed size, say 20, we read in the number of values to be sorted, N# (see line 70) and then declare the arrays to be of just that size (line 80). This is more economic because the arrays are exactly the right size and no space is reserved unnecessarily. It is an example of *dynamic* rather than *static* declaration.
- (2) The sorting is actually done by the procedure SORT which simply calls the procedure GETHIGHEST N# times. This is very similar to the way we called the procedure GETHIGHEST three times in the last program.
- (3) Instead of printing each highest value as it is found (as we did in Program 104) we put it into a new list called SORTEDLIST. The first highest value is placed in position 1 of this list, the second in position 2, and so on, thus giving us a descending list of numbers.
- (4) This SORTEDLIST is passed back to the main program through the mechanism of a 'call by reference' parameter. Of course even if we didn't explicitly want it passed back we would still have had to use REF in the procedure heading, as we did for LISST (line 460) and must for all arrays used as parameters (see discussion on pages 215, 216).
- (5) Before the procedure SORT was called the array SORTEDLIST was filled with zeros (lines 220 to 240). This is a bit of a nuisance in a way because these values will be overwritten when the procedure SORT gets to work. Some versions of COMAL would not require this.
- (6) The procedure GETHIGHEST need not have been closed in this program because we used a different variable, P#, as our loop counter for the call on GETHIGHEST - compare lines 480 to 510 in Program 105 with 170 to 210 in Program 104.

## Searching

Perhaps the main reason for sorting a list into order is to enable it to be searched more quickly and efficiently.

We have already seen that one method of searching a list is to start at the first entry and then run down the list until we either find the item we are looking for, or come to the end of the list. This is the obvious way to search. However, it is not the way we search every list, e.g. a telephone directory.

If we are given a person's name and asked to find his telephone number, we make use of the fact that the names are in alphabetical order. This enables us to proceed without starting at the first name and working our way through. Our search is much more efficient because the names are in order. Have you ever tried to find a person's name, given only his telephone number? Try it! Very tedious and slow, eh? If you think about this, you will realise that this is because the numbers are not arranged in any particular order.

Our job now is to make use of the order in a list to develop an efficient searching technique for inclusion in a computer program. The technique we will consider is known as the *Binary Search*. We will explain the method by means of an example.

**Example** Suppose we are searching for 77 in the ordered list of numbers  
42, 56, 57, 70, 75, 77, 78

(Remember it is easy for you to see that 77 is in the list. But the computer cannot see and must keep comparing 77 with the entries in the list.)

The strategy is as follows:

First examine the middle item in the list. If that is the correct one, we are finished our search.

If the middle entry is less than the one we are looking for we can eliminate the bottom half of the list and confine our search to the top half. On the other hand, if the middle entry is greater than the one we are looking for, we can eliminate the top half and confine our search to the bottom half.

In either case, we have effectively reduced the list of items to be considered to half of what it was. We now look at the middle entry in this half. If the item is the one we want, we are finished our search. Otherwise we can again confine our attention to one half of this half.

Therefore, at each stage we reduce the list by one half.

Now consider the list given above.

There are 7 items in the list. To get the middle position, add 1 and 7 and divide by 2.

$$\frac{1+7}{2} = 4 \quad \text{So, examine item 4.}$$

The 4th entry is 70, which is less than the number we are looking for, namely 77. So we can discard the first half of the list, up to and including 70. We confine our attention now to the 5th, 6th and 7th items.

The middle position now is  $\frac{5 + 7}{2} = 6$

The 6th item is 77. Therefore we have found what we were looking for and are finished.

Note that we have had to look at just two items to find the one we wanted. If we had used the linear search method, starting from the first item, we would have had to look at six items. The Binary Search offers an improvement, you will surely agree. This improvement is even more dramatic if the list is longer. If there were 1 000 items in the list, the Binary Search would look at 10 items at most, to find the one required, while the linear search might need to look at 1 000 items!

In order to make the technique absolutely clear, let us consider another example:

**Example** Search for 52 in the list

14, 22, 50, 60, 70, 75, 80, 86.

$$\text{MIDDLE} = (1 + 8) \text{DIV } 2 = 4$$

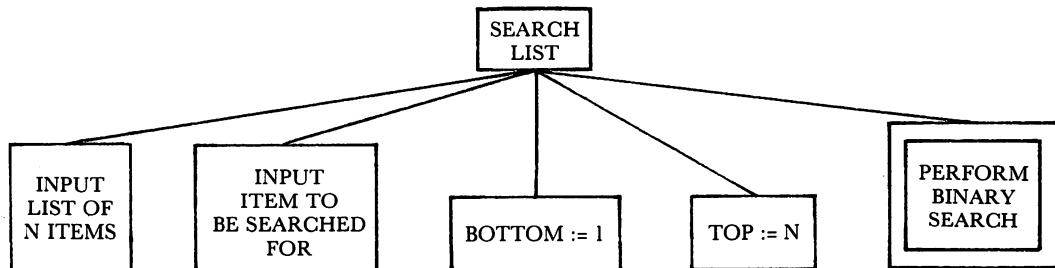
We use DIV because we want an integer answer.

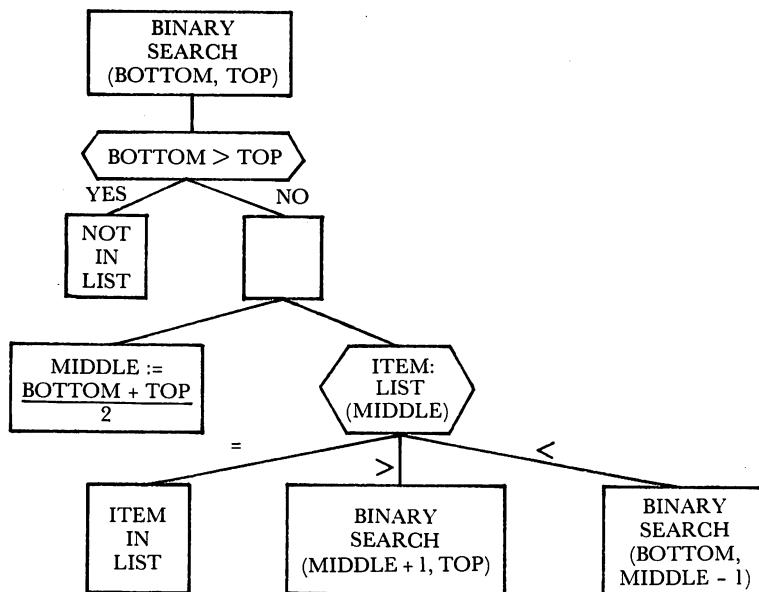
Item 4 = 60, which is greater than 52, so we discard upper half of list and confine our search now to items 1, 2 and 3.

$$\text{MIDDLE} = (1 + 3) \text{DIV } 2 = 2$$

Item 2 = 22 which is less than 52, so we discard items 1 and 2. This leaves us with just item 3. On examining this, we find it is not equal to 52, and so we conclude that 52 is not in the list.

Let us now write a program to illustrate this important binary search method.





```

0010 // PROGRAM 106
0020 //
0030 // COMELY KATE
0040 //
0050 // TO ILLUSTRATE THE BINARY SEARCH
0060 //
0070 DIM LISST(20)
0080 //
0090 // FIRST SET UP THE LIST
0100 //
0110 READ N#
0120 PRINT
0130 PRINT "THE ORIGINAL LIST IS :"
0140 PRINT
0150 FOR COUNT#:=1 TO N# DO
0160   READ LISST(COUNT#)
0170   PRINT LISST(COUNT#);
0180 NEXT COUNT#
0190 //
0200 // PERFORM SEARCH
0210 //
0220 PRINT
0230 PRINT
0240 INPUT "VALUE TO BE SEARCHED FOR ": VALUE
0250 PRINT
0260 EXEC SEARCH(VALUE,LISST,1,N#)
0270 DATA 20
0280 DATA 12, 34, 34, 34, 45, 48, 54, 56, 56, 65
0290 DATA 67, 76, 77, 78, 84, 87, 88, 89, 94, 98
0300 END
0310 //
0320 PROC SEARCH(ITEM, REF LISST(), BOTTOM#, TOP#)
0330 IF BOTTOM#>TOP# THEN
0340   PRINT ITEM," IS NOT IN LIST"

```

```
0350 ELSE
0360   MIDDLE#:=(BOTTOM#+TOP#) DIV 2
0370   IF ITEM=LISST(MIDDLE#) THEN
0380     PRINT ITEM," IS IN LIST IN POSITION ",MIDDLE#
0390   ELIF ITEM>LISST(MIDDLE#) THEN
0400     EXEC SEARCH(VALUE,LISST,MIDDLE#+1,TOP#)
0410   ELSE
0420     EXEC SEARCH(VALUE,LISST,BOTTOM#,MIDDLE#-1)
0430   ENDIF
0440 ENDIF
0450 ENDPROC SEARCH
```

RUN

THE ORIGINAL LIST IS :

12 34 34 34 45 48 54 56 56 65 67 76 77 78 84 87 88 89 94 98

VALUE TO BE SEARCHED FOR 87

87 IS IN LIST IN POSITION 16

RUN

THE ORIGINAL LIST IS :

12 34 34 34 45 48 54 56 56 65 67 76 77 78 84 87 88 89 94 98

VALUE TO BE SEARCHED FOR 98

98 IS IN LIST IN POSITION 20

RUN

THE ORIGINAL LIST IS :

12 34 34 34 45 48 54 56 56 65 67 76 77 78 84 87 88 89 94 98

VALUE TO BE SEARCHED FOR 50

50 IS NOT IN LIST

The Binary Search will only work if the entries in the list are in ascending or descending order. We have chosen to use ascending order in this example. Lines 110 to 180 set up and print the list of data. The value to be searched for is input on line 240 and then the procedure SEARCH is called.

The most significant aspect of SEARCH is that it is a **recursive procedure**.

A recursive procedure is a procedure which calls itself, just as a recursive function is a function which calls itself, as we have seen. The binary search lends itself beautifully to recursion. The list is searched by examining the middle element. If that is not the one which is sought, the procedure is repeated on half the list, then on half of that half and so on if necessary. Note carefully the role the parameters play in the recursive calls. If the item being sought is in the top half of a list (ITEM>LISST(MIDDLE#)) the bottom parameter is moved to MIDDLE#+1 and SEARCH is called once more (line 400). If the item is in the bottom half, the 'top' is 'moved' down to MIDDLE#-1 and SEARCH is called again (line 420).

You are cordially invited to meditate on the elegance of this recursive procedure (write poems in its honour, compose music if you will) and to study carefully how it works.

## 'Bubble, Bubble, Toil and Trouble'

Just as we frequently require to sort lists of numbers into ascending or descending order, so do we often need to sort lists of names into order, usually alphabetical order. Let us now turn our attention to this problem.

We could adapt our previous method of sorting, i.e. continually selecting the highest number in a list and putting it into a new list. This time we would select the alphabetically earliest name in the list, put it into a new list and cross it out in the original list, and then repeat this process until the list is sorted. However, we will develop a new method of sorting instead – a popular method called the *Bubble Sort Method*.

Let us first explain how bubble sort works, by sorting a short list of numbers into ascending order.

### Bubble Sort

**Example** Sort 4, 3, 9, 2 into ascending order.

*1st Pass*      Compare first two elements, 4 and 3.  
These are out of order, so swap them.  
The list now reads 3, 4, 9, 2.  
Compare 4 and 9.  
These are in order, so continue.  
The list now reads 3, 4, 9, 2.  
Compare 9 and 2.  
These are out of order, so swap them.  
The list now reads 3, 4, 2, 9.

The first pass over the list is now finished and the effect has been to cause the largest number, 9, to ‘bubble’ to the end.

The idea now is to leave the 9 where it is and make a second pass over the shorter list 3, 4, 2.

*2nd Pass*      Compare 3 and 4: OK, so continue.  
Compare 4 and 2: out of order, so swap.  
The list now reads 3, 2, 4, 9.

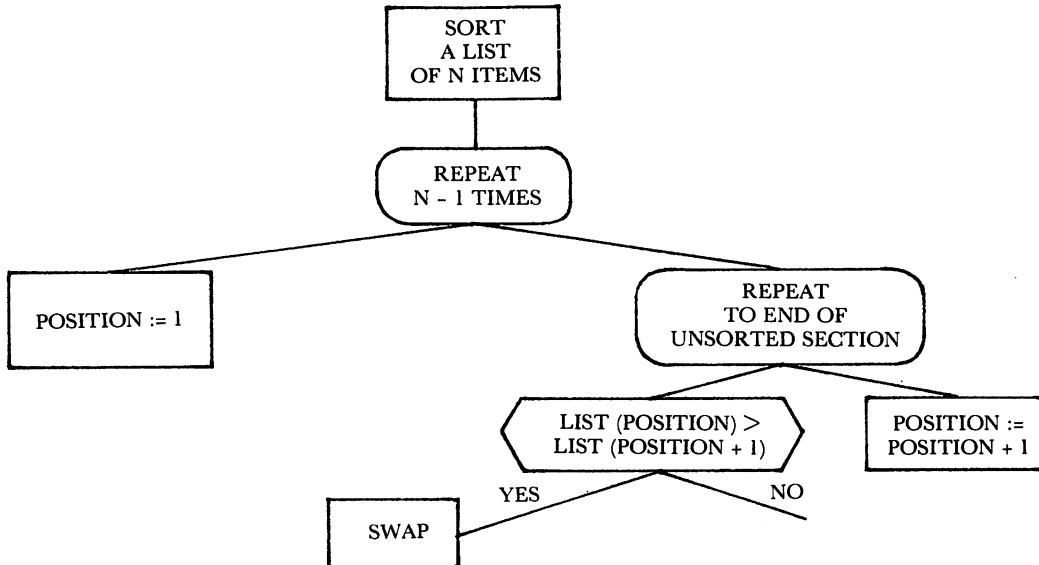
4 has ‘bubbled’ to the end of the shorter list. 4 and 9 are now in their final positions and we may simply turn our attention to the list 3, 2 for the 3rd pass.

*3rd Pass*      Compare 3 and 2: out of order, so swap.  
The list now reads 2, 3, 4, 9 and is sorted.

The bubble sort method therefore proceeds by comparing adjacent elements and swapping them if they are out of order. On the first pass through the list, the largest element moves to the end. This means that it is in its correct final position and will not be moved again.

The list can now be shortened by one and the same procedure repeated until there are only two elements left. These are compared and swapped if necessary.

Notice that with this list of 4 elements three passes over the list are made. In general, in this straightforward method,  $n - 1$  passes will be made over a list of  $n$  elements.



Applying this straightforward bubble sort we get the following program:

```

0010 // PROGRAM 107
0020 //
0030 // COMELY KATE
0040 //
0050 // TO SORT A LIST OF NUMBERS USING BUBBLE SORT
0060 //
0070 READ N#
0080 DIM LISST(N#)
0090 //
0100 // FIRST SET UP THE LIST
0110 PRINT
0120 PRINT "THE ORIGINAL LIST IS :"
0130 PRINT
0140 FOR COUNT#:=1 TO N# DO
0150 READ LISST(COUNT#)
0160 PRINT LISST(COUNT#);
0170 NEXT COUNT#
0180 PRINT
0190 //
0200 // NOW SORT LIST
0210 //
  
```

```

0220 EXEC BUBBLESORT(LISST,N#)
0230 //
0240 PRINT
0250 PRINT
0260 PRINT "THE SORTED LIST IS :"
0270 PRINT
0280 FOR COUNT#:=1 TO N# DO
0290   PRINT LISST(COUNT#);
0300 NEXT COUNT#
0310 DATA 20
0320 DATA 12, 54, 78, 67, 56, 45, 48, 98, 34, 34
0330 DATA 87, 76, 65, 84, 94, 89, 34, 56, 77, 88
0340 END
0350 //
0360 PROC BUBBLESORT(REF LISST(), TOTAL#)
0370   FOR PASS#:=1 TO TOTAL#-1 DO
0380     FOR POSITION#:=1 TO TOTAL#-PASS# DO
0390       IF LISST(POSITION#)>LISST(POSITION#+1) THEN
0400         TEMP:=LISST(POSITION#)
0410         LISST(POSITION#):=LISST(POSITION#+1)
0420         LISST(POSITION#+1):=TEMP
0430       ENDIF
0440     NEXT POSITION#
0450   NEXT PASS#
0460 ENDPROC BUBBLESORT

```

RUN

THE ORIGINAL LIST IS :

12 54 78 67 56 45 48 98 34 34 87 76 65 84 94 89 34 56 77 88

THE SORTED LIST IS :

12 34 34 34 45 48 54 56 56 65 67 76 77 78 84 87 88 89 94 98

Two FOR loops take care of the passes over the list and the movement through the list on each pass. Line 370 sets  $n - 1$  passes to be done since the parameter TOTAL# has taken the value N# from the main program. Note how line 380 causes the list through which POSITION# moves to be shortened by 1 for each pass. Thus, since TOTAL# = 20

when            PASS# = 1, 20 - PASS# = 19  
               PASS# = 2, 20 - PASS# = 18  
               PASS# = 3, 20 - PASS# = 17, etc.

You may have been puzzled by our use of the word 'straightforward' several times in connection with the bubble sort. To appreciate why, consider how the method outlined above would handle the following list: 8, 3, 5, 6, 10, 12.

*1st Pass*      Compare 8 & 3: swap — 3, 8, 5, 6, 10, 12  
               Compare 8 & 5: swap — 3, 5, 8, 6, 10, 12  
               Compare 8 & 6: swap — 3, 5, 6, 8, 10, 12  
               Compare 8 & 10: no swap  
               Compare 10 & 12: no swap: List is 3, 5, 6, 8, 10, 12.

<i>2nd Pass</i>	Compare 3 & 5: no swap
	Compare 5 & 6: no swap
	Compare 6 & 8: no swap
	Compare 8 & 10: no swap: List is 3, 5, 6, 8, 10, 12
<i>3rd Pass</i>	Compare 3 & 5: no swap
	Comapre 5 & 6: no swap
	Compare 6 & 8: no swap: List is 3, 5, 6, 8, 10, 12.
<i>4th Pass</i>	Compare 3 & 5: no swap
	Compare 5 & 6: no swap: List is 3, 5, 6, 8, 10, 12
<i>5th Pass</i>	Compare 3 & 5: no swap: List is 3, 5, 6, 8, 10, 12

It is easy to see that most of the passes were a waste of time. The list was sorted after the first pass. Yet our previous program would take no account of this and would continue on, regardless, performing the other passes. The computer has no way of seeing that the elements are in order, except by comparing them.

The problem is – how can we inform the program that the list is sorted, so that it can avoid going through unnecessary passes?

The answer is – take note of whether any swap is made on each pass. If no swaps are made, then the list is sorted. In the above example, no swaps were made in the 2nd pass and so we can conclude that the list is sorted and not bother with the 3rd, 4th and 5th passes.

But, you may say, we could have avoided the 2nd pass as well. Yes, we humans could, but the computer could not. The best we can do is allow the one extra pass and use the fact that no swaps are made, to avoid further passes. In fact we are not doing too badly! The number of passes is reduced from 5 to 2 – a saving of approximately 50% (remember the later passes are shorter).

Let us now incorporate this pass-saving idea in an improved bubble sort. We will apply it to a list of names this time rather than a list of numbers.

## Naming the Names

```

0010 // PROGRAM 108
0020 //
0030 // COMELY KATE
0040 //
0050 // TO SORT A LIST OF NAMES INTO ALPHABETICAL ORDER
0060 //
0070 //
0080 // FIRST SET UP THE LIST
0090 READ N#
0100 DIM NAMES$(N#) OF 20
0110 FOR COUNT#=1 TO N# DO
0120   READ NAMES$(COUNT#)
0130 NEXT COUNT#
0140 //
0150 // NOW SORT LIST

```

```

0160 //
0170 EXEC SORT(NAMES$,N#)
0180 //
0190 PRINT "THE ALPHABETICAL LIST IS :"
0200 PRINT
0210 PRINT
0220 FOR COUNT#:=1 TO N# DO
0230 PRINT NAMES$(COUNT#)
0240 NEXT COUNT#
0250 DATA 20
0260 DATA "JOE SOAP", "HANDY ANDY", "BILLY BONES"
0270 DATA "DILLY DREAMER", "ELLIE ESTER", "BERT BRIGHT"
0280 DATA "JILL BRIGHT", "JOEY O'NEILL", "TOM JONES"
0290 DATA "DON CANUTE", "MUHAMMAD ALI", "JOE LOUIS"
0300 DATA "BIG JIM", "MARCEL MARAT", "FREDDIE FEARLESS"
0310 DATA "GEORGE MERRIMAN", "BIG BILL", "SMALL BILL"
0320 DATA "TOM MIX", "HOPALONG HOP"
0330 END
0340 //
0350 PROC SORT(REF LISST$(), N#)
0360 DIM TEMP$ OF 20
0370 PASS#:=0
0380 REPEAT
0390   PASS#:=PASS#+1
0400   SWAP:=FALSE
0410   SCANLENGTH#:=N#-PASS#
0420   FOR SCAN#:=1 TO SCANLENGTH# DO
0430     IF LISST$(SCAN#)>LISST$(SCAN#+1) THEN
0440       TEMP$:=LISST$(SCAN#)
0450       LISST$(SCAN#):=LISST$(SCAN#+1)
0460       LISST$(SCAN#+1):=TEMP$
0470     SWAP:=TRUE
0480   ENDIF
0490   NEXT SCAN#
0500 UNTIL NOT SWAP OR PASS#=N#-1
0510 ENDPROC SORT

```

RUN

THE ALPHABETICAL LIST IS :

BERT BRIGHT  
 BIG BILL  
 BIG JIM  
 BILLY BONES  
 DILLY DREAMER  
 DON CANUTE  
 ELLIE ESTER  
 FREDDIE FEARLESS  
 GEORGE MERRIMAN  
 HANDY ANDY  
 HOPALONG HOP  
 JILL BRIGHT  
 JOE LOUIS  
 JOE SOAP  
 JOEY O'NEILL  
 MARCEL MARAT  
 MUHAMMAD ALI  
 SMALL BILL  
 TOM JONES  
 TOM MIX

## **Important points**

- (1) We cannot use a FOR loop for the passes this time, because we don't know how many passes we need. Therefore we replace the FOR loop of Program 107 (lines 370 to 450) by a REPEAT loop which continues until either no swap has been made during a pass or the maximum number, N# - 1, of passes have been executed.
- (2) At the beginning of each pass a logical variable called SWAP is set to FALSE and only if a swap is actually made during the pass is SWAP changed to TRUE. If this has not happened, then when the UNTIL statement on line 500 is reached, SWAP will still be false and so NOT SWAP will be true and will cause termination of the REPEAT loop.
- (3) As we have seen before (Program 67) swapping cannot be performed directly. Thus if a swap is necessary as determined by the test on line 430, one of the strings to be swapped must be moved into a temporary variable which we have named TEMP\$. Notice that the dimension of TEMP\$ has been declared inside the procedure itself (line 360). This is in keeping with the idea that a procedure is effectively a program in its own right.
- (4) Strings are compared using the same relational operation ( $>$ ) as was used for numbers (line 430). In effect such questions as

is "JOE SOAP"  $>$  "HANDY ANDY"

are being asked.

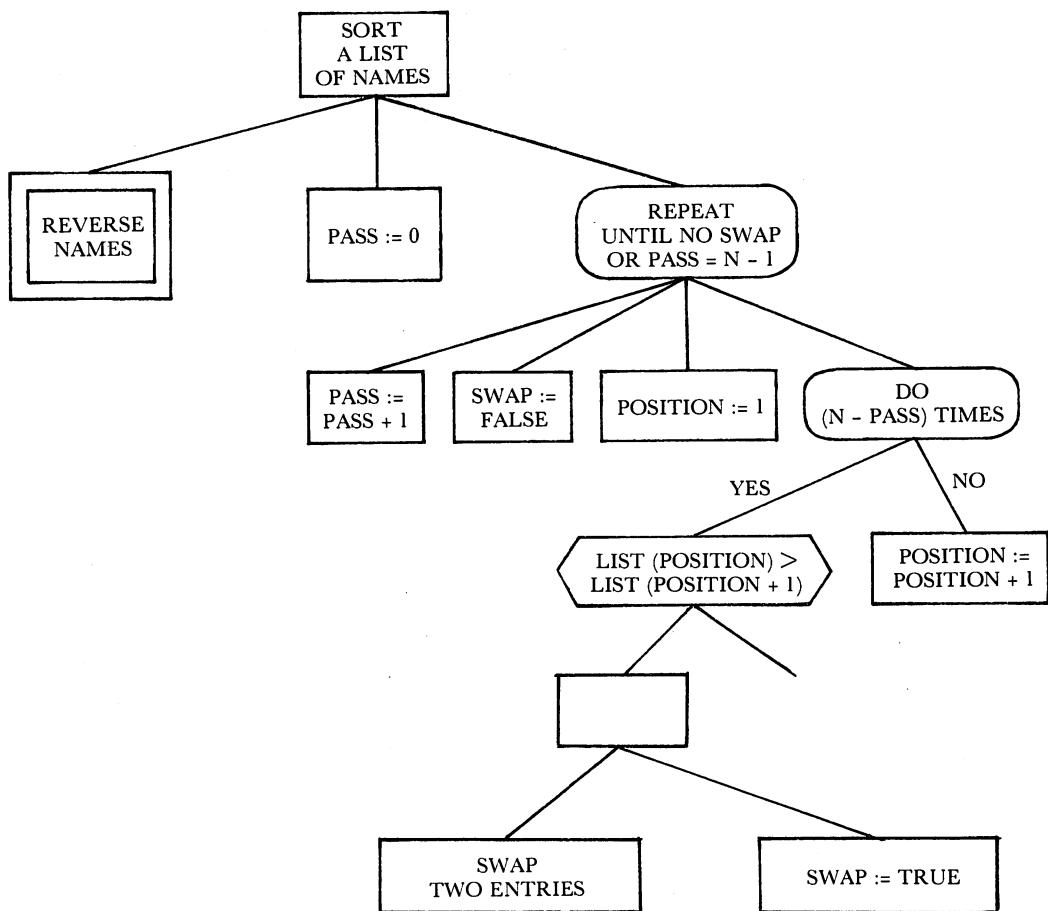
What do they mean?

When a string of characters is being compared with another string, they are compared character by character from the left. In the above example "JOE SOAP" is greater than "HANDY ANDY", because J, the very first letter of JOE SOAP is 'greater' (i.e. later in the alphabet) than H, the first letter of HANDY ANDY. But ALAN A. BLIGHT  $<$  ALAN A. BRIGHT because although they are the same up to the 10th character, the 10th character itself, L, in the first name is earlier in the alphabet than the 10th character, R, in the second name.

## **Surname first, please!**

No doubt you said to yourself as you surveyed the results of the last program - 'That is not what we mean when we think of an alphabetical list of names'. And of course you are right (as usual!). It is the surnames which should be in alphabetical order, not the christian names.

The computer, as we have seen, compares strings from left to right. If the christian name is first it will mainly determine the order. So we must reverse the christian name and surname before we make a comparison. Well, we have already seen how to do this in Program 63. Let us now convert the main idea in this program into a procedure which will reverse the christian names and surnames of all the entries in a list before sorting the list.



The program follows fairly directly from the structure diagram. Once again we see one procedure calling another.

```

0010 // PROGRAM 109
0020 //
0030 // COMELY KATE
0040 //

```

```

0050 // TO SORT A LIST OF NAMES INTO ALPHABETICAL ORDER
0060 //
0070 //
0080 // FIRST SET UP THE LIST
0090 READ N#
0100 DIM NAMES$(N#) OF 20
0110 FOR COUNT#:=1 TO N# DO
0120   READ NAMES$(COUNT#)
0130 NEXT COUNT#
0140 //
0150 // NOW SORT LIST
0160 //
0170 EXEC SORT(NAMES$,N#)
0180 //
0190 PRINT "THE ALPHABETICAL LIST IS :"
0200 PRINT
0210 PRINT
0220 FOR COUNT#:=1 TO N# DO
0230   PRINT NAMES$(COUNT#)
0240 NEXT COUNT#
0250 //
0260 DATA 20
0270 DATA "JOE SOAP", "HANDY ANDY", "BILLY BONES"
0280 DATA "DILLY DREAMER", "ELLIE ESTER", "BERT BRIGHT"
0290 DATA "JILL BRIGHT", "JOEY O'NEILL", "TOM JONES"
0300 DATA "DON CANUTE", "MUHAMMAD ALI", "JOE LOUIS"
0310 DATA "BIG JIM", "MARCEL MARAT", "FREDDIE FEARLESS"
0320 DATA "GEORGE MERRIMAN", "BIG BILL", "SMALL BILL"
0330 DATA "TOM MIX", "HOPALONG HOP"
0340 END
0350 //
0360 PROC SORT(REF LISST$(), N#)
0370   DIM TEMP$ OF 20
0380   EXEC REVERSENAMES(LISST$,N#)
0390   PASS#:=0
0400   REPEAT
0410     PASS#:=PASS#+1
0420     SWAP:=FALSE
0430     SCANLENGTH#:=N#-PASS#
0440     FOR SCAN#:=1 TO SCANLENGTH# DO
0450       IF LISST$(SCAN#)>LISST$(SCAN#+1) THEN
0460         TEMP$:=LISST$(SCAN#)
0470         LISST$(SCAN#):=LISST$(SCAN#+1)
0480         LISST$(SCAN#+1):=TEMP$
0490         SWAP:=TRUE
0500     ENDIF
0510   NEXT SCAN#
0520   UNTIL NOT SWAP OR PASS#=N#-1
0530 ENDPROC SORT
0540 //
0550 PROC REVERSENAMES(REF LISST$(), N#)
0560   DIM LEFTPART$ OF 20, RIGHTPART$ OF 20
0570   FOR COUNT#:=1 TO N# DO
0580     POSITION#:=0
0590     REPEAT
0600       POSITION#:=POSITION#+1
0610       UNTIL LISST$(COUNT#,POSITION#)=" "
0620       LEFTPART$:=LISST$(COUNT#,1,POSITION#)
0630       LENGTH:=LEN(LISST$(COUNT#))
0640       RIGHTPART$:=LISST$(COUNT#,POSITION#+1,LENGTH)
0650       LISST$(COUNT#):=RIGHTPART$+" "+LEFTPART$
0660     NEXT COUNT#
0670 ENDPROC REVERSENAMES

```

RUN  
THE ALPHABETICAL LIST IS :  
ALI MUHAMMAD  
ANDY HANDY  
BILL BIG  
BILL SMALL  
BONES BILLY  
BRIGHT BERT  
BRIGHT JILL  
CANUTE DON  
DREAMER DILLY  
ESTER ELLIE  
FEARLESS FREDDIE  
HOP HOPALONG  
JIM BIG  
JONES TOM  
LOUIS JOE  
MARAT MARCEL  
MERRIMAN GEORGE  
MIX TOM  
O'NEILL JOEY  
SOAP JOE .

## Summary

Effective problem solving requires the breaking down of problems into smaller problems.

Procedures allow programs to be composed of solutions to sub-programs.

There are two ingredients in the use of procedures -

- (1) Procedure definition
- (2) Procedure call

A procedure definition consists of

- (1) a heading where a name is given to the procedure
- (2) the body of the procedure where the instructions comprising the procedure are spelled out
- (3) the closing of the procedure using ENDPROC

A procedure is called by its name preceded by the word EXEC.

Parameters enable information to be passed explicitly between a calling program and a procedure.

Parameters which occur in a procedure call are known as actual parameters.

Parameters which occur in a procedure heading are known as formal parameters.

'Call by value' parameters accept values *from* a calling statement but do not return any values *to* the calling program.

'Call by reference' parameters can both accept values from and return values *to* a calling program.

Variables other than parameters in a procedure are open to other procedures and the main program unless the procedure in which they occur is closed. Procedures may be recursive.

#### COMAL KEYWORDS

ABS AND AUTO CASE CAT CLEAR CON COS  
DATA DEF DEL DELETE DIM DIV DO EDIT  
ELIF END ENDCASE ENDDEF ENDIF ENDPROC  
ENDWHILE EOD EXEC EXP FN FOR GLOBAL  
IF...THEN...ELSE IN INPUT INT LEN LIST LOAD  
MOD NEW NEXT NOT OR OTHERWISE PRINT  
PROC READ REF RENUM REPEAT RND ROUND  
RUN SAVE SIN STEP STOP TAB UNTIL VAL  
WHEN WHILE

#### QUESTIONS and EXERCISES

1. What are procedures? Why are they used? How are they defined? How are they called?
2. Distinguish between
  - (1) Actual and Formal Parameters
  - (2) Value and Reference Parameters
  - (3) Open and Closed Procedures
  - (4) Local and Global variables.
3. What is meant by recursion? Describe in your own words recursive procedures for
  - (1) taking a walk
  - (2) validating an identifier in COMAL.
4. Sort the following lists into descending order by hand, using the Bubble Sort technique:
  - (1) 83, 80, 75, 92, 98, 90
  - (2) ABC, ABCD, ABCDE, ABDC, ABC
5. Modify Program 105 so that it sorts into ascending order.
6. Have a friend think of a number between 1 and 100. Use the Binary Search technique to guess the number as quickly as possible, assuming you are allowed to ask questions such as - is it equal to?, is it less than?
7. Write a *non-recursive* version of the Binary Search in Program 106.
8. Write procedures to draw
  - (a) a horizontal line on the screen
  - (b) a vertical line on the screen.Use these procedures to draw a rectangle.
9. Write a program to determine the longest word in a given sentence.
10. Write a program (a) to change all the A's in a given sentence into \*'s.  
(b) to change all the vowels into \*'s.
11. Rewrite as many of the programs in the book as you are interested in using procedures.

# 14 Files

## The concept of a Data File

At this stage you may be feeling a little uneasy about some of the sample programs, particularly those at the end of the last chapter. This uneasiness probably arises from two factors:

- (1) The data lines appear to be clumsy and out of keeping with the rest of the program. This would be even more apparent if the amount of data was as much as it would be in a realistic example. Leaving out the data lines and using input would of course be no answer; it would be far too slow.
- (2) The same data is used in a number of programs. It seems desirable to hold this data somewhere separate from the programs and allow the programs to access it and use it as required.



In fact the data can be held separately from a program by making use of a **Data File**. No doubt you have heard of files and have a rough idea of what they are. For example, your school may hold a personal file of information about you. This file may contain

- (a) your name
- (b) your present address
- (c) your examination results since you entered the school
- (d) information on your aptitudes
- (e) your job interests
- (f) your sporting and recreational interests

The examination secretary will probably use this file when sending you your examination results. The Careers Guidance teacher will use it when advising you on choice of careers. The headmaster will make use of it when discussing your progress (or lack of it!) with your parents. Thus the same file may be used by different people for different purposes.

A data file used by a program is similar to a file used in school, or in business, or in any other situation.

*A file is an organised collection of data.*

The organization is usually designed to facilitate processing for some specific purpose. A file is usually considered to consist of one or more records, and records are often considered to consist of one or more fields, e.g.

#### CLASS FILE

JOE SOAP	4 GREEN ST., SOMEWHERE	7/12/1965
DILLY DREAMER	IVORY TOWER, SUNLAND	5/5/1965
FREDDIE FEARLESS	THE MAZE, DOWNTOWN	8/7/1965
ABE ATHERTON	10 UPPING ST., THERE	10/8/1965
BILLY BONES	THE MARKET, SEATOWN	7/4/1965

The file above consists of five records. Each record consists of three fields - a field holding a person's name, a field for the address and a field for the date of birth.

### Storing of Files

We are already aware that information may be recorded and stored in many ways -

- on punched cards
- on punched paper tape
- on magnetic tape
- on magnetic disk, etc.

Magnetic tape and magnetic disk are perhaps the most popular and widely used media at present and our job now is to consider how to use these media

- (a) to create and store data files
- (b) to retrieve and use data files.

Methods of dealing with files differ slightly from computer to computer. However, the basic principles are the same for all systems. Provision is always made for the following operations:

- (1) open a file
- (2) read from a file
- (3) write to a file
- (4) close a file

We begin our consideration of file operations with some very simple examples.

## **One record - one file**

Our first program simply  
opens a file for writing,  
writes a single string record to the file,  
closes the file.

```
0010 // PROGRAM 110
0020 //
0030 // COMELY KATE
0040 //
0050 // TO WRITE A RECORD TO A FILE
0060 //
0070 OPEN FILE 1, "NAME", WRITE
0080 PRINT FILE 1: "JOE SOAP"
0090 CLOSE FILE 1
0100 END
```

## **Explanation**

OPEN FILE 1, “NAME”, WRITE opens file number 1, gives it the title NAME and indicates that information is to be **written** to it. The file will be opened on the default disk drive. We could in fact mention the disk drive explicitly.

OPEN FILE 1, “DKO: NAME”, WRITE would open a file called NAME on disk drive DKO:

OPEN FILE 1, “DK1: NAME”, WRITE would open a file called NAME on disk drive DK1:

PRINT FILE 1: “JOE SOAP” writes a single record consisting of the string “JOE SOAP” onto the file NAME opened above. Notice that it is the file number which is used, not the title of the file. Notice also the colon which must be used.

CLOSE FILE 1 closes the file NAME.

Having created a file, let us now read from this file and display the information on the screen.

```

0010 // PROGRAM 111
0020 //
0030 // COMELY KATE
0040 //
0050 // TO RECOVER A RECORD FROM A FILE
0060 //
0070 DIM NAME$ OF 20
0080 OPEN FILE 1, "NAME", READ
0090 INPUT FILE 1; NAME$
0100 PRINT NAME$, " HAS BEEN RECOVERED FROM THE FILE"
0110 CLOSE FILE 1
0120 END
RUN
JOE SOAP HAS BEEN RECOVERED FROM THE FILE

```

## Explanation

OPEN FILE 1, "NAME", READ opens file number 1 called NAME for the purpose of **reading** information from it.

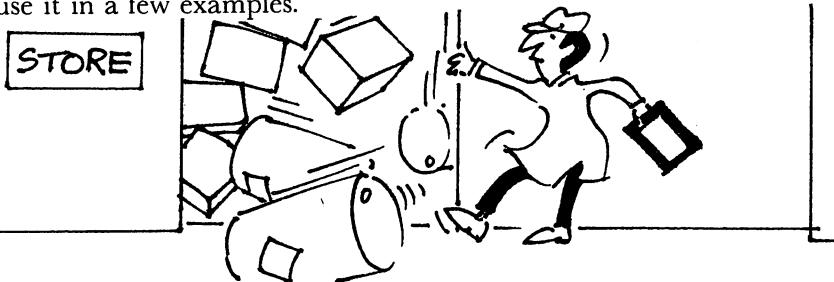
INPUT FILE 1; NAME\$ reads a record from file number 1 and stores it in NAME\$.

Note that the identifier NAME\$ has nothing to do with the identifier NAME for the file. Totally different identifiers could have been chosen.

CLOSE FILE 1 closes file number 1. Note that we used the same file number 1 in both Programs 110 and 111. We need not have done this. For example we could have used 1 in Program 110 and 3 in Program 111. The same file number must be used throughout a given use of a file, but the next time the file is opened a different number may be used. The file number must be one of 0 to 9, inclusive.

## Stock File

Now that we have learned how to create and use files, let us set up a small but realistic file and use it in a few examples.



One of the problems encountered by businesses of all kinds, day after day, is the problem of keeping track of all the items in their stocks. In order to keep customers happy a business premises must be well stocked. Yet it is costly and wasteful to carry too much stock. Therefore it is important to have a good strategy for managing stock, so that an adequate supply is kept and when the level of a particular item falls below an acceptable level, more items are ordered in good time.

Computers can be invaluable in helping with the stock control problem.

Let us begin with a simple program to set up a file containing a list of all the items in stock. Suppose our warehouse is stocked as follows:

ITEM	QUANTITY IN STOCK
Cornflakes	4560
Rice Krispies	3987
Weetabix	4138
Sugar Puffs	2262
All Bran	986
Wheat Flakes	1429
Puffa Puffa Rice	876
Frosties	1345

```
0010 // PROGRAM 112
0020 //
0030 // COMELY KATE
0040 //
0050 // TO SET UP A SIMPLE STOCK FILE
0060 //
0070 // FIRST READ NUMBER OF ITEMS IN STOCK
0080 READ NUMBER#
0090 //
0100 DIM STOCKITEM$ OF 20
0110 //
0120 // READ IN STOCK INFORMATION AND PUT IN FILE
0130 //
0140 PRINT
0170 OPEN FILE 1, "DK1:STOCK", WRITE
0180 FOR COUNT#:=1 TO NUMBER# DO
0190   READ STOCKITEM$, NUMBER
0200   PRINT FILE 1: STOCKITEM$
0210   PRINT FILE 1: NUMBER
0220 NEXT COUNT#
0230 CLOSE FILE 1
0240 //
0250 // NOW READ BACK FILE AND PRINT IT ON THE SCREEN
0260 //
0270 DIM ITEM$ OF 20, FIELD$ OF 20
0280 FIELD$#="*****"
0290 OPEN FILE 1, "DK1:STOCK", READ
0300 WHILE NOT EOF(1) DO
0310   INPUT FILE 1: ITEM$, NUMBER
0320   PRINT USING FIELD$: ITEM$,
0330   PRINT USING "*****": NUMBER
0340 ENDWHILE
0350 CLOSE FILE 1
0360 //
0370 DATA 8
0380 DATA "CORNFLAKES", 4560
0390 DATA "RICE KRISPIES", 3987
0400 DATA "WEETABIX", 4138
0410 DATA "SUGAR PUFFS", 2262
0420 DATA "ALL BRAN", 986
0430 DATA "WHEAT FLAKES", 1429
0440 DATA "PUFFA PUffa RICE", 876
0450 DATA "FROSTIES", 1345
0460 END
```

RUN

CORNFLAKES	4560
RICE KRISPIES	3987
WEETABIX	4138
SUGAR PUFFS	2262
ALL BRAN	986
WHEAT FLAKES	1429
PUFFA PUFFA RICE	876
FROSTIES	1345

Line 170 opens a file called STOCK on disk drive DK1: for writing. The loop from line 180 to 220 then repeatedly takes an item name e.g. CORNFLAKES and a corresponding number e.g. 4560 from the data list and puts them into the file (lines 200, 210). The file is then closed.

In order to check that everything is OK we open the file again, read back the information and print it on the screen. So that the information is displayed neatly on the screen we introduce a new aspect of the PRINT statement, namely the PRINT USING command. The PRINT USING command essentially allows us to specify the field width of the item to be printed in the actual PRINT line itself. Thus on line 330 we specify that NUMBER is to be printed in a field of width 5 by putting 5# signs in inverted commas before NUMBER. Similarly line 320 specifies that ITEM\$ is to have a field width of 20 because the string FIELD\$ has been given 20# signs in line 280.

An important point to remember about PRINT USING is that strings are printed starting from the left. If the string being printed does not fill the field specified, blanks are printed to the right to fill out the field. On the other hand, with numbers the extra blanks are put in at the left so that the number appears at the right-most part of the field.

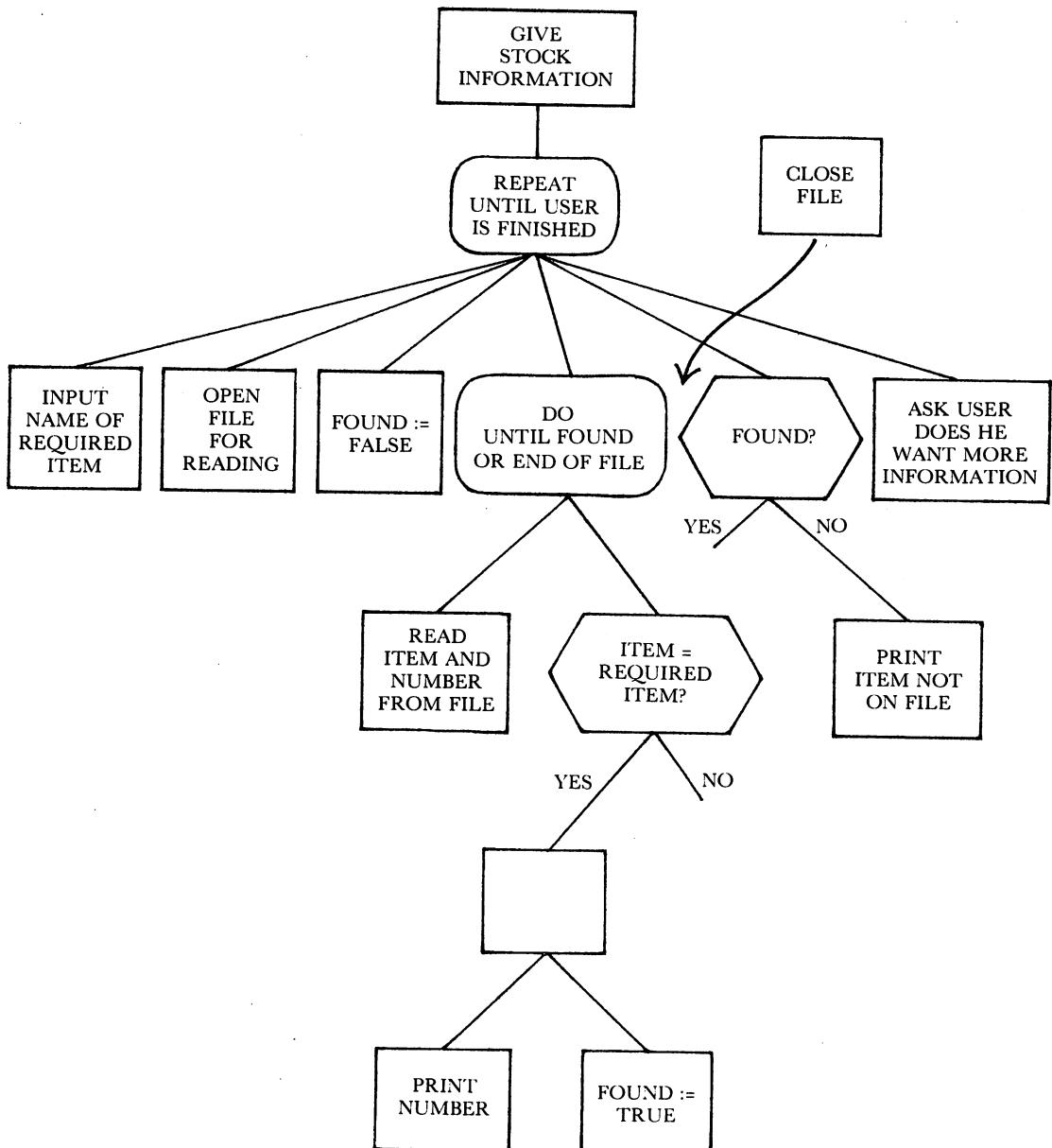
There are a number of rules relating to the full use of the PRINT USING statement. Rather than detail them here we will leave you to experiment for yourself. Exciting, isn't it?

Returning to our program we see that the rest of the loop to display the stock file is straightforward. Note carefully however how neatly we can set up the loop (line 300) by using the end of file identifier, EOF. When a file is opened EOF is automatically set to FALSE by the system and when the last value in the file has been read EOF is set to TRUE. Therefore WHILE NOT EOF (1) says - while the last value in file 1 has not yet been read. This is usually the most convenient way to set up a loop to read through a file.

Line 310 simply reads two items from file 1 and places them in ITEM\$ and NUMBER respectively.

## Examining the File

Now that we have set up our stock file we may use it in lots of ways. Let us write a program to answer queries from users concerning the amount of a particular item held in stock.



```
0010 // PROGRAM 113
0020 //
0030 // COMELY KATE
0040 //
0050 // TO PROVIDE STOCK INFORMATION
0060 //
0070 DIM ITEM$ OF 20, DESCRIPTION$ OF 20, REPLY$ OF 3
0080 REPEAT
0090 PRINT
0100 INPUT "PLEASE ENTER NAME OF ITEM ON WHICH INFORMATION IS REQUIRED ";
ITEM$
0110 PRINT
0120 //
0130 OPEN FILE 1, "DK1:STOCK", READ
0140 FOUND:=FALSE
0150 WHILE NOT FOUND AND NOT EOF(1) DO
0160   INPUT FILE 1: DESCRIPTION$, QUANTITY
0170   IF ITEM$=DESCRIPTION$ THEN
0180     PRINT "THERE ARE ",QUANTITY," BOXES OF ",ITEM$," IN STOCK"
0190   FOUND:=TRUE
0200 ENDIF
0210 ENDWHILE
0220 CLOSE FILE 1
0230 IF NOT FOUND THEN PRINT ITEM$," IS NOT IN STOCK"
0240 PRINT
0250 PRINT
0260 INPUT "WOULD YOU LIKE MORE INFORMATION (Y/N) "; REPLY$
0270 UNTIL REPLY$(1)="N"
0280 END
```

RUN

PLEASE ENTER NAME OF ITEM ON WHICH INFORMATION IS REQUIRED CORNFLAKES

THERE ARE 4560 BOXES OF CORNFLAKES IN STOCK

WOULD YOU LIKE MORE INFORMATION (Y/N) Y

PLEASE ENTER NAME OF ITEM ON WHICH INFORMATION IS REQUIRED FROSTIES

THERE ARE 1345 BOXES OF FROSTIES IN STOCK

WOULD YOU LIKE MORE INFORMATION (Y/N) Y

PLEASE ENTER NAME OF ITEM ON WHICH INFORMATION IS REQUIRED ALL BRAN

THERE ARE 986 BOXES OF ALL BRAN IN STOCK

WOULD YOU LIKE MORE INFORMATION (Y/N) Y

PLEASE ENTER NAME OF ITEM ON WHICH INFORMATION IS REQUIRED CRACKERJACKS

CRACKERJACKS IS NOT IN STOCK

WOULD YOU LIKE MORE INFORMATION (Y/N) N

The program should be fairly easy to follow. Note that we are not obliged to use the same variable names when reading information back from a file as when putting the information into the file. Thus we use DESCRIPTION\$ and QUANTITY in this program for the item name and amount in stock respectively, whereas we used ITEM\$ and NUMBER in the last program.

## Out of Stock?

We have already mentioned that computers can help in a re-ordering strategy for stock. To see this let us add some more information to our stock file. For each item we will include a re-order number in addition to the number of items in stock. The idea is that if the quantity in stock falls below the re-order level, the computer will print a reminder to order more of the item in question. Our stock file will now contain three entries for each item and will look as follows:

ITEM	QUANTITY IN STOCK	RE-ORDER LEVEL
CORNFLAKES	4560	1000
RICE KRISPIES	3987	800
WEETABIX	4138	700
SUGAR PUFFS	2262	500
ALL BRAN	986	400
WHEATFLAKES	1429	400
PUFFA PUFFA RICE	876	300
FROSTIES	1345	400

The following program adds this re-order information to the stock file.

```
0010 // PROGRAM 114
0020 //
0030 // COMELY KATE
0040 //
0050 // TO SET UP A STOCK FILE WITH REORDER INFORMATION
0060 //
0070 DIM ITEM$ OF 20
0080 //
0090 // READ IN STOCK INFORMATION AND SET UP FILE
0100 //
0110 OPEN FILE 1, "DK1:STOCK", READ
0120 OPEN FILE 2, "DK1:STOCKFIL", WRITE
0130 WHILE NOT EOF(1) DO
0140   INPUT FILE 1: ITEM$, NUMBERINSTOCK
0150   READ REORDERLEVEL
0160   PRINT FILE 2: ITEM$
0170   PRINT FILE 2: NUMBERINSTOCK
0180   PRINT FILE 2: REORDERLEVEL
0190 ENDWHILE
0200 CLOSE FILE 1
0210 CLOSE FILE 2
0220 //
0230 DATA 1000, 800, 700, 500, 400, 400, 300, 400
0240 END
```

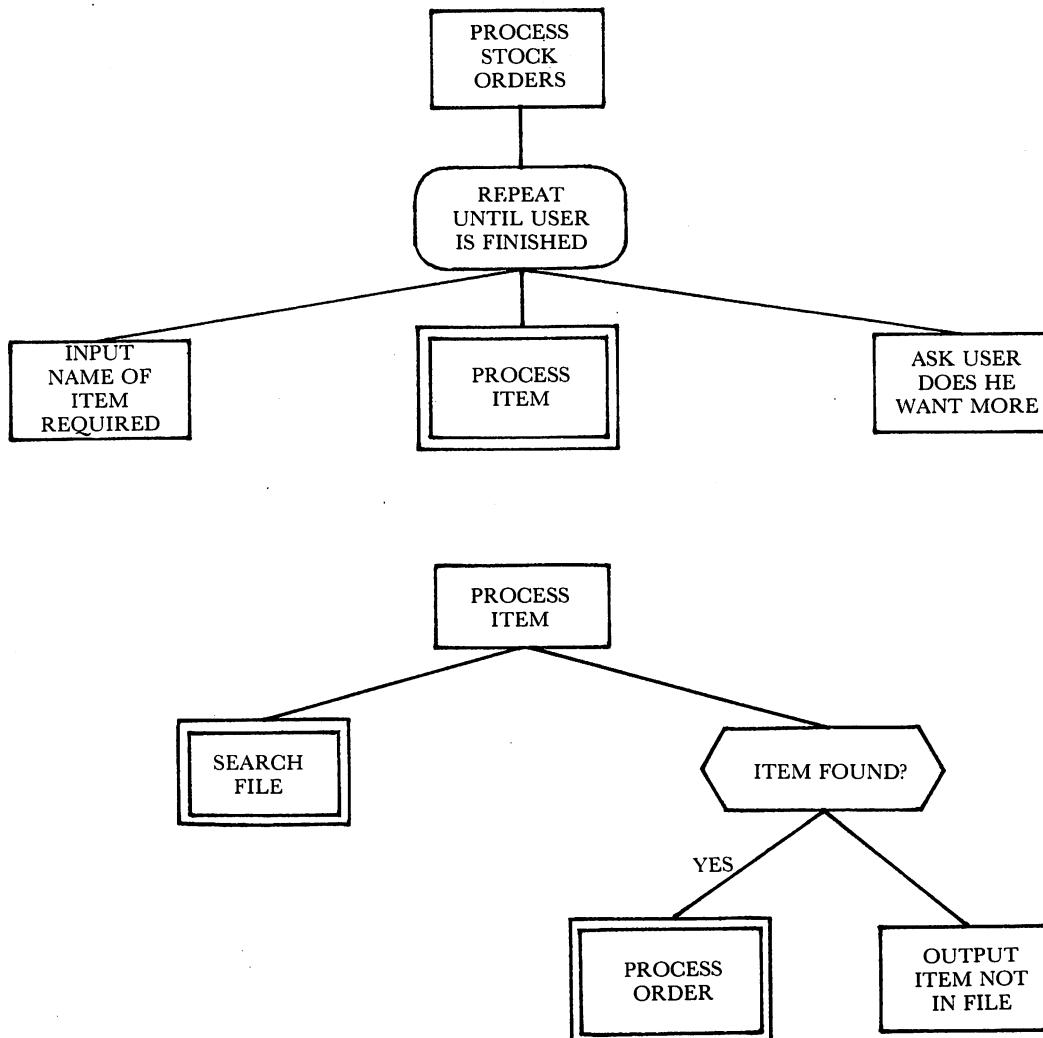
Two files are open at the same time in this program, both on disk drive DK1:  
(1) File 1 called STOCK for reading from,  
(2) File 2 called STOCKFIL for writing to.

Rather than trying to add the re-order numbers to the existing STOCK file we create a new file called STOCKFIL and write to it a combination of information from the STOCK file and the data in the program. Thus line 140 gets two items of information from the STOCK file, while line 150 gets one item from the program

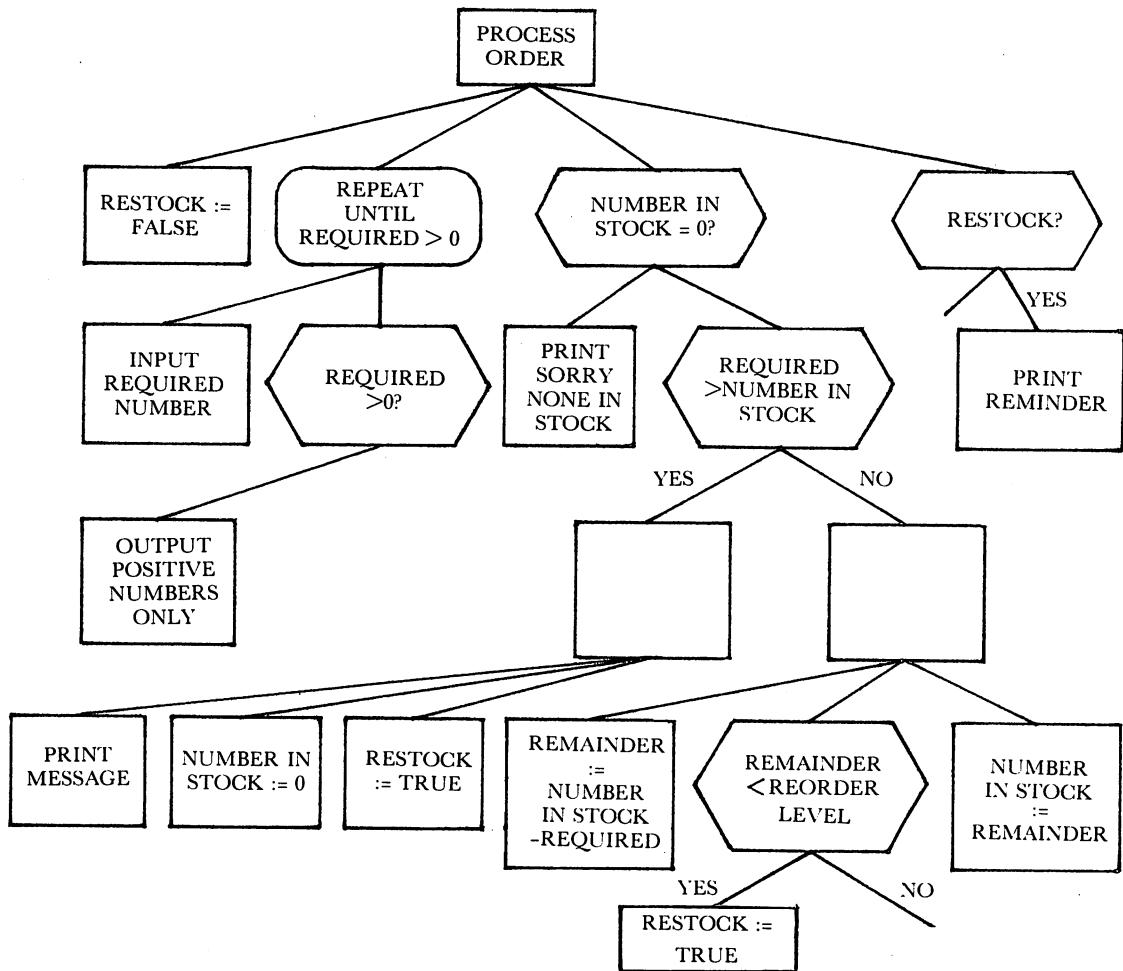
data. Lines 160 to 180 write the three items to the new file. All this is done over and over again until the end of the STOCK file is reached. The two files are then gracefully closed.

## Warehouse Management

Having set up this fuller version of the stock file we will use it to process orders for items and to issue reminders when the stock level falls too low.



We have already seen how a file may be searched (Program 113) so we will concentrate on elaborating PROCESS ORDER.



Following the breaking up of the task into sub-problems the program naturally breaks up into procedures.

```

0010 // PROGRAM 115
0020 //
0030 // COMELY KATE
0040 //
0050 // TO ILLUSTRATE SIMPLE STOCK MANAGEMENT
0060 //
0070 DIM ITEM$ OF 20, REQUIRED$ OF 20, REPLY$ OF 3
0080 //
0090 PRINT
0100 REPEAT
  
```

```

0110 PRINT
0120 INPUT "WHICH ITEM DO YOU REQUIRE  ": REQUIRED$
0130 PRINT
0140 //
0150 // SEARCH FOR ITEM
0160 //
0170 EXEC PROCESSITEM
0180 //
0190 PRINT
0200 PRINT
0210 INPUT "WOULD YOU LIKE ANYTHING ELSE(Y/N) ?  ": REPLY$
0220 UNTIL REPLY$="N"
0230 END
0240 //
0250 PROC PROCESSITEM
0260 FOUND:=FALSE
0270 EXEC SEARCHFILE(FOUND)
0280 IF NOT FOUND THEN
0290   PRINT REQUIRED$," IS NOT IN STOCK LIST"
0300 ELSE
0310   EXEC PROCESSORDER
0320 ENDIF
0330 ENDPROC PROCESSITEM
0340 //
0350 PROC SEARCHFILE(REF FOUND)
0360 OPEN FILE 1, "DK1:STOCKFIL", READ
0370 WHILE NOT FOUND AND NOT EOF(1) DO
0380   INPUT FILE 1: ITEM$, NUMBERINSTOCK, REORDERLEVEL
0390   IF REQUIRED$=ITEM$ THEN FOUND:=TRUE
0400 ENDWHILE
0410 CLOSE FILE 1
0420 ENDPROC SEARCHFILE
0430 //
0440 PROC PROCESSORDER
0450 PRINT
0460 RESTOCK:=FALSE
0470 REPEAT
0480   INPUT "HOW MANY BOXES ?  ": REQUIRED
0490   PRINT
0500   PRINT
0510   IF REQUIRED<0 THEN
0520     PRINT "POSITIVE INTEGER VALUES ONLY "
0530   ELIF NUMBERINSTOCK=0 THEN
0540     PRINT "SORRY - THERE ARE NONE LEFT"
0550   PRINT
0560   ELIF REQUIRED>NUMBERINSTOCK THEN
0570     PRINT "THERE ARE ONLY ",NUMBERINSTOCK," LEFT .YOU MAY TAKE ALL OF
          THEM. THANK YOU FOR YOUR CUSTOM "
0580   NUMBERINSTOCK:=0
0590   RESTOCK:=TRUE
0600 ELSE
0610   PRINT "YOUR ORDER HAS BEEN PROCESSED. THANK YOU FOR YOUR CUSTOM"
0620   REMAINDER:=NUMBERINSTOCK-REQUIRED
0630   IF REMAINDER<REORDERLEVEL THEN RESTOCK:=TRUE
0640   NUMBERINSTOCK:=REMAINDER
0650 ENDIF
0660 UNTIL REQUIRED>0
0670 IF RESTOCK THEN
0680   PRINT
0690   PRINT "THE QUANTITY OF ",REQUIRED$," IS NOW ",NUMBERINSTOCK
0700   PRINT "MORE SHOULD BE ORDERED ***"
0710 ENDIF
0720 ENDPROC PROCESSORDER

```

RUN

WHICH ITEM DO YOU REQUIRE SUGAR PUFFS

HOW MANY BOXES ? 350

YOUR ORDER HAS BEEN PROCESSED. THANK YOU FOR YOUR CUSTOM

WOULD YOU LIKE ANYTHING ELSE(Y/N) ? Y

WHICH ITEM DO YOU REQUIRE FROSTIES

HOW MANY BOXES ? 5000

THERE ARE ONLY 1345 LEFT .YOU MAY TAKE ALL OF THEM. THANK YOU FOR YOUR CUSTOM

THE QUANTITY OF FROSTIES IS NOW 0  
MORE SHOULD BE ORDERED \*\*\*\*

WOULD YOU LIKE ANYTHING ELSE(Y/N) ? Y

WHICH ITEM DO YOU REQUIRE CORNFLAKES

HOW MANY BOXES ? 4000

YOUR ORDER HAS BEEN PROCESSED. THANK YOU FOR YOUR CUSTOM

THE QUANTITY OF CORNFLAKES IS NOW 560  
MORE SHOULD BE ORDERED \*\*\*\*

WOULD YOU LIKE ANYTHING ELSE(Y/N) ? N

Although the program is reasonably long you should have no great difficulty following it. Only one small attempt is made to allow for mistakes by the user on input. If the user enters a negative number when he is asked how many boxes he requires, he is given a warning message and the program loops around again (lines 470 to 520).

The procedure PROCESSITEM calls both of the other procedures SEARCHFILE and PROCESSORDER. The parameter FOUND in the procedure SEARCHFILE is a 'call by reference' parameter since it must return information to the calling program as to whether the item being sought has been found or not.

## Random or Direct Files

The files we have been dealing with up to now are known as sequential files. In order to read any entry in the file it is necessary to work through all the entries from the beginning until the required entry is found. It is not possible to pick out a particular record at random directly from the file. Sometimes it is useful to be able to read a particular record directly. COMAL provides us with a facility for doing this by allowing us to specify RANDOM rather than sequential files. Let us study this type of file by setting up a small library of books on a direct file.

```
0010 // PROGRAM 116
0020 //
0030 // COMELY KATE
0040 //
0050 // TO SET UP LIBRARY FILE
0060 //
0070 // FIRST READ NUMBER OF BOOKS IN LIBRARY
0080 READ NUMBER#
0090 //
0100 DIM BOOK$ OF 40
0110 //
0120 // READ IN LIST OF BOOKS AND SET UP FILE
0130 //
0140 OPEN FILE 3, "LIBRARY", RANDOM , 40
0150 FOR COUNT#:=1 TO NUMBER# DO
0160   READ BOOK$
0170   PRINT FILE 3, COUNT#: BOOK$
0180 NEXT COUNT#
0190 CLOSE FILE 3
0200 //
0210 DATA 10
0220 DATA "ESSAYS ON EDUCATION"
0230 DATA "THE MIGHTY MICRO"
0240 DATA "THE PENGUIN BOOK OF CHESS POSITIONS"
0250 DATA "THE AIMS OF EDUCATION"
0260 DATA "MEN OF MATHEMATICS"
0270 DATA "VISION IN ELEMENTARY MATHEMATICS"
0280 DATA "LESSONS IN CHESS STRATEGY"
0290 DATA "COMPULSORY MISEDUCATION"
0300 DATA "PRACTICAL CHESS ENDGAMES"
0310 DATA "MATHEMATICS IN WESTERN CULTURE"
0320 END
```

The first thing to notice is line 140

OPEN FILE 3, "LIBRARY", RANDOM, 40

↑  
Same as always  
↑  
specifies RANDOM file for reading  
or writing      ↑  
                    ↑  
                    specifies size  
                    of each record

The first parts of the OPEN statement are the same as before where we specify the file number and the name of the file. Then the word RANDOM takes the place of READ or WRITE. A RANDOM file may be read from, or written to, when it has been opened.

The last entry on the line specifies the size of each record in the file, i.e. the total number of bytes taken up by each record. In general integers require 2 bytes, decimal numbers 4 bytes and strings 2 more than the number of symbols specified in the DIM statement for the string.

e.g.        987 requires 2 bytes  
          9.87 requires 4 bytes  
          "JOE SOAP" requires 10 bytes.

In our program we specify a record length of 40 which is a little longer than we need for the longest title in our library.

Line 170 puts a record consisting of a book name into the file and *assigns a record number* (COUNT# in this case). Notice carefully the syntax of the PRINT statement. When COUNT# = 2, for example, it reads PRINT FILE 3, 2: "THE MIGHTY MICRO" and thus "THE MIGHTY MICRO" is entered as the second record in the file. It is because each record is numbered that we can later retrieve any record directly. To see this direct retrieval let us write a very simple program to extract the 7th and 4th books from our little library.

```
0010 // PROGRAM 117
0020 //
0030 // COMELY KATE
0040 //
0050 // TO CHECK LIBRARY FILE
0060 //
0070 DIM BOOK$ OF 40
0080 //
0090 OPEN FILE 3, "LIBRARY", RANDOM , 40
0100 INPUT FILE 3, 7: BOOK$
0110 PRINT
0120 PRINT "THE 7TH BOOK IN THE FILE IS "
0130 PRINT
0140 PRINT BOOK$
0150 PRINT
0160 INPUT FILE 3, 4: BOOK$
0170 PRINT
0180 PRINT "THE 4TH BOOK IN THE FILE IS "
0190 PRINT
0200 PRINT BOOK$
0210 CLOSE FILE 3
0220 END
```

RUN

THE 7TH BOOK IN THE FILE IS

LESSONS IN CHESS STRATEGY

THE 4TH BOOK IN THE FILE IS

THE AIMS OF EDUCATION

The program scarcely requires any comment. However note that the INPUT statements (lines 100 and 160) use the record number to extract the appropriate record directly from the file.

## Letting it all out

Although the main reason for using RANDOM files is to enable individual records to be retrieved directly, these files can be used in a sequential manner as the following program shows. The problem is to search through the library looking for all books with CHESS in their titles.

```
0010 // PROGRAM 118
0020 //
0030 // COMELY KATE
0040 //
0050 // TO FIND ALL BOOKS ON CHESS
0060 //
0070 DIM BOOK$ OF 40, WORD$ OF 30
0080 //
0090 PRINT
0100 PRINT "THE FOLLOWING BOOKS ARE ABOUT CHESS"
0110 PRINT
0120 TOTAL#:=10
0130 OPEN FILE 3, "LIBRARY", RANDOM , 40
0140 FOR COUNT#:=1 TO TOTAL# DO
0150 INPUT FILE 3, COUNT#: BOOK$
0160 EXEC CHECKTITLE("CHESS",BOOK$)
0170 NEXT COUNT#
0180 CLOSE FILE 3
0190 END
0200 //
0210 PROC CHECKTITLE(WORD$, TITLE$)
0220 IF WORD$ IN TITLE$ THEN
0230   PRINT
0240   PRINT TITLE$
0250 ENDIF
0260 ENDPROC CHECKTITLE
```

RUN

THE FOLLOWING BOOKS ARE ABOUT CHESS

THE PENGUIN BOOK OF CHESS POSITIONS

LESSONS IN CHESS STRATEGY

PRACTICAL CHESS ENDGAMES

Each record is read in turn from the random file LIBRARY and checked by the procedure CHECKTITLE to see if the word CHESS appears in the title. The value "CHESS" is passed directly to the formal parameter WORD\$ in the calling statement on line 160. This is the first time we have used a constant as an actual parameter but it can be done whenever value parameters are being used. *Constants cannot however be used in calling statements when reference parameters are involved.* This is because values could not be passed back to constants - constants have their own values. For example in the above program if WORD\$ was a reference parameter in the procedure CHECKTITLE we could not have used the constant "CHESS" in the calling statement on line 160. Although WORD\$ does not change in the procedure, reference parameters must be able to accommodate changes and WORD\$ should be

in a position to pass back any new value to its corresponding actual parameter. But WORD\$ could not change the constant "CHESS". *When reference parameters are used, the corresponding actual parameters must be variables.*

Notice that we do not use the natural WHILE NOT EOF loop to search through the file. Why is this? Because we cannot use EOF with a random file. Therefore we are obliged to remember the number of records in the file and set up a loop accordingly. We remember that there are 10 records in the file, so we set TOTAL# to 10 in line 120 and run a FOR loop from 1 to TOTAL# (lines 140 to 170). Of course our memories are fallible (well, mine certainly is!) and it might be better to record the number of records in the file itself, say as the first record. You are invited to do this yourself and to modify Programs 117 and 118 accordingly.

## Hello there!

We have already referred to the fact that it is extremely tedious to have to search a telephone book looking for someone's name when we know only the telephone number. Can random files be used to relieve this tedium? Well, let us investigate by setting up a small telephone list on file and then searching it for a name when the telephone number is given.

```
0010 // PROGRAM 119
0020 //
0030 // COMELY KATE
0040 //
0050 // TO SET UP A TELEPHONE FILE
0060 //
0070 DIM NAME$ OF 20
0080 // FIRST SET UP THE LIST
0090 READ N#
0100 OPEN FILE 1, "TELEFON", RANDOM , 22
0110 FOR COUNT#:=1 TO N# DO
0120   READ NAME$, PHONENUMBER#
0130   PRINT FILE 1, PHONENUMBER#: NAME$
0140 NEXT COUNT#
0150 CLOSE FILE 1
0160 END
0170 //
0180 DATA 20
0190 DATA "JOE SOAP", 9018, "HANDY ANDY", 7776, "BILLY BONES", 3456
0200 DATA "DILLY DREAMER", 2109, "ELLIE ESTER", 5646, "BERT BRIGHT", 3335
0210 DATA "JILL BRIGHT", 3345, "JOEY O'NEILL", 9652, "TOM JONES", 8578
0220 DATA "DON CANUTE", 9098, "MUHAMMAD ALI", 3835, "JOE LOUIS", 7616
0230 DATA "BIG JIM", 8242, "MARCEL MARAT", 4908, "FREDDIE FEARLESS", 5754
0240 DATA "GEORGE MERRIMAN", 9997, "BIG BILL", 4984, "SMALL BILL", 5674
0250 DATA "TOM MIX", 5846, "HOPALONG HOP", 9097
```

Program 119 sets up the telephone file as a random file with records of length 22 - note 2 more than the dimension of the names which are to be placed in the file.

The important point is that the phone number is used to number each record. The phone number is not actually placed in the record but is used as an index pointing to the associated name (see line 130). This imposes one restriction on us, however. Record numbers must be integers and integers must be in the range -32767 to 32767.

This means we cannot use the six digit telephone numbers which apply in Dublin, for example. Well, Dublin is not the centre of the world and so we have chosen to use 4-digit numbers such as would apply in important towns like Wicklow, etc. There are of course ways around this restriction, but to keep our program simple we have shied away from too much sophistication.

To make use of this file we now present the following program:

```
10 // PROGRAM 120
0020 //
0030 // COMELY KATE
0040 //
0050 // TO FIND NAME OF PERSON WITH GIVEN TELEPHONE NUMBER
0060 //
0070 DIM NAME$ OF 20
0080 INPUT "PHONE NUMBER ? ":" PHONE#
0090 PRINT
0100 TRAP ERR-
0110 OPEN FILE 2, "TELEFON", RANDOM , 22
0120 INPUT FILE 2, PHONE#: NAME$
0130 CLOSE FILE 2
0140 TRAP ERR+
0150 IF ERR THEN
0160 PRINT "NAME NOT LISTED"
0170 ELSE
0180 PRINT NAME$
0190 ENDIF
0200 END
```

RUN

PHONE NUMBER ? 2109

DILLY DREAMER

RUN

PHONE NUMBER ? 9097

HOPALONG HOP

RUN

PHONE NUMBER ? 6677

NAME NOT LISTED

Notice how beautifully direct our telephone query is. No searching is required. We simply input from the file the name which is in the record indexed by PHONE# - line 120.

But what happens if that telephone number is not listed in the file? Well, under normal circumstances the program would halt with an ugly and embarrassing error message on the screen staring accusingly at us. But in this program we have circumvented all that, and that is where the two lines 100 TRAP ERR- and 140 TRAP ERR+ come in.

The COMAL system has a little private variable called **ERR**. Whenever it

recognises an error it puts a number corresponding to the error into this variable. If no error has arisen the value in ERR will be 0. The command TRAP ERR- tells the system to record the error number as usual but not to stop the program. TRAP ERR+ tells the system to go back to its normal method of handling errors.

So in Program 120 we place the command TRAP ERR- before we deal with the file so that if a non-existent record is requested ERR will be set but the program will not stop. We restore the error handling to its normal state by TRAP ERR+ after the file has been dealt with. If ERR has been set we know that no record for that phone number was found and we print a message to that effect (see lines 150 & 160).

The use of TRAP ERR is extremely helpful in many cases. For example, you have probably found that very often a program comes grinding to a halt because you made a silly mistake when INPUT was requested, say by pressing RETURN instead of Y or N in answer to a question. Now you can go back and improve all those programs by using TRAP ERR.

## Summary

Files are organised collections of data usually maintained for specific purposes.

Four operations are important in dealing with files:

- (1) Opening a file
- (2) Writing to a file
- (3) Reading from a file
- (4) Closing a file.

Files may be sequential or random.

EOF is a COMAL system variable which is TRUE when the end of a sequential file is reached, FALSE otherwise.

EOF cannot be used with random files.

Record size must be specified for random files.

ERR is a system variable which holds an error number when an error occurs, 0 when no error occurs.

TRAP ERR allows normal error handling to be switched on and off.

## COMAL KEYWORDS

ABS AND AUTO CASE CAT CLEAR CON COS  
DATA DEF DEL DELETE DIM DIV DO EDIT  
ELIF END ENDCASE ENDDEF ENDIF ENDPROC  
ENDWHILE EOD EOF ERR EXEC EXP FN FOR

GLOBAL IF...THEN...ELSE IN INPUT INT LEN  
LIST LOAD MOD NEW NEXT NOT OR  
OTHERWISE PRINT PROC READ REF RENUM  
REPEAT RND ROUND RUN SAVE SIN STEP STOP  
TAB TRAP ERR UNTIL VAL WHEN WHILE

## QUESTION and EXERCISES

1. What is a file? Why are files important?
2. Write statements to open a file for (a) reading (b) writing on both disk drives.
3. Open a random file to contain
  - (a) integers
  - (b) real numbers
  - (c) strings of maximum length 30.
4. Write programs to set up and write files containing
  - (a) your name, address and telephone number
  - (b) the odd numbers between 1 and 100
  - (c) the square numbers between 1 and 100
  - (d) the names of all the students in your class
  - (e) a list of all the Bank Holidays in 1983
  - (f) the dates of all the World Cup matches in 1982.
5. Using file (a) from Question 4, change the telephone number.
6. Write a program which allows the following changes to be made to the stock file used in this chapter:
  - (a) decrease quantity in stock when orders are filled
  - (b) increase quantities in stock
  - (c) include new items in stock.
7. Set up a file of stolen cars and use it to check a suspect number.
8. Using file (d) from Question 4 write programs to
  - (a) sort the file into alphabetical order and create a new file
  - (b) print out all the students whose names begin with K or O.
9. Project: Create a file of items of local historical interest. Structure the file so as to allow various queries to be answered.
10. Modify Programs 119 and 120 to allow telephone numbers with more than five digits to be included.
11. Write a program to merge two files of alphabetically sorted names to produce a new file which is also alphabetically sorted.
12. Program 120 assumes that the only error which can occur is a name not listed. This may not be the case. Can you take account of this?

# 15 Problem Solving and Program Production

## What we have been at

Throughout the book we have been solving little problems and writing programs. We have found that problems have had to be broken down into small manageable steps and that structure diagrams were frequently of great assistance in providing a clear view of the solution. In many cases the program could be written directly from the structure diagram while in others just a little extra attention to detail was required.

When solving problems it is vital to give due consideration to the data on which the solution will operate. Very often the method of solution will depend on whether the data is numeric or string, and if it is numeric whether it is integer or real. We have seen how the data – telephone numbers – was an important consideration in the programs at the end of the last chapter.

No less important than the given data of course is the output data or results – what form the results will take, how they will be laid out, what remarks will accompany them, etc.

In general it is important to clearly visualise both input and output data and to develop your algorithms accordingly. Gradual refinement of the view of the data and of the corresponding instructions must usually be done before the full solution finally emerges. Do not be afraid to make mistakes. It's how we learn! In particular, mistakes can help us to see what errors to allow for when somebody who is inexperienced is using our programs. As we have already mentioned we have not made allowance for many of the input errors which could arise in our programs because we wished to keep things simple. You are once again encouraged to amend all the programs to try and make them error-proof. You may find it useful to use TRAP ERR.

Small problems may not require too much by way of refinement before the solution is fully and finally clear. But larger problems may take a considerable amount of time to solve. To help you in this absorbing task we now set out some advice.

## Some Hints on Program Construction and Debugging

### Steps in Solving Problems by Computer

#### (1) Define the problem carefully

Make sure you know what the input will be, how much is required, what form it should take, what mistakes might arise, etc. What processing is required? How much output is required? What form should it take? How much should there be?

*(2) Draft an outline of the solution to the problem*

Use a structure diagram. Break the solution down into small steps and think through each section carefully. Maybe you have solved similar problem sections before. Could you use your previous solutions? Document your solution as you go along, i.e. write down comments, explanations and reasons for making different choices.

*(3) Write a program to express your solution.*

Think of writing procedures for each of the different sections. Dry run at least the important stages.

*(4) Type in the program and run it.*

*(5) When it doesn't work, debug the program.*

Trace the execution in order to detect the sources of error.

*(6) When the program appears to be running correctly try it out on different sets of data.*

Make sure all possibilities are taken into account. Make sure it deals correctly with the awkward cases.

## **Documentation**

Each program of any reasonable size should be accompanied by supporting documentation. The documentation may perhaps be included in the program in the form of comments, or it may reside in a separate document apart from the program. In the latter case the program should nevertheless contain remarks to assist a reader to understand it. Documentation should include the following:

- (a) Programmer's Name
- (b) Date
- (c) What the problem is
- (d) What the program does
- (e) A description of the input data and some specimen test data
- (f) A description of the output
- (g) A description of how to use the program
- (h) A sample run
- (i) An outline of what interaction with the program is possible
- (j) A discussion of the capabilities and limitations of the program.

## **Nods and Winks**

1. Think first, write your program later.
2. Never write a large program. Break each problem into small steps.
3. Don't try to be too clever. Simple is best.
4. Make everything clear. Choose clarity in preference to efficiency. Perhaps use recursive procedures.

5. Use variable names which have some relation to what they are used for e.g. COUNT#, STARTINGVALUE, NAME\$.
6. Include meaningful comments, but don't over-comment. Comment to explain, not to describe. There is no need to comment when an instruction itself is clear, e.g.

```
50 // THE NEXT STATEMENT SETS COUNT# TO THE VALUE 1
60 COUNT := 1
```

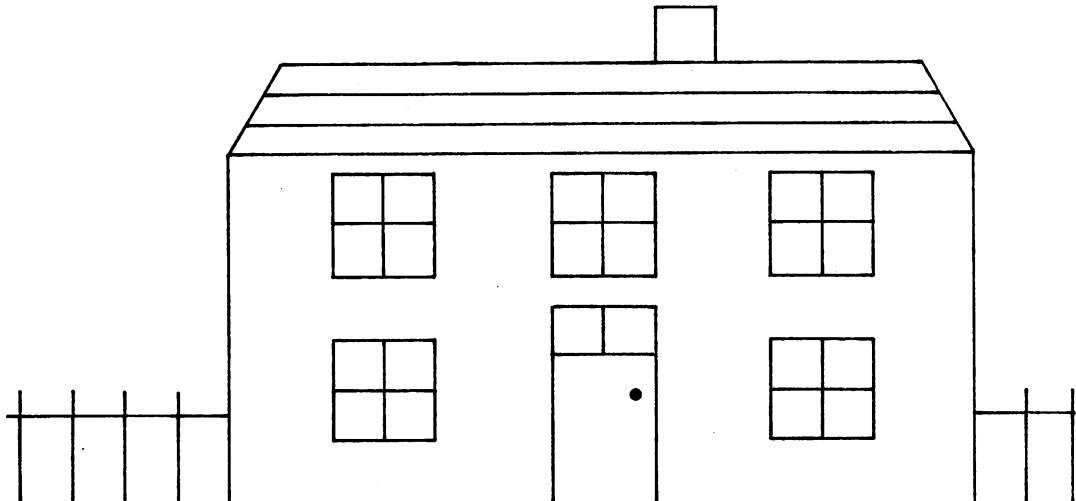
This is a useless comment.

7. Make programs read clearly and logically from top to bottom. Use well-named and purposeful procedures.
8. Even if the output does not require it, always print the input data with the results. It is always more meaningful to see the result in relation to the given data.
9. Make your output readable. Never give just a bare number or list of numbers. Include explanatory messages. Make your output clear, tidy and stylish.
10. Choose the correct algorithmic structure for the task in hand e.g. FOR loops for fixed iteration, CASE statement for multiple choice, etc.
11. When debugging a program, put in *extra* PRINT statements, so that you can see how the program was going before it went wrong.

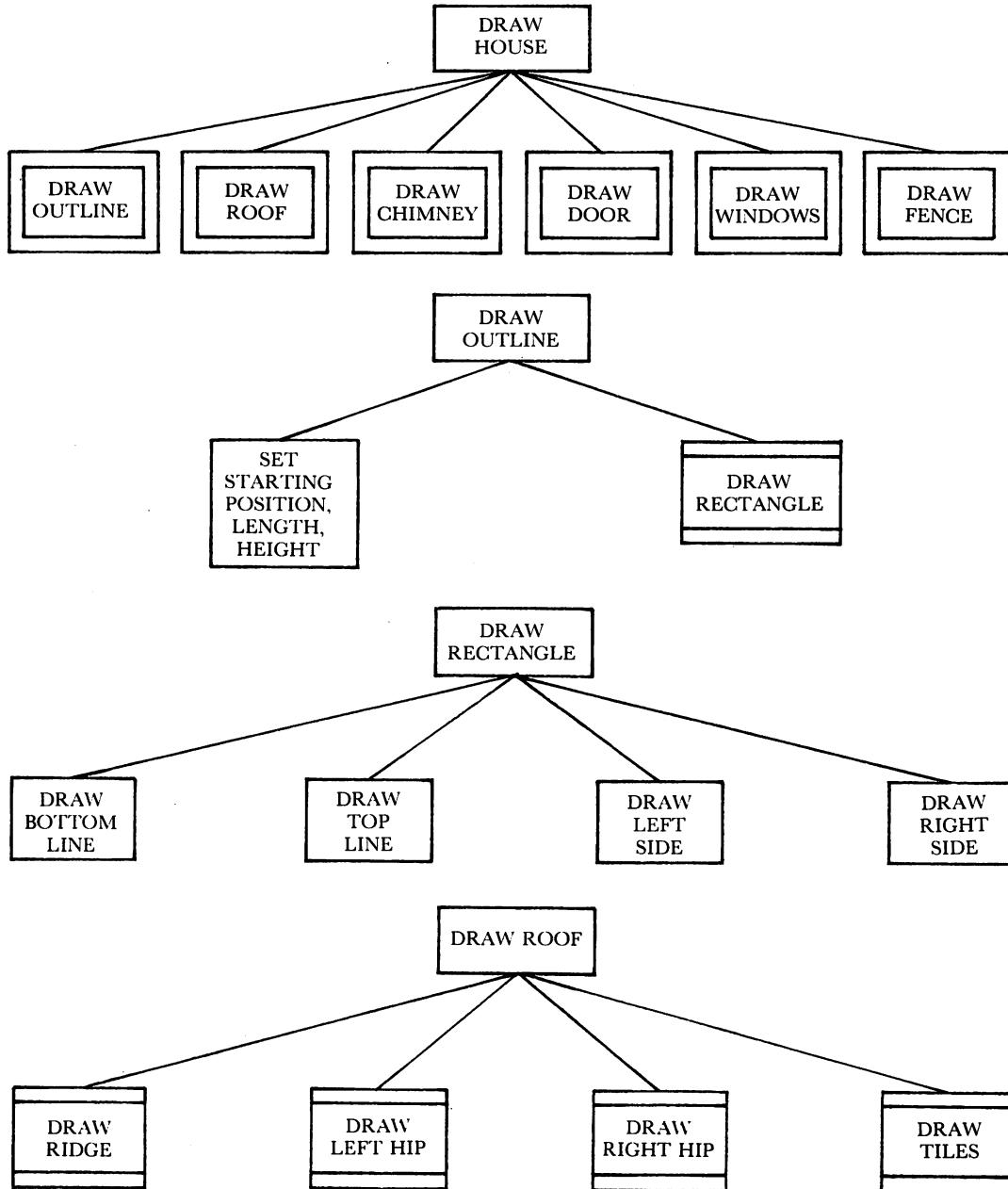
## The House that Jack Built

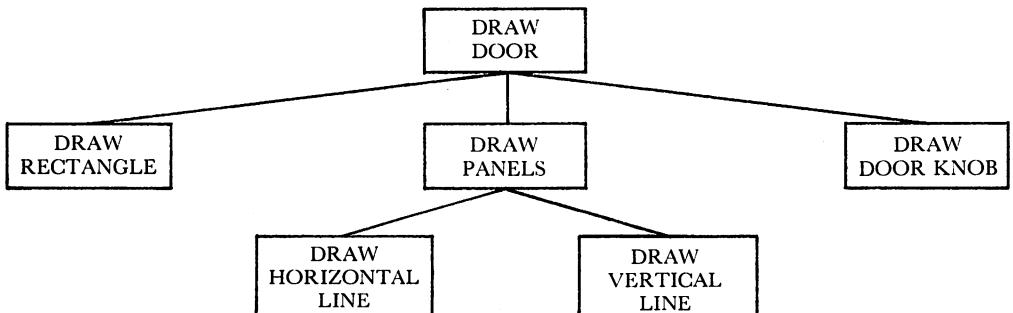
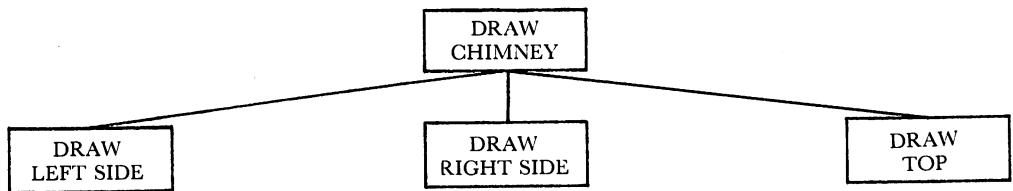
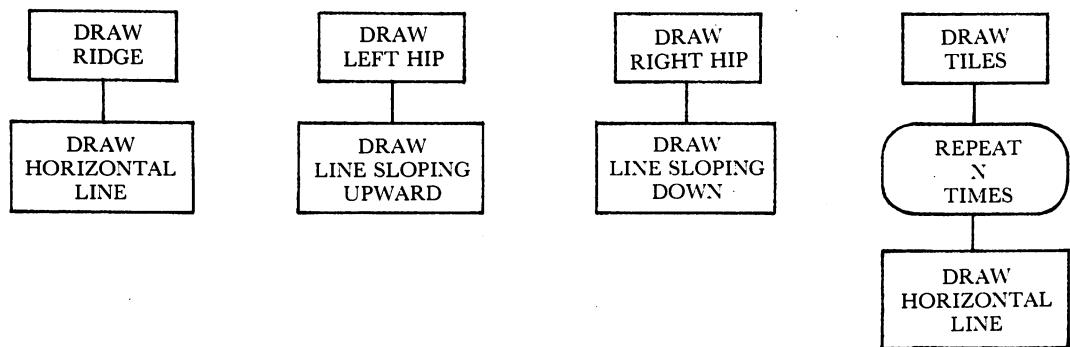
Let us now tackle a somewhat longer program than we have been accustomed to so far in the book. Our problem is to draw a front view of a house on the screen.

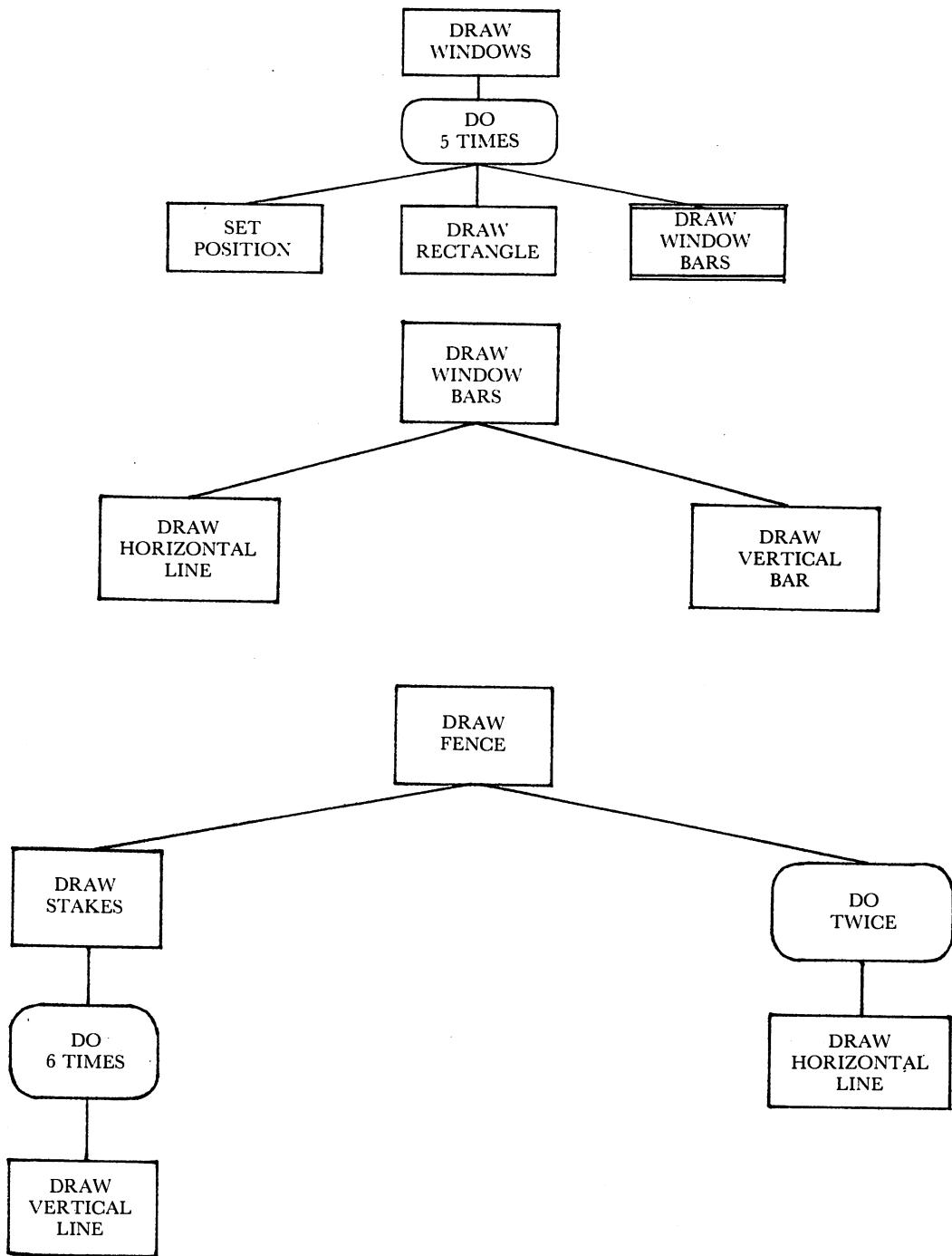
To begin with let us 'picture' to ourselves the form of the output. Since the output is a diagram let us draw the diagram.



The problem breaks down nicely into clear separate tasks. We will use a series of structure diagrams to indicate the development of a solution:







The sequence of diagrams pretty well tells the full story, and the program will follow these fairly closely, using procedures for each separate diagram. Some of the procedures may seem fairly trivial but their importance lies in the clarity and structure they give to the solution. In addition capturing an important task in the form of a procedure gives great flexibility in modifying the program at any stage to take account of new ideas.

Two developments may therefore be incorporated relatively easily:

- (1) change an existing procedure to achieve a slightly different effect e.g. door
- (2) add new procedures to extend the effect of the program e.g. draw a garage.

Although the program is a fairly long one it should be easy enough to follow from the previous development.

```
0010 // PROGRAM 121
0020 //
0030 // COMELY KATE
0040 //
0050 // TO DRAW A HOUSE
0060 //
0070 CLEAR
0080 EXEC HOUSE
0090 END
0100 //
0110 PROC HOUSE
0120 EXEC OUTLINE
0130 EXEC ROOF
0140 EXEC CHIMNEY
0150 EXEC DOOR
0160 EXEC WINDOWS
0170 EXEC FENCE
0180 ENDPROC HOUSE
0190 //
0200 PROC OUTLINE
0210 X0:=10
0220 Y0:=24 // X0 AND Y0 ARE STARTING COORDINATES
0230 LENGTH:=21
0240 HEIGHT:=17
0250 EXEC RECTANGLE(X0,Y0,LENGTH,HEIGHT)
0260 ENDPROC OUTLINE
0270 //
0280 PROC ROOF
0290 X0:=14; Y0:=3; X1:=28
0300 EXEC RIDGE(X0,X1,Y0)
0310 EXEC LEFTHIP(X0-4,X0-1,Y0+3)
0320 EXEC RIGHTHIP(X1,X1+3,Y0)
0330 EXEC TILES(X0,X1,Y0)
0340 ENDPROC ROOF
0350 //
0360 PROC RIDGE(REF X0, REF X1, REF Y0)
0370 EXEC HLINE(X0,X1,Y0)
0380 ENDPROC RIDGE
0390 //
0400 PROC TILES(X0, X1, Y0)
0410 EXEC HLINE(X0-1,X1,Y0+1)
0420 EXEC HLINE(X0-2,X1+1,Y0+2)
0430 ENDPROC TILES
0440 //
0450 PROC CHIMNEY
0460 CURSOR (21), (2)
```

```

0470 PRINT "!"
0480 CURSOR (24), (2)
0490 PRINT "!"
0500 CURSOR (22), (1)
0510 PRINT "--"
0520 //
0530 ENDPROC CHIMNEY
0540 //
0550 PROC DOOR
0560 X0:=18
0570 Y0:=24
0580 LENGTH:=4
0590 HEIGHT:=7
0600 EXEC RECTANGLE(X0,Y0,LENGTH,HEIGHT)
0610 EXEC HLINE(X0+1,X0+LENGTH-1,Y0-HEIGHT+2)
0620 EXEC VLINE(Y0-HEIGHT,Y0-1,X0+LENGTH DIV 2)
0630 CURSOR (21), (20)
0640 PRINT "*"
0650 ENDPROC DOOR
0660 //
0670 PROC WINDOWS
0680 //
0690 X0:=12
0700 Y0:=21
0710 LENGTH:=4
0720 HEIGHT:=3
0730 EXEC RECTANGLE(X0,Y0,LENGTH,HEIGHT)
0740 EXEC WINDOWBARS(X0+1,Y0,LENGTH-2,HEIGHT)
0750 X0:=24
0760 EXEC RECTANGLE(X0,Y0,LENGTH,HEIGHT)
0770 EXEC WINDOWBARS(X0+1,Y0,LENGTH-2,HEIGHT)
0780 X0:=12
0790 Y0:=11
0800 EXEC RECTANGLE(X0,Y0,LENGTH,HEIGHT)
0810 EXEC WINDOWBARS(X0+1,Y0,LENGTH-2,HEIGHT)
0820 X0:=18
0830 EXEC RECTANGLE(X0,Y0,LENGTH,HEIGHT)
0840 EXEC WINDOWBARS(X0+1,Y0,LENGTH-2,HEIGHT)
0850 X0:=24
0860 EXEC RECTANGLE(X0,Y0,LENGTH,HEIGHT)
0870 EXEC WINDOWBARS(X0+1,Y0,LENGTH-2,HEIGHT)
0880 ENDPROC WINDOWS
0890 //
0900 PROC WINDOWBARS(X, Y, L, H)
0910 EXEC HLINE(X,X+L,Y-2)
0920 CURSOR (X+1), (Y-1)
0930 PRINT "!"
0940 CURSOR (X+1), (Y-3)
0950 PRINT "!"
0960 ENDPROC WINDOWBARS
0970 //
0980 PROC FENCE
0990 X:=-2
1000 EXEC STAKES(X)
1010 X:=-31
1020 EXEC STAKES(X)
1030 EXEC HLINE(1,9,20)
1040 EXEC HLINE(1,9,22)
1050 EXEC HLINE(32,40,20)
1060 EXEC HLINE(32,40,22)
1070 ENDPROC FENCE
1080 //
1090 PROC STAKES(POSITION)

```

```

1100 FOR STAKE:=1 TO 3 DO
1110   POSITION:=POSITION+3
1120   EXEC VLINE(19,23,POSITION)
1130   NEXT STAKE
1140 ENDPROC STAKES
1150 //
1160 PROC RECTANGLE(X0, Y0, LENGTH, HEIGHT)
1170   EXEC HLINE(X0+1,X0+LENGTH-1,Y0)
1180   EXEC HLINE(X0+1,X0+LENGTH-1,Y0-HEIGHT-1)
1190   EXEC VLINE(Y0-HEIGHT,Y0-1,X0)
1200   EXEC VLINE(Y0-HEIGHT,Y0-1,X0+LENGTH)
1210 ENDPROC RECTANGLE
1220 //
1230 PROC HLINE(X0, X1, Y)
1240   FOR P:=X0 TO X1 DO
1250     CURSOR (P), (Y)
1260     PRINT "-",
1270   NEXT P
1280 ENDPROC HLINE
1290 //
1300 PROC VLINE(Y0, Y1, X)
1310   FOR P:=Y0 TO Y1 DO
1320     CURSOR (X), (P)
1330     PRINT "!"
1340   NEXT P
1350 ENDPROC VLINE
1360 //
1370 PROC LEFTHIP(X0, X1, Y0)
1380   Y:=Y0+1
1390   FOR P:=X0 TO X1 DO
1400     Y:=Y-1
1410     CURSOR (P), (Y)
1420     PRINT "^"
1430   NEXT P
1440 ENDPROC LEFTHIP
1450 //
1460 PROC RIGHTHIP(X0, X1, Y0)
1470   Y:=Y0-1
1480   FOR P:=X0 TO X1 DO
1490     Y:=Y+1
1500     CURSOR (P), (Y)
1510     PRINT "^"
1520   NEXT P
1530 ENDPROC RIGHTHIP

```

The main program consists of just two statements. The first one clears the screen and the second calls the procedure HOUSE. The procedure HOUSE in turn calls all the important procedures corresponding to level 1 of our first structure diagram on page 257.

The procedures generally follow the scheme indicated in the structure diagrams, with a few entries not indicated as procedures in the diagrams, e.g. HLINE for drawing a horizontal line and VLINE for drawing a vertical line. The procedure WINDOWS is not written as a loop as shown in the diagram because the parameters need to be set afresh for each of the five window positions (in fact not all need to be reset, some remain unchanged).

Having decided on the procedures required the main work in the program is taking care of the positions on the screen for the various components by setting the values of

the X and Y coordinates appropriately. The screen is considered to form a  $24 \times 40$  grid, i.e. 24 positions down by 40 across. The function of the command CURSOR is to position the cursor at a specified position on the screen preparatory to printing a symbol.

e.g. CURSOR (5,18) places the cursor five positions down from the top and 18 positions across from the left.

CURSOR (1,1) puts the cursor in the upper left hand corner.

CURSOR (24,40) puts the cursor in the lower right hand corner.

Finally note that it would be easy to write some extra procedures and call them from the main program or any other procedure. Several times during the above program we have called procedures from within procedures.

## Summary

Problems are best solved by successive breaking down or refinement. Programs should be similarly developed.

There are many principles of good program development. Structure diagrams play an important role.

Input and output data must be given due consideration in developing programs.

## QUESTIONS and EXERCISES

1. Why should programs be developed by successive refinement?
2. What are the main stages in developing programs?
3. Mention some important principles to keep in mind when developing programs.  
Can you think of some of your own?
4. Add procedures to the house program to
  - (a) have smoke emerging from the chimney
  - (b) build a garage
  - (c) display a car at the side of the house

### Projects

5. Write a program to play a game of noughts and crosses.
6. Write a program to play the card game Pontoon.
7. Develop and write a program to draw a diagram of a room in a house in some detail.
8. Develop a payroll-processing program for a small company which takes into account
  - (a) employee names
  - (b) departments
  - (c) hours worked at normal rate
  - (d) overtime hours
  - (e) tax
  - (f) PRSI

# 16 Great Graphics

(See Appendix 6 for notes on Commodore-64 COMAL, used in this chapter.)

## Easy Draw

In the last chapter we spent a great deal of time building a house. It gave us a valuable opportunity to see how useful procedures can be in providing a clear structure for a reasonably long program. However, we were making use of a very crude graphics facility – simply positioning the cursor and plotting a point. The number of points which can be plotted in this way is very small ( $24 \times 40$ ) and the resulting picture is not very refined or detailed. In addition it was quite tedious and troublesome to represent lines by sequences of dots. We really could do with a better, more refined and easier-to-use graphics facility. Well, some versions of COMAL provide such a system and we will enjoy using it in this chapter.

We begin by studying the so-called **Turtle Graphics** facility, which allows us to build beautiful pictures by drawing line segments in different directions. The name Turtle Graphics comes from the idea that the line segments are traced out by a moving turtle represented by a triangle on the screen. In order to set the turtle to work drawing our pictures, we must enter the **COMAL Graphics Mode** (the mode we have been using so far is called **Text Mode**). To enter graphics mode we simply enter the command

SETGRAPHIC 0

The text on the screen vanishes and is replaced by a coloured (or possibly black) rectangle, approximately  $300 \times 200$  units, surrounded by a border in a different colour. In the centre of the screen is a triangle (**turtle**) ‘facing’ up the screen. Our job is to move this triangle around to trace out diagrams.

## Straight Forward

Type in the command

FORWARD 50

Turtle movement



The turtle should move 50 units directly up the screen. Easy!

Type FORWARD 30 and see what happens.

Now type FORWARD 100. Has the turtle gone off the top of the screen?

To clear off the screen, simply type

CLEAR

To bring the turtle back to its original position at the centre of the screen, type

HOME

## **Back Down**

We can move the turtle backwards by using the command

BACK

e.g. BACK 50, BACK 30, BACK 100, etc. Try these commands and see what happens.  
Now try the following sequence of instructions:

CLEAR  
HOME  
FORWARD 80  
BACK 100

Turtle movement



Get the idea? So easy!

## **Right Now – Right Turn**

It gets a bit tedious just exploring the one-dimensional world up and down the screen, doesn't it? So let's break out! We can easily issue a command to make the turtle turn to the right – yes, you've guessed it!

RIGHT 90

This causes the triangle to turn to the right through an angle of  $90^\circ$ , while still remaining in the centre of the screen. Now type

FORWARD 100

Turtle movement



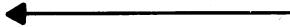
The turtle moves 100 steps across the screen to the right.

## **Attention – Left Turn**

We can turn the turtle to the left just as easily. Type in the following commands:

CLEAR  
HOME  
LEFT 90  
FORWARD 100

Turtle movement



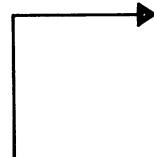
The turtle moves 100 steps across the screen to the left.

## Getting the Angle Right

Tracing out a right angle on the screen is simplicity itself! Simply type the following commands:

```
CLEAR  
HOME  
FORWARD 50  
RIGHT 90      or LEFT 90  
FORWARD 50
```

Turtle movement

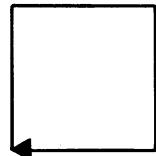


## Real Square Man!

Now that we have traced out a right angle, can we resist completing the square?

```
CLEAR  
HOME  
FORWARD 50  
RIGHT 90  
FORWARD 50  
RIGHT 90  
FORWARD 50  
RIGHT 90  
FORWARD 50
```

Turtle movement



At the end of this exercise the turtle looks a bit awkward and spoils the neatness of the figure. Can we improve on this situation? Of course! Simply type

```
HIDETURTLE
```

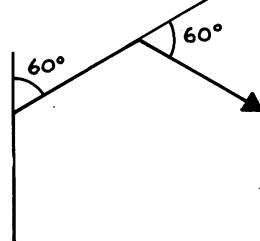
and the triangle disappears, leaving us with a nice neat square.

## The Oul' Triangle

We can draw a triangle just as easily as we can draw a square. Just trace three lines instead of four. Of course we also have to change the angle of turn. So let's try the following for a start:

```
FORWARD 50  
RIGHT 60  
FORWARD 50  
RIGHT 60  
FORWARD 50
```

Turtle movement



Oops! We don't seem to have got our angles right (I mean correct!). What were we thinking of? Well, the idea was to draw an equilateral triangle where each of the angles

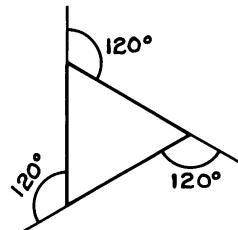
is  $60^\circ$ . But of course it is the *internal* angles which are each  $60^\circ$ . The external angles are  $120^\circ$  each. This makes sense when we consider that in completing the triangle we make a full turn, i.e. through an angle of  $360^\circ$ .

So let's try this sequence:

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

That's better!

Turtle movement

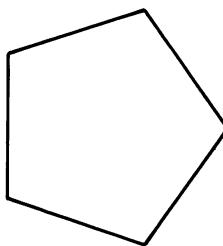


## The Pentagon

Now that we have grasped the idea of turning through  $360^\circ$  to form a complete closed figure, we give free rein to our imagination and draw lots of examples. Let's start with a pentagon. Naturally, since we are about to turn 5 times, each angle of turn will be  $360/5$ , i.e.  $72^\circ$ .

```
FORWARD 50  
RIGHT 72  
FORWARD 50  
RIGHT 72  
FORWARD 50  
RIGHT 72  
FORWARD 50  
RIGHT 72  
FORWARD 50  
HIDETURTLE
```

Turtle movement



Well, the effect is nice but the work is a bit tedious. Is there a shorter way to express these instructions? Yes — there is! Just write a program. So let's return to programming.

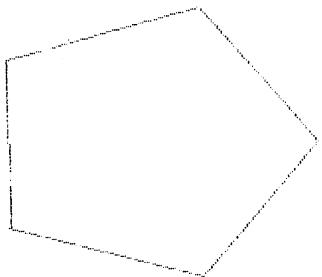
## Back to Programming

Graphics programs employ the same programming structures and techniques as other programs. Thus we think in terms of the same fundamental ingredients —

- Sequence
- Selection
- Iteration

Our program for the pentagon will consist of a simple iteration using FOR ... NEXT

```
0010 // PROGRAM 122
0020 // COMELY KATE
0030 // TO DRAW A PENTAGON
0040 //
0050 SETGRAPHIC 0
0060 FOR COUNT:=1 TO 5 DO
0070 FORWARD 50
0080 RIGHT 72
0090 ENDFOR COUNT
0100 HIDETURTLE
0110 END
```

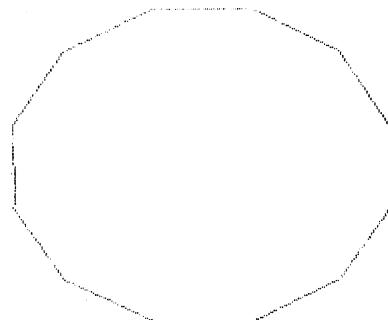


Simple, isn't it? Just move and turn five times.

## Dodecagon

It is just as easy to move and turn twelve times to produce a dodecagon (you know, of course, that a dodecagon is a 12-sided figure!)

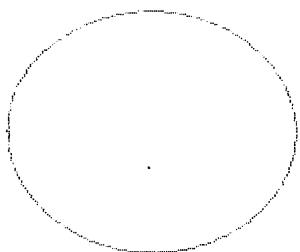
```
0010 // PROGRAM 123
0020 // COMELY KATE
0030 // TO DRAW A DODECAGON
0040 //
0050 SETGRAPHIC 0
0060 FOR COUNT:=1 TO 12 DO
0070 FORWARD .35
0080 RIGHT 30
0090 ENDFOR COUNT
0100 HIDETURTLE
0110 END
```



## Circling Round

What happens if we take a very small turn each time, say  $1^\circ$ . Let's try it and see!

```
0010 // PROGRAM 124
0020 // COMELY KATE
0030 // TO DRAW A CIRCLE(1)
0040 //
0050 SETGRAPHIC 0
0060 FOR COUNT:=1 TO 360 DO
0070 FORWARD 1
0080 RIGHT 1
0090 ENDFOR COUNT
0100 HIDETURTLE
0110 END
```



Note that we reduced our forward step considerably to avoid going off the screen. I think you will agree that we get a very good approximation to a circle.

## Flying Colours

We have been content up to this to draw our graphs in whatever colour was presented to us. In fact we can change *the colour of the line traced out on the screen*, by using the command

PENCOLOR

For example, PENCOLOR 5 sets the colour to green.

We can also change *the background screen colour* by using the command

BACKGROUND

e.g., BACKGROUND 8 sets the background colour to orange.

Finally, we can change *the colour of the border* by using the instruction

BORDER

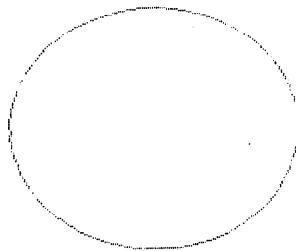
e.g., BORDER 6 sets the border colour to blue.

Sixteen colours in all are provided. They are specified by numbers from 0 to 15 as follows:

0	Black	8	Orange
1	White	9	Brown
2	Red	10	Light Red
3	Cyan	11	Dark Grey
4	Purple	12	Medium Grey
5	Green	13	Light Green
6	Blue	14	Light Blue
7	Yellow	15	Light Grey

Let us re-draw our circle, using some of those colours:

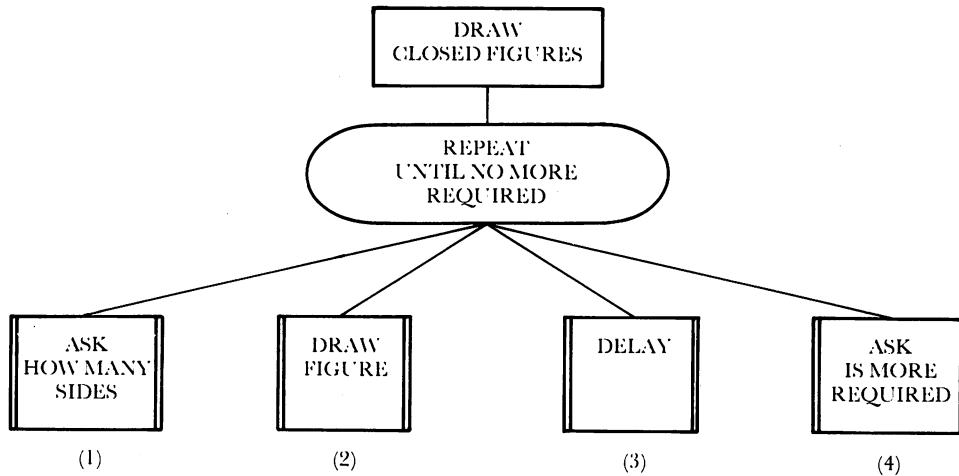
```
0010 // PROGRAM 125
0020 // COMELY KATE
0030 // TO DRAW A CIRCLE(2)
0040 //
0050 SETGRAPHIC 0
0060 PENCOLOR 5
0070 BORDER 6
0080 FOR COUNT:=1 TO 60 DO
0090 FORWARD 6
0100 RIGHT 6
0110 ENDFOR COUNT
0120 HIDETURTLE
0130 END
```



You will notice that we have only taken 60 steps around the circle this time, turning through an angle of  $6^\circ$  each time. This speeds up the drawing of the circle without really making it appear any less 'circular'. Strictly speaking we have drawn a regular 60-sided polygon, but I'm sure you are happy with the result!

## The Best Procedure

Let us now combine our knowledge of graphics with our knowledge of procedures and produce some properly-structured programs. We will construct a procedure to draw a closed figure — not simply a triangle, or a pentagon, or a circle, but a whole range of figures, on request.



The program consists of a repetition of the four stages shown in the structure diagram (see Program 126, lines 80–130). Each stage is handled by a procedure:

- Stage 1 corresponds to procedure REQUEST1 (lines 160–210)
- Stage 2 corresponds to procedure DRAWFIGURE (lines 330–410)
- Stage 3 corresponds to procedure DELAY (lines 430–460)
- Stage 4 corresponds to procedure REQUEST2 (lines 230–310)

```

0010 // PROGRAM 126
0020 // COMELY KATE
0030 // TO DRAW VARIOUS CLOSED FIGURES(1)
0040 //
0050 TIM:=2000
0060 FINISHED:=FALSE
0070 DIM ANSWER$ OF 3
0080 REPEAT
0090 REQUEST1(N£)
0100 DRAWFIGURE(N£)
0110 DELAY(TIM)
0120 REQUEST2(FINISHED)
0130 UNTIL FINISHED
0140 END
0150 //
0160 PROC REQUEST1(REF N£)
0170 REPEAT
0180 PRINT
0190 INPUT "HOW MANY SIDES SHOULD THE FIGURE HAVE (AT LEAST 3 PLEASE) ": N£
0200 UNTIL N£>=3
0210 ENDPROC REQUEST1
0220 //
0230 PROC REQUEST2(REF FINISHED)
0240 REPEAT
0250 SETTEXT
0260 CLEAR // See Appendix 6, p. 348
0270 PRINT
0280 INPUT "MORE (Y/N) ? ": ANSWER$
0290 UNTIL ANSWER$(1)="Y" OR ANSWER$(1)="N"
0300 IF ANSWER$(1)="N" THEN FINISHED:=TRUE
0310 ENDPROC REQUEST2

```

```

0320 //
0330 PROC DRAWFIGURE (NUMBEROFSIDES£)
0340   SETGRAPHIC 0
0350   ANGLE:=360/NUMBEROFSIDES£
0360   DISTANCE:=ANGLE
0370   FOR COUNT£:=1 TO NUMBEROFSIDES£ DO
0380     FORWARD DISTANCE
0390     RIGHT ANGLE
0400   ENDFOR COUNT£
0410 ENDPROC DRAWFIGURE
0420 //
0430 PROC DELAY(PERIOD)
0440   FOR D:=1 TO PERIOD DO
0450   ENDFOR D
0460 ENDPROC DELAY

```

### Notes

In the procedure REQUEST1, the request for input is repeated if a number of sides less than 3 is entered in response to the question "how many sides should the figure have (at least 3 please)". The repetition goes on until a number greater than or equal to 3 is entered. This number is then passed back to the calling program via the reference parameter N#, where it is passed in turn to the procedure DRAWFIGURE through the value parameter NUMBEROFSIDES#.

A similar method of dealing with unacceptable input is adopted in the procedure REQUEST2, where the user is repeatedly asked to answer the question "MORE (Y/N)?" until he enters a word beginning with Y or N, e.g., YES, YEAH, NO, NOPE, etc. If the answer begins with N, the logical variable FINISHED is given the value TRUE. This is passed back to the main program and causes termination of the program. Otherwise FINISHED remains FALSE and the program continues.

The procedure DELAY is easy to understand. It simply causes a FOR loop to be executed for a number of times given by the parameter PERIOD (which has the value 2000 in this program — see line 50). Nothing is done during the loop except waste time. It has the effect, however, of holding the computer back from the next task until you can see what is on the screen.

In the procedure DRAWFIGURE the graphics mode is entered with the command SETGRAPHIC 0 on line 340. The angle of turn is then calculated by dividing 360° by the number of sides. The distance of movement corresponding to the length of each side of the polygon is adjusted on line 360 so as to give a reasonably-sized figure which won't go off the screen. It is found that having the length of the side equal to the size of the angle of turn works very nicely.

**N.B.** The listings in this chapter all display the £ sign instead of #, e.g. N£ should be read as N#.

## Recursion

You remember the concept of recursion — a procedure which calls itself? Of course you do! Well, an interesting way to write the above program is to have the procedure DRAWFIGURE call itself, i.e. to make it recur as follows:

```
0010 // PROGRAM 127
0020 // COMELY KATE
0030 // TO DRAW VARIOUS CLOSED FIGURES(2)
0040 //
0050 REQUEST1(N£)
0060 ANGLE:=360/N£
0070 DISTANCE:=ANGLE
0080 SETGRAPHIC 0
0090 DRAWFIGURE
0100 END
0110 //
0120 PROC REQUEST1(REF N£)
0130 REPEAT
0140 PRINT
0150 INPUT "HOW MANY SIDES SHOULD THE FIGURE HAVE (AT LEAST 3 PLEASE) ": N£
0160 UNTIL N£>=3
0170 ENDPROC REQUEST1
0180 //
0190 PROC DRAWFIGURE
0200 FORWARD DISTANCE
0210 RIGHT ANGLE
0220 DRAWFIGURE
0230 ENDPROC DRAWFIGURE
```

We have slimmed down the program by taking away the procedures REQUEST2 and DELAY. Also for simplicity we have dropped the parameter NUMBEROFSIDES# and have calculated DISTANCE# and ANGLE# outside the procedure for efficiency. The resulting procedure DRAWFIGURE is so beautifully simple, isn't it? — and indeed the whole program. However, one thing is wrong! The procedure keeps calling itself forever and so we have to manually intervene to stop the program, by pressing the STOP key or something similar.

Remember that for a properly-controlled recursive procedure we need two basic elements:

- (1) a base part which serves to terminate the procedure
- (2) a recursive part where the procedure calls itself.

Let us re-write the procedure DRAWFIGURE and provide a proper termination point. To do this we put back the parameter NUMBEROFSIDES# and use it this time as a counter to count down to zero. When the counter reaches zero the recursion will cease. Our new version of the procedure will therefore read as follows:

```
PROC DRAWFIGURE (NUMBEROFSIDES#)
  IF NUMBEROFSIDES# > 0 THEN
    FORWARD DISTANCE#
    RIGHT ANGLE#
    NUMBEROFSIDES# := NUMBEROFSIDES# -1
    DRAWFIGURE (NUMBEROFSIDES#)
  ENDIF
ENDPROC DRAWFIGURE
```

Note that the procedure is effectively executed only if NUMBEROFSIDES# is greater than zero. During the execution the turtle moves forward a certain distance, then turns right through a certain angle. The number of sides is decreased by 1 and the procedure is called again, with this reduced number of sides as the value of its parameter. Eventually NUMBEROFSIDES# will be reduced to zero and the procedure will stop.

The following sets the above procedure into a full program:

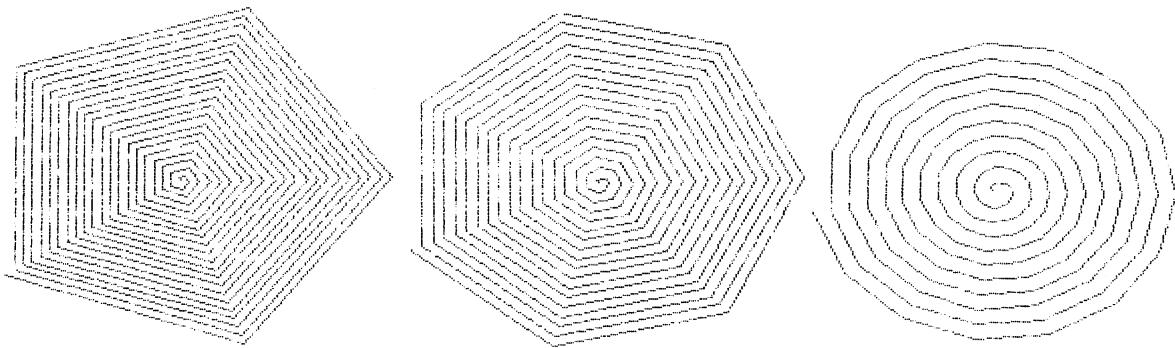
```
0010 // PROGRAM 12B
0020 // COMELY KATE
0030 // TO DRAW VARIOUS CLOSED FIGURES(3)
0040 //
0050 TIM:=2000
0060 DIM ANSWER$ OF 3
0065 FINISHED:=FALSE
0070 REPEAT
0080 REQUEST1(N£)
0090 ANGLE:=360/N£
0100 DISTANCE:=ANGLE
0110 SETGRAPHIC 0
0120 DRAWFIGURE(N£)
0130 DELAY(TIM)
0140 REQUEST2(FINISHED)
0150 UNTIL FINISHED
0160 END
0170 //
0180 PROC REQUEST1(REF N£)
0190 REPEAT
0200 PRINT
0210 INPUT "HOW MANY SIDES SHOULD THE FIGURE HAVE (AT LEAST 3 PLEASE) ":" N£
0220 UNTIL N£>=3
0230 ENDPROC REQUEST1
0240 //
0250 PROC REQUEST2(REF FINISHED)
0260 FINISHED:=FALSE
0270 REPEAT
0280 SETTEXT
0290 CLEAR // See Appendix 6, p. 348
0300 PRINT
0310 INPUT "MORE (Y/N) ? ":" ANSWER$
0320 UNTIL ANSWER$(1)="Y" OR ANSWER$(1)="N"
0330 IF ANSWER$(1)="N" THEN FINISHED:=TRUE
0340 ENDPROC REQUEST2
0350 //
0360 PROC DRAWFIGURE(NUMBEROFSIDES£)
0370 IF NUMBEROFSIDES£>0 THEN
0380 FORWARD DISTANCE
0390 RIGHT ANGLE
0400 NUMBEROFSIDES£:=NUMBEROFSIDES£-1
0410 DRAWFIGURE(NUMBEROFSIDES£)
0420 ENDIF
0430 ENDPROC DRAWFIGURE
0440 //
0450 PROC DELAY(PERIOD)
0460 FOR D:=1 TO PERIOD DO
0470 ENDFOR D
0480 ENDPROC DELAY
```

## Spiral

A further very simple modification to the recursive procedure allows us to draw some very nice spirals. The idea is to keep increasing the amount of forward movement for each call of DRAWFIGURE so that the figure being drawn does not close on itself but spirals outward. To achieve this we use DISTANCE as a parameter (Program 129, lines 140, 380, 420). Note carefully that in the recursive call (line 420) the actual parameter is DISTANCE + INC which has the effect of passing an increased value to the formal parameter DISTANCE on line 380. This of course is further increased on the next call. The value of INC has been set to 5 divided by the number of sides, on line 110. You should experiment with this value yourself so as to achieve different-sized spirals.

The parameter COUNT# is used to control the recursion. It is arbitrarily set to 100 on line 130. Change this also if you wish.

```
0010 // PROGRAM 129
0020 // COMELY KATE
0030 // TO DRAW SOME SPIRALS
0040 //
0050 DIM ANSWER$ OF 3
0060 TIM:=2000
0070 REPEAT
0080 REQUEST1(N£)
0090 ANGLE:=360/N£
0100 DISTANCE:=1
0110 INC:=5/N£
0120 SETGRAPHIC 0
0130 COUNT£:=100
0140 DRAWFIGURE(COUNT£,DISTANCE)
0150 DELAY(TIM)
0160 REQUEST2(FINISHED)
0170 UNTIL FINISHED
0180 END
0190 //
0200 PROC REQUEST1(REF N£)
0210 REPEAT
0220 PRINT
0230 INPUT "HOW MANY SIDES SHOULD THE FIGURE HAVE (AT LEAST 3 PLEASE) ": N£
0240 UNTIL N£>=3
0250 ENDPROC REQUEST1
0260 //
0270 PROC REQUEST2(REF FINISHED)
0280 FINISHED:=FALSE
0290 REPEAT
0300 SETTEXT
0310 CLEAR // See Appendix 6, p. 348
0320 PRINT
0330 INPUT "MORE (Y/N) ? ": ANSWER$
0340 UNTIL ANSWER$(1)="Y" OR ANSWER$(1)="N"
0350 IF ANSWER$(1)="N" THEN FINISHED:=TRUE
0360 ENDPROC REQUEST2
0370 //
0380 PROC DRAWFIGURE(COUNT£,DISTANCE)
0390 IF COUNT£>0 THEN
0400 FORWARD DISTANCE
0410 RIGHT ANGLE
0420 DRAWFIGURE(COUNT£-1,DISTANCE+INC)
0430 ENDIF
0440 ENDPROC DRAWFIGURE
0450 //
0460 PROC DELAY(PERIOD)
0470 FOR D:=1 TO PERIOD DO
0480 ENDFOR D
0490 ENDPROC DELAY
```



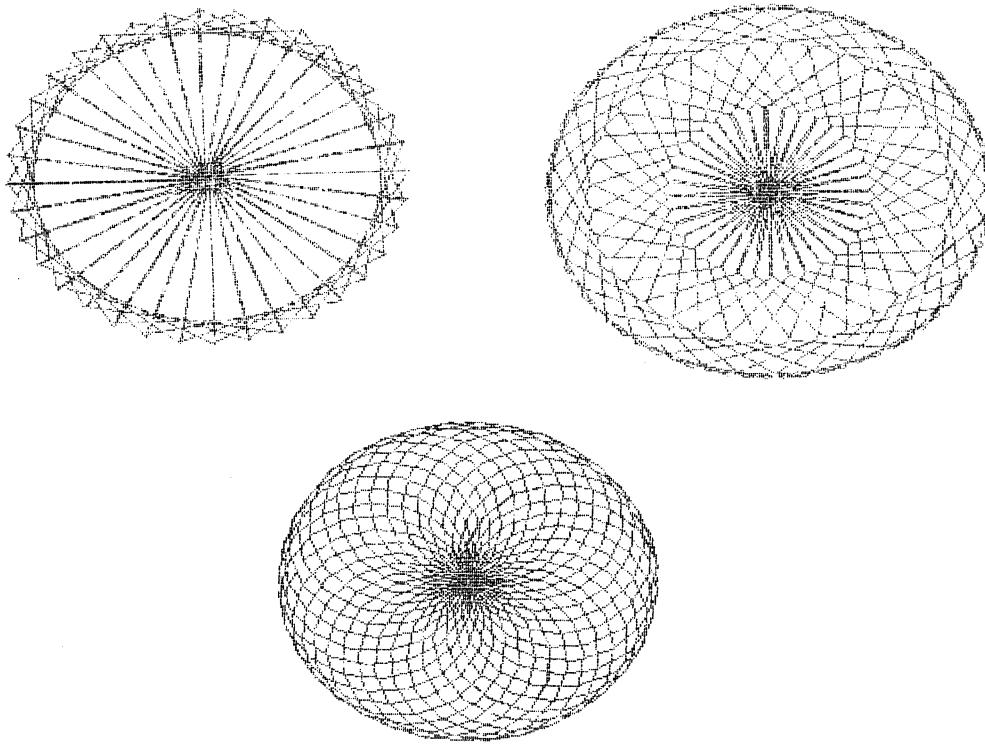
## Pretty Patterns

It is worth while investing a little more time and effort in this simple procedure DRAWFIGURE. This time let us draw a closed figure, turn through a small angle (say 10°), draw the figure again and keep doing this to see what we get. The following program does the job.

```

0010 // PROGRAM 130
0020 // COMELY KATE
0030 // TO ROTATE VARIOUS CLOSED FIGURES
0040 //
0050 ANGLE2:=10
0060 REQUEST1 (N£)
0070 ANGLE1:=360/N£
0080 DISTANCE:=ANGLE1
0090 SETGRAPHIC 0
0100 ROTATE (N£,ANGLE2)
0110 END
0120 //
0130 PROC REQUEST1 (REF N£)
0140 REPEAT
0150 PRINT
0160 INPUT "HOW MANY SIDES SHOULD THE FIGURE HAVE (AT LEAST 3 PLEASE) ": N£
0170 UNTIL N£>=3
0180 ENDPROC REQUEST1
0190 //
0200 PROC DRAWFIGURE (COUNT£)
0210 IF COUNT£>0 THEN
0220 FORWARD DISTANCE
0230 RIGHT ANGLE1
0240 DRAWFIGURE (COUNT£-1)
0250 ENDIF
0260 ENDPROC DRAWFIGURE
0270 //
0280 PROC ROTATE (N£,ANGLE)
0290 PENCOLOR RND(1,15)
0300 DRAWFIGURE (N£)
0310 RIGHT ANGLE
0320 ROTATE (N£,ANGLE)
0330 ENDPROC ROTATE

```



Two angles are involved:

- (1) the angle required to draw the closed figure, i.e. ANGLE1, calculated to be  $360^\circ/N\#$  as usual (line 70)
- (2) the angle through which the figure is rotated, i.e. ANGLE2, arbitrarily set to  $10^\circ$  (line 50).

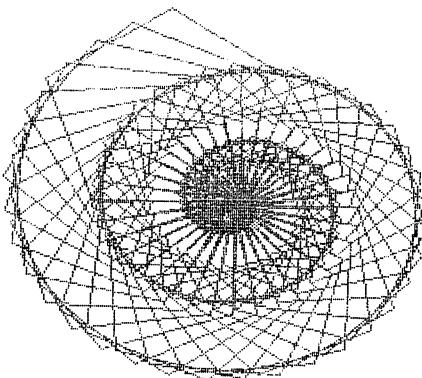
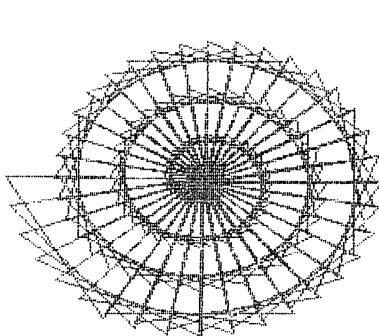
The procedure DRAWFIGURE is the familiar recursive procedure with COUNT# counting down from the number of sides N# to 0. This procedure is called for the first time (line 300) not from the main program but from the new procedure ROTATE.

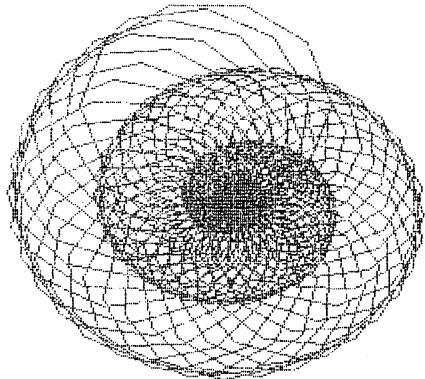
The procedure ROTATE (lines 280–330) is also a recursive procedure whose job is essentially to call the procedure DRAWFIGURE to draw a polygon and then to turn through an angle preparatory to another drawing of the figure. One other feature is included (line 290), namely the colour is randomly changed for each call.

## Spiralling Patterns

Once again some minor modifications allow us to achieve brilliant new effects. We will draw a figure, rotate, draw an enlarged figure, rotate, and so on. All we have to do is put another parameter DISTANCE (for length of side) into the procedure ROTATE and increment it each time the procedure calls itself (Program 131, line 330). DISTANCE is given a small value 1 to start with (line 90) and the figure gradually spirals outwards.

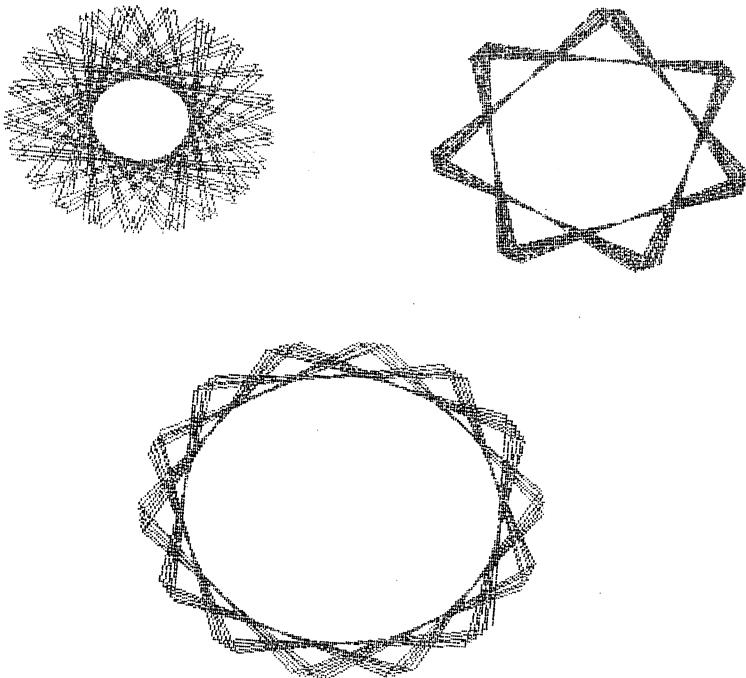
```
0010 // PROGRAM 131
0020 // COMELY KATE
0030 // TO DRAW VARIOUS SPIRALS
0040 //
0050 ANGLE2:=10
0060 INC:=1
0070 REQUEST1 (NE)
0080 ANGLE1:=360/NE
0090 DISTANCE:=1
0100 SETGRAPHIC 0
0110 ROTATE (NE,ANGLE2,DISTANCE)
0120 END
0130 //
0140 PROC REQUEST1 (REF NE)
0150 REPEAT
0160   PRINT
0170   INPUT "HOW MANY SIDES SHOULD THE FIGURE HAVE (AT LEAST 3 PLEASE) ": NE
0180 UNTIL NE>=3
0190 ENDPROC REQUEST1
0200 //
0210 PROC DRAWFIGURE (COUNT£,DISTANCE)
0220 IF COUNT£>0 THEN
0230   FORWARD DISTANCE
0240   RIGHT ANGLE1
0250   DRAWFIGURE (COUNT£-1,DISTANCE)
0260 ENDIF
0270 ENDPROC DRAWFIGURE
0280 //
0290 PROC ROTATE (NE,ANGLE,DISTANCE)
0300 PENCOLOR RND(1,15)
0310 DRAWFIGURE (NE,DISTANCE)
0320 RIGHT ANGLE
0330 ROTATE (NE,ANGLE,DISTANCE+INC)
0340 ENDPROC ROTATE
```





'What beautiful patterns!' you are saying to yourself — 'and so easy!' The simplicity is based on the geometrical regularity of the figures, of course. As a final geometrical indulgence, let us create some figures using a very simple 'figure' — a line segment!

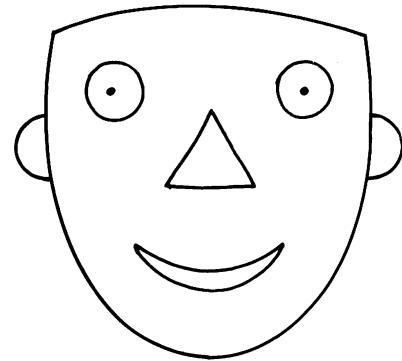
```
0010 // PROGRAM 132
0020 // COMELY KATE
0030 // TO STITCH A PATTERN
0040 //
0050 DIM ANSWER$ OF 3
0060 TIM:=2000
0070 REPEAT
0080 INPUT "ENTER NUMBER OF LINES,ANGLE OF TURN,LENGTH OF LINE ": N,A,L
0090 SETGRAPHIC 0
0110 PENCOLOR 7
0120 DRAW(N,A,L)
0130 DELAY(TIM)
0140 SETTEXT
0150 PRINT
0160 INPUT "MORE(Y/N) ? ": ANSWER$
0170 UNTIL ANSWER$(1)="N"
0180 END
0190 //
0200 PROC DRAW(N,ANGLE,LENGTH)
0210 FOR COUNT:=1 TO N DO
0220   FORWARD LENGTH
0230   RIGHT ANGLE
0240 ENDFOR COUNT
0245 HIDE TURTLE
0250 ENDPROC DRAW
0260 //
0270 PROC DELAY(PERIOD)
0280 FOR D:=1 TO PERIOD DO
0290 ENDFOR D
0300 ENDPROC DELAY
```



The important procedure here is the procedure DRAW (lines 200–250). As you can see, it simply causes the turtle to move forward a certain length, turn through a certain angle and repeat this N times. The length of the line segment, the angle of turn and the number of repetitions can all be specified by the user (line 80).

## Face the face

As we have already remarked, the ease of drawing patterns like the above is essentially based on their geometrical nature. Let us now turn to something a little less regular — the human face. It is not our intention to do Leonardo Da Vinci out of a job! We will content ourselves with something a little less elegant than the Mona Lisa — say something like we see displayed in the rough drawing shown here.



Apart from the rugged triangular nose everything else is composed of arcs of circles. This suggests that we should begin by composing a procedure to draw an arc of a circle. Obviously this will be similar to the procedure for drawing a full circle. However, we need to control the length of the arc and its degree of 'roundness'. We will make use of three parameters - length of step, number of steps and angle of turn. The following procedure should do the job:

```

PROC DRAWARC (LENGTH, NUMOFSTEPS, ANGLE)
  IF NUMOFSTEPS > 0 THEN
    FORWARD LENGTH
    RIGHT ANGLE
    DRAWARC (LENGTH, NUMOFSTEPS-1, ANGLE)
  ENDIF
ENDPROC DRAWARC

```

We may test this procedure by calling it from a simple program as follows:

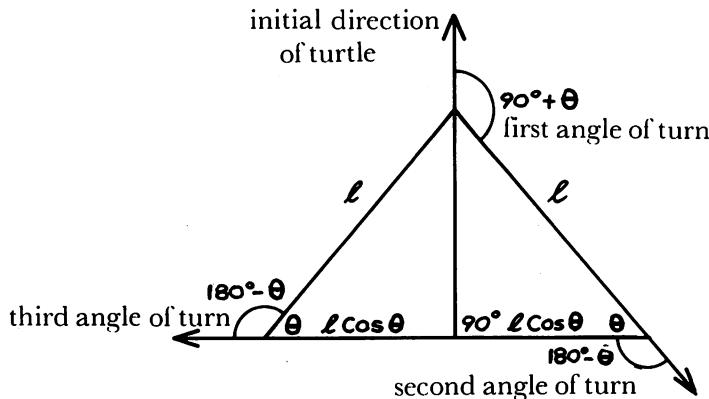
```

10 INPUT "LENGTH OF STEP, NUMBER OF STEPS,
ANGLE OF TURN": L, N, A
20 SETGRAPHIC 0
30 DRAWARC (L, N, A)
40 END

```

You should type in this little program, including the above procedure, and run it a few times. If you give the appropriate parameter values you will notice that the DRAWARC procedure can also be used to draw a complete circle (useful for eyes, you see!).

Since nearly all aspects of the face are composed of arcs, we now have the basis for most of our drawing. The other procedure we need is a triangle. We have already drawn equilateral triangles but perhaps an isosceles triangle would look better here. This is a little more difficult, but the following diagram should indicate the appropriate distances and amounts of turn required:



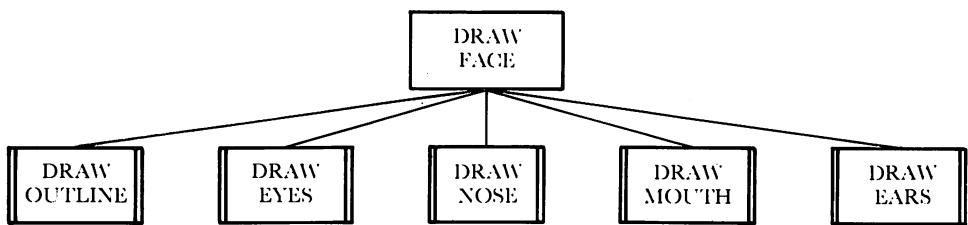
The following procedure should be easy to follow: Note that we need to convert the angle from degrees to radians before getting the cosine. This we do by multiplying the number of degrees by  $\pi/180$ .

```

PROC DRAWTRIANGLE (LENGTH, ANGLE)
  PI := 3.1416
  RIGHT 90 + ANGLE
  FORWARD LENGTH
  RIGHT 180 - ANGLE
  FORWARD 2 * LENGTH * COS (PI * ANGLE/180)
  RIGHT 180 - ANGLE
  FORWARD LENGTH
ENDPROC DRAWTRIANGLE

```

Let us now set out the full plan for drawing a face, in the form of a structure diagram:



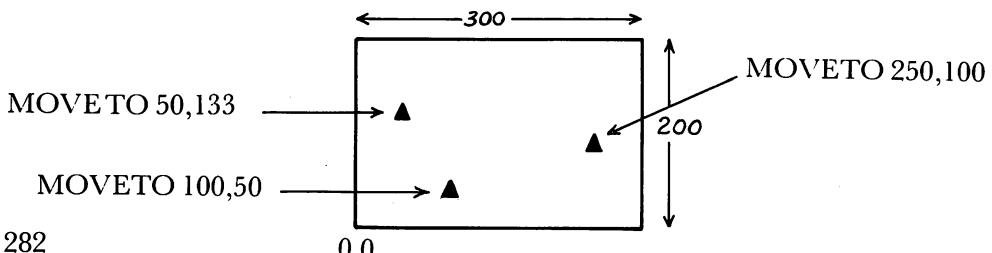
One further detail needs to be attended to before we launch into the full program. It is important to position the turtle correctly for the start of each section of the drawing. There are really two ingredients involved:

- (1) the actual position of the turtle
- (2) the direction in which the turtle will move on receiving the first FORWARD command.

We know that the turtle can be positioned at the centre of the screen by using the HOME command. To position it at some other point of the screen we use the command

MOVETO

e.g., MOVETO 100, 50 will place the turtle 100 units across and 50 units up from the bottom left hand corner. The bottom left hand corner has the coordinates 0, 0 and the full screen may be thought of as a grid approximately 300 units across by 200 units up.



Following the HOME or MOVETO commands, the turtle's normal starting direction is directly up the screen. We can change this with the command

### SETHEADING

Thus SETHEADING 60 causes the turtle to face in a direction 60° clockwise from north.

SETHEADING 220° causes the turtle to face in a direction 220° clockwise from north.

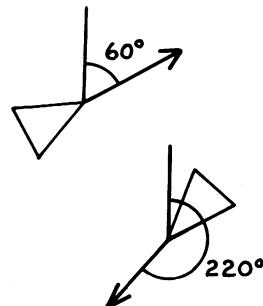
The pair of commands

```
MOVETO 80,120  
SETHEADING 30
```

cause the turtle to move to the point (80,120) and face in a direction 30° clockwise from north.

And now to the program (at last!):

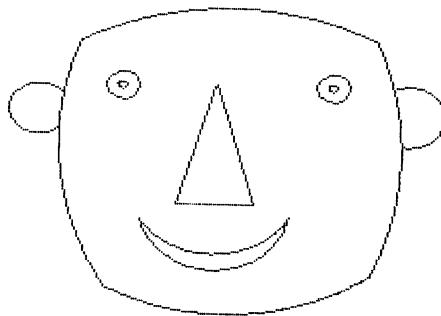
```
0010 // PROGRAM 133  
0020 // COMELY KATE  
0030 // TO DRAW A FACE  
0040 //  
0050 SETGRAPHIC 0  
0070 DRAWFACE  
0080 HIDETURTLE  
0090 END  
0100 //  
0110 PROC DRAWFACE  
0120 DRAWOUTLINE  
0130 DRAWEYES  
0140 DRAWNOSE  
0150 DRAWMOUTH  
0160 DRAWEARS  
0170 ENDPROC DRAWFACE  
0180 //  
0190 PROC DRAWOUTLINE  
0200 MOVETO 80,175  
0210 SETHEADING 60  
0220 DRAWARC(5,32,2)  
0230 RIGHT 40  
0240 DRAWARC(5,30,1.8)  
0250 RIGHT 30  
0260 DRAWARC(4,25,1.8)  
0270 RIGHT 30  
0280 DRAWARC(5,31,1.7)  
0290 ENDPROC DRAWOUTLINE  
0300 //  
0310 PROC DRAWEYES  
0320 MOVETO 100,150  
0330 DRAWARC(4,10,36)  
0340 MOVETO 105,150  
0350 DRAWARC(1,10,36)  
0360 MOVETO 200,150  
0370 DRAWARC(4,10,36)  
0380 MOVETO 205,150  
0390 DRAWARC(1,10,36)  
0400 ENDPROC DRAWEYES
```



```

0410 //
0420 PROC DRAWNOSE
0430 MOVETO 155,150
0440 SETHEADING 0
0460 DRAWTRIANGLE(50,75)
0470 ENDPROC DRAWNOSE
0480 //
0490 PROC DRAWMOUTH
0500 MOVETO 185,80
0510 SETHEADING 260
0520 DRAWARC(5,12,2)
0530 MOVETO 185,80
0540 SETHEADING 240
0550 DRAWARC(1,60,1)
0560 ENDPROC DRAWMOUTH
0570 //
0580 PROC DRAWEARS.
0590 MOVETO 73,130
0600 SETHEADING 220
0610 DRAWARC(2,50,6)
0620 MOVETO 237,150
0630 SETHEADING 20
0640 DRAWARC(2,50,6)
0650 ENDPROC DRAWEARS
0660 //
0670 PROC DRAWARC(LENGTH,NUMOFSSTEPS,ANGLE)
0680 IF NUMOFSSTEPS>0 THEN
0690 FORWARD LENGTH
0700 RIGHT ANGLE
0710 DRAWARC(LENGTH,NUMOFSSTEPS-1,ANGLE)
0720 ENDIF
0730 ENDPROC DRAWARC
0740 //
0750 PROC DRAWTRIANGLE(LENGTH,ANGLE)
0760 PI:=3.1416
0770 RIGHT 90+ANGLE
0780 FORWARD LENGTH
0790 RIGHT 180-ANGLE
0800 FORWARD 2*LENGTH*COS(PI*ANGLE/180)
0810 RIGHT 180-ANGLE
0820 FORWARD LENGTH
0830 ENDPROC DRAWTRIANGLE

```



The main program is incredibly simple! — really only two instructions,  
SETGRAPHIC 0 and DRAWFACE.

The DRAWFACE procedure is also very simple - it follows exactly the structure diagram on page 282. The other procedures are mostly concerned with moving to the correct starting position and setting the correct heading for each section of the face. Every procedure except the DRAWNOSE procedure calls the DRAWARC procedure. A little bit of experimenting was needed to get all the positions and headings and sizes of arc correct, but the end result is not too bad, eh?

Note that the DRAWTRIANGLE procedure uses the values 50 and 75 for the length of the equal sides and the base angles respectively (see line 460). Changing these values will give you noses which are shorter, longer, thinner, fatter, as you will!

## In and Out

Returning now to regular geometrical figures let us see if we can breathe a bit of life into our diagrams. We will attempt to draw a set of squares decreasing in size, gradually rub them out, draw a set increasing in size, rub these out, and so on.

How can we rub out without switching from graphics to text? As usual the answer is easy — we simply plot the figure over again, *in the background colour*.

```
0010 // PROGRAM 134
0020 // COMELY KATE
0030 // TO DRAW AN IMPLODING, EXPLODING SQUARE
0040 //
0050 START:=180
0060 FINISH:=10
0070 INC:=-10
0080 SETGRAPHIC 0
0090 BACKGROUND 12
0100 SQUAREAWAY
0110 END
0120 //
0130 PROC SQUARE(LENGTH)
0140 FOR SIDE:=1 TO 4 DO
0150 FORWARD LENGTH
0160 RIGHT 90
0170 ENDFOR SIDE
0180 ENDPROC SQUARE
0190 //
0200 PROC DRAWANDERASE(LENGTH)
0210 MOVETO 150-LENGTH/2,100-LENGTH/2
0220 SQUARE(LENGTH)
0230 ENDPROC DRAWANDERASE
0240 //
0250 PROC SQUAREAWAY
0260 PENCOLOR 7
0270 FOR LENGTH:=START TO FINISH STEP INC DO
0280 DRAWANDERASE(LENGTH)
0290 ENDFOR LENGTH
0300 TEMP:=START
0310 START:=FINISH
0320 FINISH:=TEMP
0330 INC:=-INC
0340 PENCOLOR 12
0350 FOR LENGTH:=START TO FINISH STEP INC DO
0360 DRAWANDERASE(LENGTH)
0370 ENDFOR LENGTH
0380 SQUAREAWAY
0390 ENDPROC SQUAREAWAY
```

The squares start at size  $180 \times 180$  (line 50) and finish at size  $10 \times 10$  (line 60). The length of the side decreases by 10 for each square (line 70).

The important procedure here is **SQUAREAWAY**. It begins by setting the pencolour to 7 (line 260). Then, using a simple FOR loop (lines 270–290), squares are drawn with sides of length START to FINISH by steps of size INC.

START and FINISH are then swapped and the increment is changed to the negative of its previous value (lines 300–330). This has the effect of changing from decreasing to increasing squares, or vice versa. The pencolour is changed to 12, which was set as the background colour on line 90. Squares are now drawn in this background colour (lines 350–370). This achieves the ‘rubbing out’ effect. The whole job is repeated endlessly by calling the procedure **SQUAREAWAY** recursively (line 380).

The drawing is actually done by the two procedures **DRAWANDERASE** and **SQUARE**. **DRAWANDERASE** places the turtle in its correct starting position for each square, i.e. half the length of a side left and down from the centre of the screen, by using **MOVETO** on line 210. The **SQUARE** procedure is pretty obvious.

Run the program and watch the effect.

## Move Along

That was all very nice, but a little bit slow! Let’s see if we can move along a little faster. To do this we will simplify matters and set ourselves the task of moving a point across the screen. Up to this stage we have only drawn line segments. We can, however, plot a point by using the command

### PLOT

e.g. PLOT 50, 100 plots a point at the position (50,100) in the current pen colour. To achieve the effect of movement we plot a point, rub it out, plot it a little further on, rub it out, etc. And of course you remember how to rub out — plot in the background colour.

```
0010 // PROGRAM 135
0020 // COMELY KATE
0030 // TO DRAW A MOVING POINT
0040 //
0050 TIM:=10
0060 Y:=100
0070 SETGRAPHIC 0
0080 BACKGROUND 0
0090 MOVE (Y)
0100 END
0110 //
0120 PROC MOVE(Y)
0130 HIDETURTLE
0140 FOR X:=1 TO 300 DO
0150   PENCOLOR 7
0160   PLOT X,Y
0170   DELAY(TIM)
0180   PENCOLOR 0
0190   PLOT X,Y
0200 ENDFOR X
0210 ENDPROC MOVE
```

```
0220 //  
0230 PROC DELAY(TIM)  
0240 FOR COUNT:=1 TO TIM DO  
0250 ENDFOR COUNT  
0260 ENDPROC DELAY
```

The main program sets a short delay period (line 50), establishes the Y coordinate for the point (about the middle of the screen), switches to graphics mode and sets the background colour to 0 (black). The procedure MOVE is then called (line 90).

The procedure MOVE (lines 120-210) begins by removing the turtle from the screen through the command

#### HIDETURTLE

Then a point is plotted in yellow (pencolour = 7) at X, Y where X changes from 1 to 300, i.e. right across the screen, while Y remains constant at 100. After each plot in yellow, there is a delay (line 170) and then the point is rubbed out by plotting it again in the background colour (lines 180-190).

The net effect is to move the point across the centre of the screen. Run the program and see.

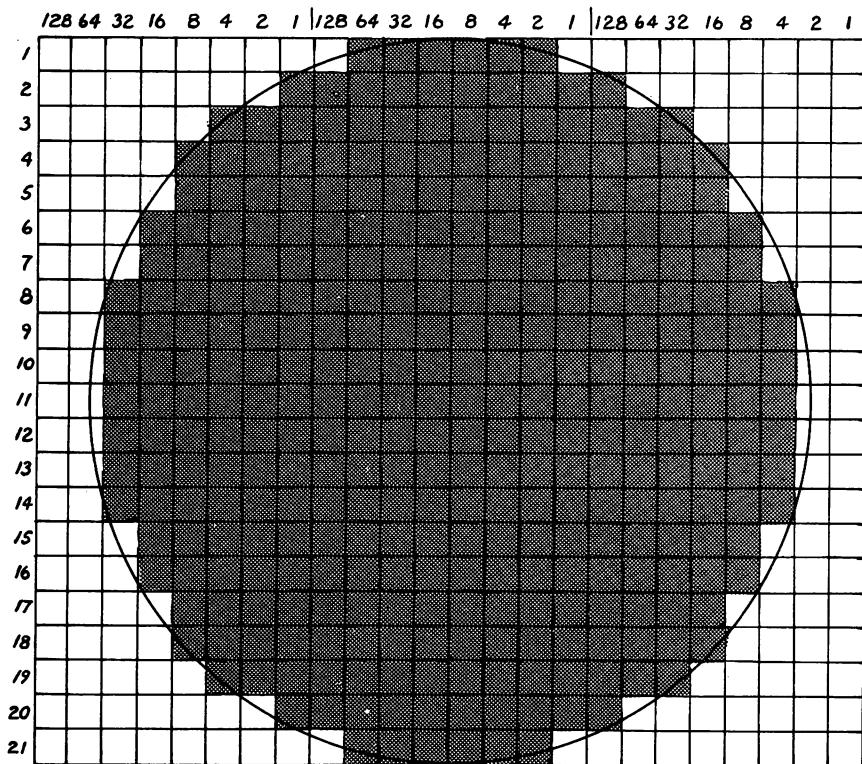
## Spritley now!

Well, we've got some movement all right, but the point is a bit small and the motion is a bit jerky and we are probably not all that satisfied. It would be much better if we had something big enough to see clearly and if we could get it to move smoothly. Once again, COMAL has the answer — **sprites**!

Sprites? 'What on earth is a sprite?' I hear you say! A sprite is an object capable of movement. How do we create one of these sprites and how do we get it to move? The answer to the second part of the question is easy, as we shall shortly demonstrate. We simply plot the sprite at different points (X, Y positions) on the screen. We don't have to worry about rubbing the sprite out, plotting it again, rubbing out, etc. COMAL takes care of all that for us. We simply have to specify the positions on the screen where the sprite is to appear. It's that easy!

To create a sprite needs a little bit of care. Essentially the shape required must be plotted on a grid and a scheme of numbers must be used to tell COMAL about the shape. This will all become clear through the following discussion.

The grid on which we must work is a  $21 \times 24$  grid. Suppose we wish to specify a ball as the shape of our sprite. This is done as follows:



First use a compass to draw a circle. Our circle fills the grid. We could have made it a good deal smaller. The choice is ours.

Then fill in the squares on the grid enclosed by the circle. In some cases the circle does not quite fully enclose a small square. In such a case we usually fill in the square if at least half of it is enclosed.

Now comes the tedious part! Each of the 21 rows of the grid is considered to be made of three sets of 8 small squares. Within each set, the little squares have numbers 1, 2, 4, 8, 16, 32, 64, 128 associated with them. These values are written above the grid for convenience. (It is no accident that these numbers are all powers of 2. The set of 8 small squares is in fact the equivalent of 1 byte).

In order to describe the ball, we add up the numbers within each set of 8 corresponding to the filled squares. For example, in the first row the first 8 squares are all blank, so the value is 0. The next eight squares in the same row give a value of

$$64 + 32 + 16 + 8 + 4 + 2 = 126$$

The next eight squares in that row are blank, giving 0.

$\therefore$  The first row is represented by the three numbers 0, 126, 0.

As another example, take the 7th row.

The first set of 8 little squares gives  $16 + 8 + 4 + 2 + 1 = 31$ .

The second set gives  $128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255$ .

The third set gives  $128 + 64 + 32 + 16 + 8 = 248$ .

So the 7th row is represented by 31, 255, 248.

The full diagram can be represented by the following 63 numbers (written as 21 sets of 3 numbers).

0, 126, 0  
1, 255, 128  
7, 255, 224  
15, 255, 240  
15, 255, 240  
31, 255, 248  
31, 255, 248  
63, 255, 252  
63, 255, 252  
63, 255, 252  
63, 255, 252  
63, 255, 252  
63, 255, 252  
31, 255, 248  
31, 255, 248  
15, 255, 240  
15, 255, 240  
7, 255, 224  
1, 255, 128  
0, 126, 0

The next task is to use these numbers to define a sprite. The way this is done is to convert each number to a *character*, using the function CHR\$ and to put all the characters together into a 63-byte string. This is easily done as follows:

```
DIM BALL$ OF 63
FOR COUNT := 1 TO 63 DO
    READ NUMBER
    *      BALL$ := BALL$ + CHR$(NUMBER)
NEXT COUNT
```

The 63 numbers can be placed in DATA lines to be read by the instruction READ NUMBER. Each number is then converted to a character by CHR\$(NUMBER) and added to the string BALL\$. This is done on the line marked \* above.

In fact (I must confess!) we use a 64-byte string for the sprite. However, the last byte is not derived from the diagram. It is used to indicate whether the sprite is to be a multicoloured sprite or a high-resolution sprite. 0 indicates high resolution, while any

other number indicates multicolour. We will always use high-resolution sprites so we simply add the following line to our program

BALL\$ := BALL\$ + CHR\$(0)

Of course we must dimension our string as follows: DIM BALL\$ OF 64.

Having associated a string variable with the shape of a sprite, our next job is to attach this string to an actual sprite. This is done in two stages:

(1) associate a definition number with the string

(2) associate a sprite number with the definition number,

e.g., DEFINE 1, BALL\$ associates the definition number 1 with the string BALL\$. We can have a maximum of 32 sprite definitions.

Now IDENTIFY 0, 1 associates the above definition with sprite number 0. We can have a maximum of eight sprites, numbered from 0 to 7. Thus

IDENTIFY 0, 1

IDENTIFY 3, 1

IDENTIFY 4, 1

would set up the three sprites 0, 3 and 4 with the ball shape.

As a matter of interest, sprite 7 is our familiar and well-loved turtle. If we issued the instruction IDENTIFY 7,1 following the above definition, we would change our turtle into a ball.

Now that we know how to create a sprite, how can we get it to move around the screen? Well, all we have to do is specify the X, Y coordinates of its position on the screen. As we change these coordinates, the sprite will move around.

We specify the sprite's position by using the command

SPRITEPOS

e.g. SPRITEPOS 0, 100, 150 will cause sprite number 0 to appear at position 100, 150 on the screen.

At this stage you are probably becoming a little impatient to see a full program. Here goes!

```
0010 // PROGRAM 136
0020 // COMELY KATE
0030 // TO DRAW A MOVING BALL USING A SPRITE
0040 //
0050 Y:=100
0060 DIM BALL$ OF 64
0070 FOR BYT:=1 TO 63 DO
0080 READ VAL
0090 BALL$:=BALL$+CHR$(VAL)
0100 ENDFOR BYT
0110 BALL$:=BALL$+CHR$(0)
0120 SETGRAPHIC 0
0130 DEFINE 1,BALL$
0140 IDENTIFY 0,1
0150 SPRITECOLOR 0,7
0160 HIDETURTLE
```

```

0170 FOR X:=1 TO 300 DO
0180   SPRITEPOS 0,X,Y
0190 ENDFOR X
0200 //
0210 DATA 0,126,0
0220 DATA 1,255,128
0230 DATA 7,255,224
0240 DATA 15,255,240
0250 DATA 15,255,240
0260 DATA 31,255,248
0270 DATA 31,255,248
0280 DATA 63,255,252
0290 DATA 63,255,252
0300 DATA 63,255,252
0310 DATA 63,255,252
0320 DATA 63,255,252
0330 DATA 63,255,252
0340 DATA 63,255,252
0350 DATA 31,255,248
0360 DATA 31,255,248
0370 DATA 15,255,240
0380 DATA 15,255,240
0390 DATA 7,255,224
0400 DATA 1,255,128
0410 DATA 0,126,0
0420 //
0430 END

```

Run the program and watch the action. Isn't it nice? Everything should be pretty clear from our previous discussion. We have changed a few variable names e.g. BYT for COUNT, VAL for NUMBER, and have added a few items. Line 150 gives sprite 0 the colour 7 (yellow) and line 160 rubs out the turtle. The movement of the sprite is effected by means of the FOR loop in lines 170 to 190. The Y coordinate was set to 100 on line 50, that is, about half way up the screen. The loop causes the X coordinate to change from 1 to 300 so that the sprite moves straight across the screen.

## **Down, down, down**

It is just as easy to move the sprite down the screen, as our next program shows:

```

0010 // PROGRAM 137
0020 // COMELY KATE
0030 // TO DRAW A MOVING BALL USING A SPRITE
0040 //
0050 SETGRAPHIC 0
0060 TIM:=50
0070 DIM BALL$ OF 64
0080 X:=150
0090 FOR BYT:=1 TO 63 DO
0100   READ VAL
0110   BALL$:=BALL$+CHR$(VAL)
0120 ENDFOR BYT

```

```

0130 BALL$:=BALL$+CHR$(0)
0140 DEFINE 1,BALL$
0150 IDENTIFY 0,1
0160 SPRITECOLOR 0,7
0170 HIDETURTLE
0180 SPRITESIZE 0, FALSE, TRUE
0190 FOR Y:=200 TO 1 STEP -1 DO
0200 SPRITEPOS 0,X,Y
0210 DELAY(TIM)
0220 ENDFOR Y
0230 //
0240 DATA 0,126,0
0250 DATA 1,255,128
0260 DATA 7,255,224
0270 DATA 15,255,240
0280 DATA 15,255,240
0290 DATA 31,255,248
0300 DATA 31,255,248
0310 DATA 63,255,252
0320 DATA 63,255,252
0330 DATA 63,255,252
0340 DATA 63,255,252
0350 DATA 63,255,252
0360 DATA 63,255,252
0370 DATA 63,255,252
0380 DATA 31,255,248
0390 DATA 31,255,248
0400 DATA 15,255,240
0410 DATA 15,255,240
0420 DATA 7,255,224
0430 DATA 1,255,128
0440 DATA 0,126,0
0450 //
0460 END
0470 //
0480 PROC DELAY(TIM)
0490 FOR COUNT:=1 TO TIM DO
0500 ENDFOR COUNT
0510 ENDPROC DELAY

```

This time we slow down the movement of the sprite by introducing a delay between successive plots of the sprite (line 210). And we have added an extra feature (line 180). The command

**SPRITESIZE 0, FALSE, TRUE**

causes the sprite to expand in the Y direction. This gives the effect of a taller thinner sprite. We could expand the sprite in the X direction by

**SPRITESIZE 0, TRUE, FALSE**

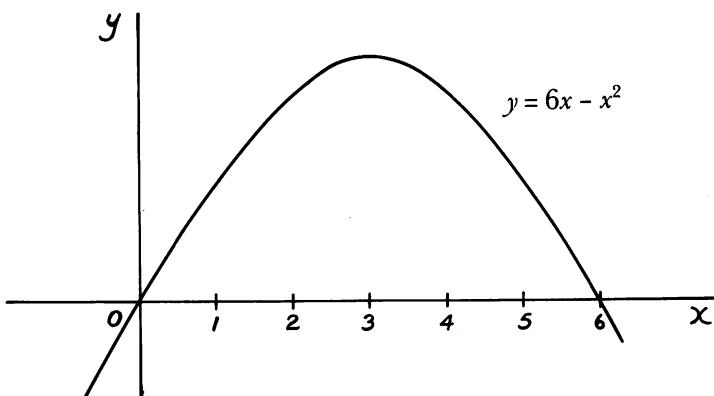
and in both directions by

**SPRITESIZE 0, TRUE, TRUE.**

## Watch the bounce!

Let us now try to make the movement a little more interesting and achieve the effect of a bouncing ball.

Obviously we can set up the sprite exactly as before, so we need only worry about plotting a suitable path for the sprite to imitate the motion of bouncing. To do this we remember a little bit of mathematics, namely, the equation of a parabola. ‘Oh, my God!’ you say, ‘that sounds very difficult!’ Not at all — remember those  $X^2$  graphs? — they were parabolas. For example, the graph of  $Y = 6X - X^2$  is as shown in the diagram:



The plan is to move the sprite repeatedly along a parabolic path. Every time it arrives at the bottom of the screen we will send it up again along another parabola, to achieve the bouncing effect. In addition, we want to keep reducing the height of the bounce. We can do this by multiplying the Y coordinate by a fraction between 0 and 1, e.g. 8/10 or 0.8.

The particular parabolic equation that does the job for us is

$$\begin{aligned} Y &= 10X - X^2/4 \\ &= X(10 - X/4) \end{aligned}$$

The values 10 and  $-1/4$  were chosen to make reasonably good use of the screen space. You will, of course, express your individuality by exploring different values.

The full program is shown below.

```
0010 // PROGRAM 13B
0020 // COMELY KATE
0030 // TO DRAW A MOVING BALL USING A SPRITE
0040 //
0050 SETGRAPHIC 0
0070 DIM BALL$ OF 64
0080 FOR BYT:=1 TO 63 DO
0090   READ VAL
0100   BALL$:=BALL$+CHR$(VAL)
0110 ENDFOR BYT
```

```

0120 BALL$:=BALL#+CHR$(0)
0130 DEFINE 1,BALL$
0140 IDENTIFY 0,1
0150 HIDETURTLE
0160 MOVE
0170 //
0180 DATA 0,126,0
0190 DATA 1,255,128
0200 DATA 7,255,224
0210 DATA 15,255,240
0220 DATA 15,255,240
0230 DATA 31,255,248
0240 DATA 31,255,248
0250 DATA 63,255,252
0260 DATA 63,255,252
0270 DATA 63,255,252
0280 DATA 63,255,252
0290 DATA 63,255,252
0300 DATA 63,255,252
0310 DATA 63,255,252
0320 DATA 31,255,248
0330 DATA 31,255,248
0340 DATA 15,255,240
0350 DATA 15,255,240
0360 DATA 7,255,224
0370 DATA 1,255,128
0380 DATA 0,126,0
0390 //
0400 END
0410 //
0470 PROC MOVE
0480 C:=1
0490 SPRITECOLOR 0,RND(1,15)
0500 FOR BOUNCE:=0 TO 6 DO
0510   FOR X:=1 TO 40 DO
0520     Y:=C*X*(10-X/4)
0530     SPRITEPOS 0,40*BOUNCE+X,2*Y
0540   ENDFOR X
0550   C:=.8*C
0560 ENDFOR BOUNCE
0570 C:=1
0580 MOVE
0590 ENDPROC MOVE

```

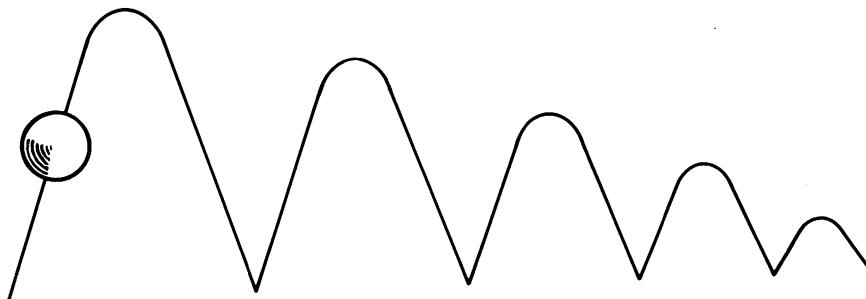
The sprite is set up in the usual fashion. The instructions governing the movement of the sprite are placed in a procedure. The variable C is used to reduce the height of the bounce and is set to 1 to begin with (line 480). It is reduced to 0.8 of its value for each bounce (line 550). Seven bounces across the screen are achieved by the FOR loop from lines 500 to 560. The actual bounce is plotted in lines 510 to 540. X goes from 1 to 40. Y is calculated on line 520. Notice how the normal value of Y, i.e.  $10X - X^2/4$ , is multiplied by C.

The actual position of the sprite is calculated on line 530. It is not a straightforward X, Y plot. The X position is got by adding an appropriate multiple of 40 to X. The multiple used is the value of BOUNCE. Thus if X = 15 and BOUNCE = 0, the X position is actually 15; if X = 15 and BOUNCE = 2, the X position is  $40 * 2 + 15 = 95$ , and so on.

The Y position is a little easier to understand. We simply multiply Y by 2, which has the effect of sending the ball twice as high twice as fast as if we left Y unaltered.

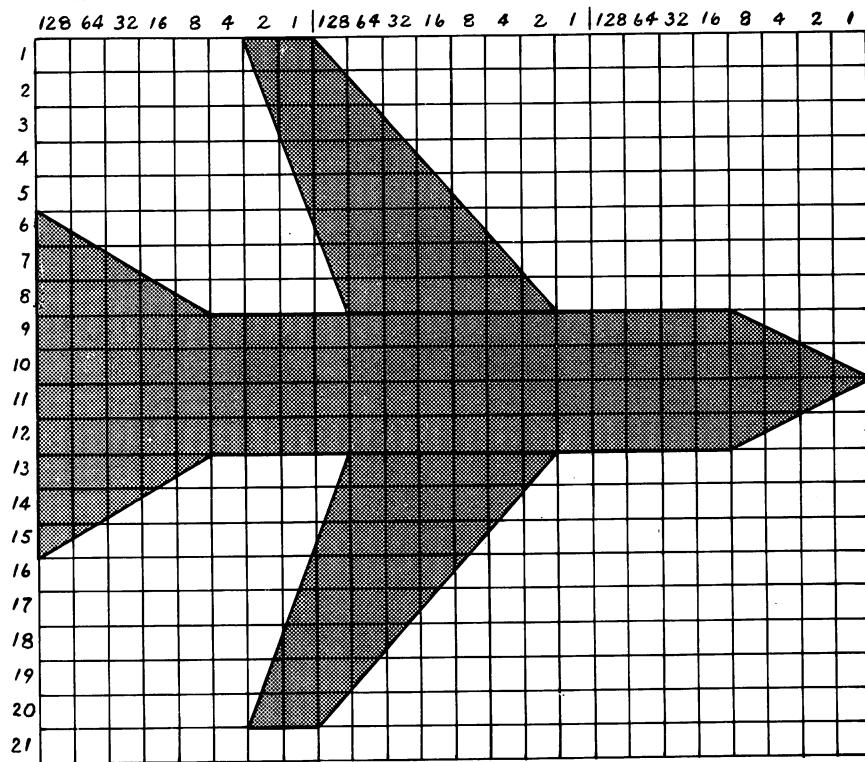
One other little feature is the setting of the colour of the ball to a random colour (line 490).

The procedure calls itself recursively (line 580) and so the ball will go on bouncing forever until we press the STOP button.



## Action Play

Let us finish off with a sprite extravaganza! We will set up definitions for all eight sprites. Let us make five balls and three aeroplanes. The shaping of a plane is carried out on the same grid as before.



```
0010 // PROGRAM 139
0020 // COMELY KATE
0030 // TO FILL THE SCREEN WITH MOVING SPRITES
0040 //
0050 SETGRAPHIC 0
0060 TIM1:=50
0062 Y5:=50
0064 Y6:=100
0066 Y7:=150
0070 DIM BALL$ OF 64
0080 DIM PLANE$ OF 64
0090 FOR BYT:=1 TO 63 DO
0100   READ VAL
0110   BALL$:=BALL$+CHR$(VAL)
0120 ENDFOR BYT
0130 BALL$:=BALL$+CHR$(0)
0140 //
0150 FOR BYT:=1 TO 63 DO
0160   READ VAL
0170   PLANE$:=PLANE$+CHR$(VAL)
0180 ENDFOR BYT
0190 PLANE$:=PLANE$+CHR$(0)
0200 //
0210 DEFINE 1,BALL$
0220 DEFINE 2,PLANE$
0230 IDENTIFY 0,1
0240 IDENTIFY 1,1
0250 IDENTIFY 2,1
0260 IDENTIFY 3,1
0270 IDENTIFY 4,1
0280 IDENTIFY 5,2
0290 IDENTIFY 6,2
0300 IDENTIFY 7,2
0310 SPRITESIZE 6,TRUE,TRUE
0320 SPRITESIZE 0,TRUE,FALSE
0330 SPRITESIZE 1,TRUE,TRUE
0340 SPRITESIZE 2,FALSE,TRUE
0350 MOVE
0360 END
0370 //
0380 DATA 0,126,0
0390 DATA 1,255,128
0400 DATA 7,255,224
0410 DATA 15,255,240
0420 DATA 15,255,240
0430 DATA 31,255,248
0440 DATA 31,255,248
0450 DATA 63,255,252
0460 DATA 63,255,252
0470 DATA 63,255,252
0480 DATA 63,255,252
0490 DATA 63,255,252
0500 DATA 63,255,252
0510 DATA 63,255,252
0520 DATA 31,255,248
0530 DATA 31,255,248
0540 DATA 15,255,240
0550 DATA 15,255,240
0560 DATA 7,255,224
0570 DATA 1,255,128
0580 DATA 0,126,0
0590 //
0600 DATA 3,128,0
0610 DATA 1,192,0
0620 DATA 1,192,0
0630 DATA 1,224,0
0640 DATA 0,240,0
```

```

0650 DATA 128,248,0
0660 DATA 224,124,0
0670 DATA 248,126,0
0680 DATA 255,255,240
0690 DATA 255,255,254
0700 DATA 255,255,254
0710 DATA 255,255,240
0720 DATA 248,126,0
0730 DATA 224,124,0
0740 DATA 128,248,0
0750 DATA 0,240,0
0760 DATA 1,224,0
0770 DATA 1,192,0
0780 DATA 1,192,0
0790 DATA 3,128,0
0800 DATA 0,0,0
0810 //
0870 PROC MOVE
0880 X0:=RND(1,300)
0890 X1:=RND(1,300)
0900 X2:=RND(1,300)
0910 X3:=RND(1,300)
0920 X4:=RND(1,300)
0960 SPRITECOLOR 0,RND(1,15)
0970 SPRITECOLOR 1,RND(1,15)
0980 SPRITECOLOR 2,RND(1,15)
0990 SPRITECOLOR 3,RND(1,15)
1000 SPRITECOLOR 4,RND(1,15)
1010 SPRITECOLOR 5,RND(1,15)
1020 SPRITECOLOR 6,RND(1,15)
1030 SPRITECOLOR 7,RND(1,15)
1040 FOR P:=1 TO 200 DO
1050 SPRITEPOS 0,X0,200-P
1060 SPRITEPOS 1,X1,200-2*P
1070 SPRITEPOS 2,X2,200-.5*P
1080 SPRITEPOS 3,X3,200-P
1090 SPRITEPOS 4,X4,200-P
1100 SPRITEPOS 5,1.7*P,Y5
1110 SPRITEPOS 6,1.7*P+10,Y6
1120 SPRITEPOS 7,1.7*P,Y7
1130 ENDFOR P
1140 MOVE
1150 ENDPROC MOVE

```

Lines 70 to 190 create the strings BALL\$ and PLANE\$ from the two blocks of data (lines 380 to 580 and 600 to 800 respectively). These strings give us the two sprite shapes we require. Shape 1 is then defined as BALL\$, and shape 2 as PLANE\$ on lines 210 and 220. Sprites 0 to 4 are then identified with shape 1 (BALL\$) and sprites 5, 6, 7 with shape 2 (PLANE\$) on lines 230–300. Finally we specify that sprite 6 is to be expanded in both directions (line 310), sprite 0 is to be expanded in the X direction (line 320), sprite 1 in both directions (line 330) and sprite 2 in the Y direction (line 340).

The procedure MOVE is responsible for the movement. It is called on line 350. Within the procedure (lines 870 to 1150) the balls are given random X coordinates so that they will float in random positions down the screen for each repetition of the procedure. The Y coordinates of the planes had already been set to fixed values on lines 62, 64, 66, so that they move in fixed positions across the screen.

The sprites are given random colours in lines 960 to 1030.

The actual movement is controlled by the FOR loop from lines 1040 to 1130. Sprite 1 is made to move twice as fast and sprite 2 half as fast as the others (see lines 1060 and 1070). The middle plane, sprite 6, is made to move slightly ahead (10 units) of the other two (see line 1110).

The procedure MOVE calls itself recursively from line 1140 and so the program continues forever.

## Other Graphics Commands

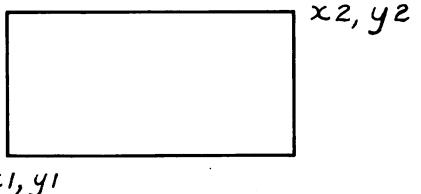
There are other graphics commands in COMAL which we did not yet make use of. You will derive much pleasure and benefit from exploring them. We list them here for your enlightenment!

### **FILL X, Y**

Fills in the screen area with the current colour of the pen.

### **FRAME X1, X2, Y1, Y2**

Sets up a screen window within which graphics commands take effect.



### **FULLSCREEN**

Causes the full screen frame to be used for graphics.

### **PENDOWN**

Puts the pen (turtle) down on the screen so that it shows the line it draws.

### **PENUP**

Raises the pen so that its movement is not traced out on the screen.

### **PLOTTEXT X, Y, TEXT\$**

Prints the text in TEXT\$, starting at position X, Y on the screen frame.

### **SETXY X, Y**

Puts the turtle at position X, Y on the screen.

### **SPLITSCREEN**

Gives room for two text lines above the graphics screen.

### **TURTLESIZE N**

Sets the turtle size to N where N is a number in the range 0 to 10.

## **DATACOLLISION N, TRUE**

Tests for a collision between turtle number N and data on the screen and resets the collision flag. If we had used FALSE instead of TRUE, the collision flag would not have been reset.

## **HIDESPRITE N**

Turns off and hides sprite N.

## **PRIORITY N, TRUE**

Gives data priority over a sprite in the event of a collision (or vice versa if FALSE).

## **SHOWSPRITE N**

Turns on sprite N so that it may be seen.

## **SPRITECOLLISION N, TRUE**

Checks to see if sprite N has collided with another sprite and resets the collision flag (doesn't reset if FALSE)

## **Summary**

The full list of COMAL graphics commands which may be available is:

### **Turtle Commands**

BACK BACKGROUND BORDER CLEAR DRAWTO FILL  
FORWARD FRAME FULLSCREEN HIDETURTLE HOME  
LEFT MOVETO PENCOLOR PENDOWN PENUP PLOT  
PLOTTEXT RIGHT SETGRAPHIC SETHEADING  
SETTEXT SETXY SHOWTURTLE SPLITSCREEN  
TURTLESIZE

### **Sprite Commands**

DATACOLLISION DEFINE HIDESPRITE IDENTIFY  
PRIORITY SHOWSPRITE SPRITECOLLISION  
SPRITECOLOR SPRITEPOS SPRITESIZE

## **QUESTIONS and EXERCISES**

1. How do you switch
  - (i) from text mode to graphics mode
  - (ii) from graphics mode to text mode?
2. How can you erase a diagram?
3. Write programs to draw each of the following:
  - (a) a rectangle
  - (b) a hexagon
  - (c) a decagon
  - (d) an isosceles triangle with base angles of  $45^\circ$
  - (e) a circle of radius 40 units
  - (f) a circle of radius 40 units with centre at the centre of the screen
  - (g) an ellipse
  - (h) a hyperbola
  - (i) a rotating circle
  - (j) a rotating ellipse
  - (k) a house
  - (l) a wheel with spokes
  - (m) a clock
  - (n) a wallpaper pattern
4. Write a program to show a plane taking off (use sprites).
5. Write a program to show a ball bouncing off the four sides of the screen.
6. Write a program to show a plane releasing 'parachutes' which float down the screen.
7. Alter Program 132 so that the input invitations to the user are printed on the graphics screen.
8. Write a program to illustrate a simple version of the solar system.

# 17 Computers at work

## Education

Computers are used to assist in the educational process in a number of different ways:

- (a) Administration
- (b) Simulation of experiments in Science, Geography, etc.
- (c) Computer Aided Instruction (CAI)
- (d) Computer Managed Learning (CML)
- (e) Timetabling

The use of the computer in school administration ranges from dealing with the staff payroll to keeping detailed student records. It can be used to keep lists of stock and equipment, class lists, games lists, attendance records, examinations information, minutes of meetings. It can be used to produce "sticky" labels for addressing letters to parents and students. It can keep complete student records containing such items as

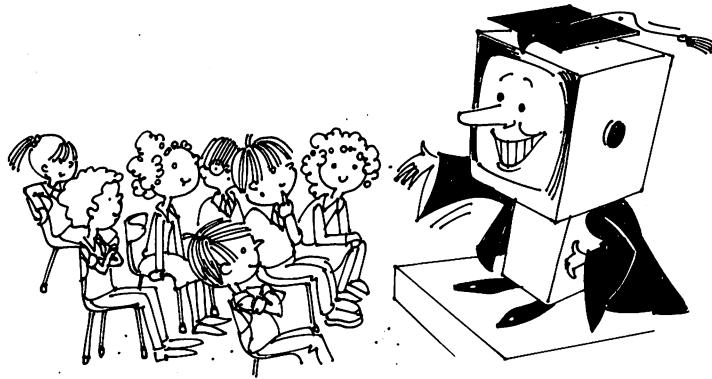
- name
- date of birth
- address
- parents' names
- previous school
- health and dental record
- school test and
  - public examination results
- academic progress report
- behaviour report
- religion
- aptitudes
- career preferences, etc.

One of the advantages of using the computer is the ease with which information may be accessed, modified and printed.

### Computer Aided Instruction (CAI)

A great deal of work is going on all over the world in the area of CAI. Broadly speaking, there are two types of CAI programs:

- (a) Drill and practice programs
- (b) Tutorial type question and answer programs.



Drill and practice programs are extremely useful either when learning a new skill or subject area, or when revising. The student can gain as much practice as he needs and the questions can be varied in an interesting manner.

Tutorial type programs usually guide students through frames of information, encouraging them to learn by investigation and directing them to suitable resources when a lack of understanding is apparent.

## **Computer Managed Learning (CML)**

CML usually refers to the ways in which the computer can assist the teacher in managing the learning process, by way of

- (1) setting test questions suited to individual students
- (2) marking of objective type tests
- (3) keeping records of students' progress
- (4) advising on suitable revision or advanced work.

## **Simulation**

Simulation can be a very effective method of presenting scientific information in an interesting and intelligible way. It can be used in many situations, e.g.

- (a) when actual equipment to conduct the experiments is too costly
- (b) when the experiments are dangerous
- (c) when the experiments would take too long, e.g. studying genetic changes in biology
- (d) when it is difficult or impossible to deal with the real situation, e.g. learning to fly a jet aircraft, or conducting experiments in outer space.

## Timetabling

Timetabling has proved to be a difficult problem for solution by computer and very few 100% successful timetables have been constructed to date. The computer is usually used to assist in the task of timetabling by carrying out much of the routine work. It is also invaluable in giving various types of printed information, such as

- (1) individual teachers' timetables
- (2) class timetables
- (3) activities throughout the school at a particular time, etc.

## Networks

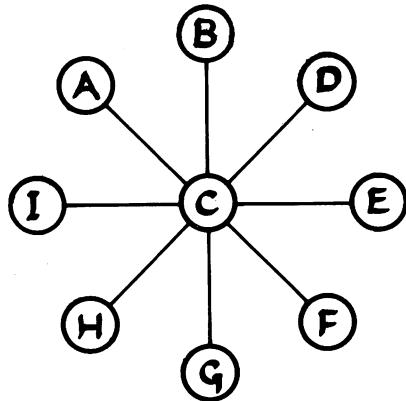
The advent of microcomputers has been a great boon to the development of computer studies in schools. Powerful computer systems which previously would have been large, cumbersome and very expensive, became available at very low cost. Microcomputer systems are small, rugged, reliable and by and large ideally suited to the demanding schools environment.

Nevertheless, as computer studies programmes expand, schools will probably find that one or two 'stand-alone' microcomputers are insufficient to serve their needs fully. It is likely that a computer network system will offer an attractive solution to the problem.

A **network** is a group of interconnected computing systems capable of exchanging information with each other and of sharing facilities such as disk drives and printers. The individual computers attached to a network are often called **nodes**.

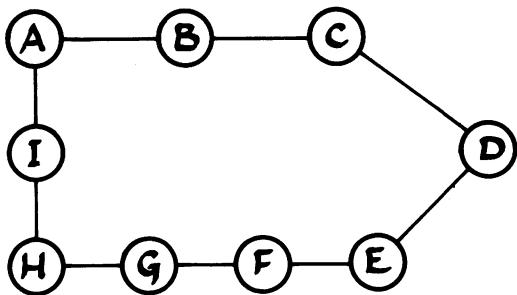
Some of the more popular network configurations are:

### (1) Star Network



Any node can communicate with any other node by sending messages through the central node C. The star arrangement suffers from the disadvantage that if C is out of action the whole network 'goes down' (i.e. fails).

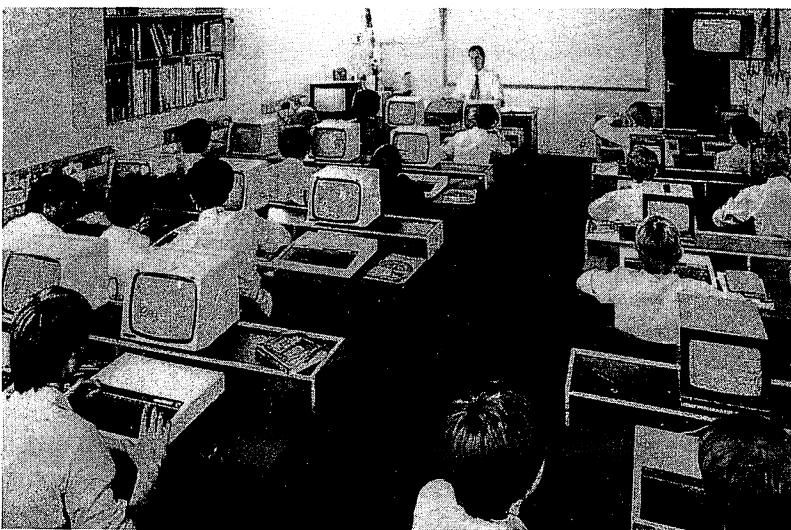
(2) *Distributed Network*



Here there is no central node and alternative paths between nodes usually exist, so that if one node fails the other nodes can still communicate with each other.



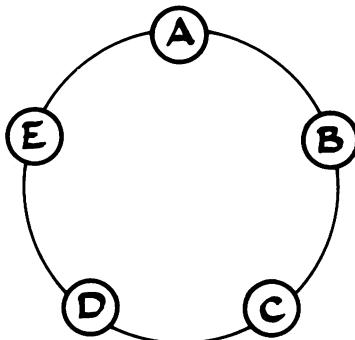
BBC microcomputer



Econet Network  
in action  
in a school

### (3) Ring Network

The ring configuration is very popular for local networks i.e. networks with nodes separated by relatively short distances, e.g. within a building.



Usually messages are sent circulating around the ring and are taken off by whichever node they are addressed to. Again, the construction of the ring is often such that the failure of one node does not affect the circulating of the messages between the other nodes.

A ring network system based on microcomputers is an attractive option for schools, for at least three reasons:

- 1 The cost per microcomputer unit can be lower than for 'stand-alone' systems.
- 2 The units can be used as stand-alone systems if so desired.
- 3 The network offers teachers extra communications and control features. For example, a teacher could transmit a message to an individual terminal or examine the state of affairs at a particular node quite easily.

## Commerce and Industry

More use is made of computers in business than in any other area. They are used in

- Banking
- Insurance
- Stockbroking
- Management Information services
- Market Research
- Payroll Processing
- Word Processing, etc.

Almost 90% of commercial applications of computers are concerned with file-handling. Usually one or more copies of a master file are kept. Changes to the data in the master file are put onto a transactions file, where they may be sorted and validated before being used to update the master file.

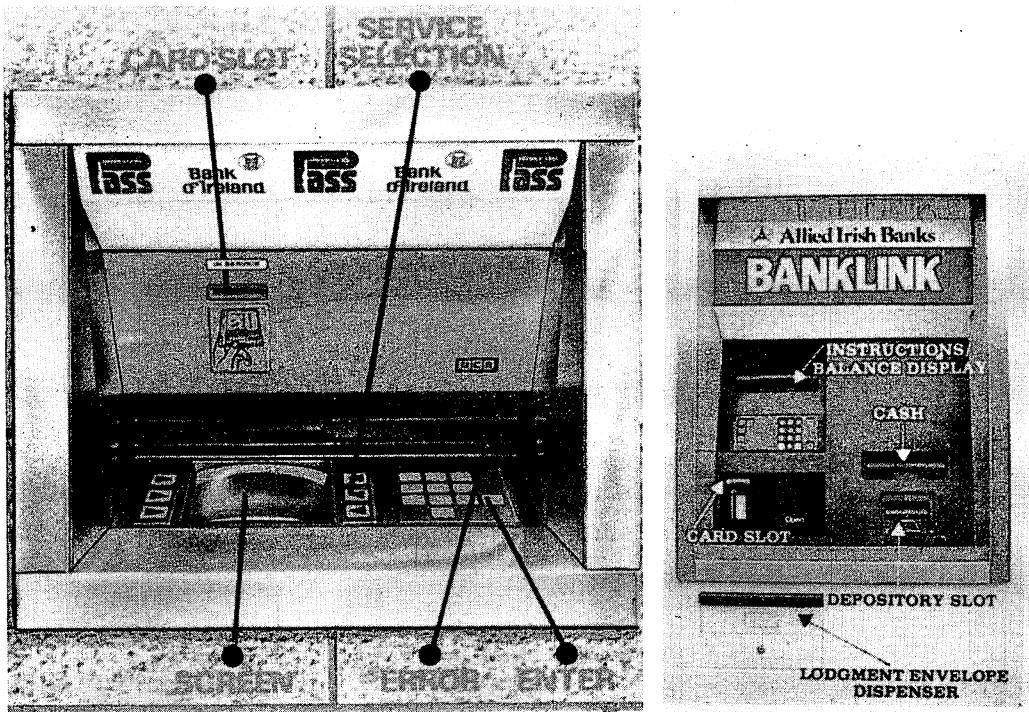
## Banking

Modern banking would not be possible without computers. The range of services and the huge volume of transactions simply could not be handled. Some of the jobs which make use of computers are:

- Calculating interest and bank charges
- Updating foreign currency information
- Updating customer accounts
- Printing customer statements
- Processing cheques

Providing up-to-date information for customers on their accounts.

We have seen the Magnetic Ink Character Recognition (MICR) devices are used to process cheques and that information on customers' accounts is printed in microfiche.



A recent development in this country was the introduction of automatic self-service facilities which allow customers to withdraw cash, order a cheque book, or request a statement, at any time of the day or night. Special machines are used which check

customers' numbers and record the details of transactions on magnetic tape. These machines are, in fact, small computers in their own right. In the future it is likely that these machines will be directly linked to the banks' central computers.

Computers are, of course, essential for the provision of a credit card service.

## **Insurance**

The computer is used in the insurance world for such jobs as

- Calculating policy premiums
- Calculating interest rates and bonuses
- Updating policy holders' files
- Printing policies, schedules for payment, renewal notices
- Issuing payments
- Performing statistical research and formulating new policies.

## **Stockbroking**

In the fast-moving world of stockbroking, the computer can be invaluable in keeping track of changes in share prices and in providing up-to-the-minute information on an on-line enquiry system. It can also help in the calculating of purchase and sale prices of shares and commissions and in the printing of contract notes.

## **Management Information services and Market Research**

Effective management relies heavily on accurate, comprehensive and up-to-date information. This information may include

- Accounts
- Stock information
- Sales analyses
- Personnel problems
- Deployment of resources
- Planning of large projects.

The computer enables this information to be stored, collated, made readily accessible and analysed in diverse ways.

In addition, Market Research information through analysis of sales, questionnaires, etc, and through responses to advertising, can assist in short-term and long-term forecasting. Computer models are often constructed to assist in such forecasting, and also to enhance the training of management personnel.

A much-used technique in stimulating better business methodology is the playing of business games which simulate a real commercial environment and enable the consequences of decision-making to be seen and analysed very quickly. The best-known of such games in this country is the International Computers Ltd. (ICL) business game. ICL runs two competitions based on this game each year, one for schools and one for business firms.



**ICL Management Game Finals**

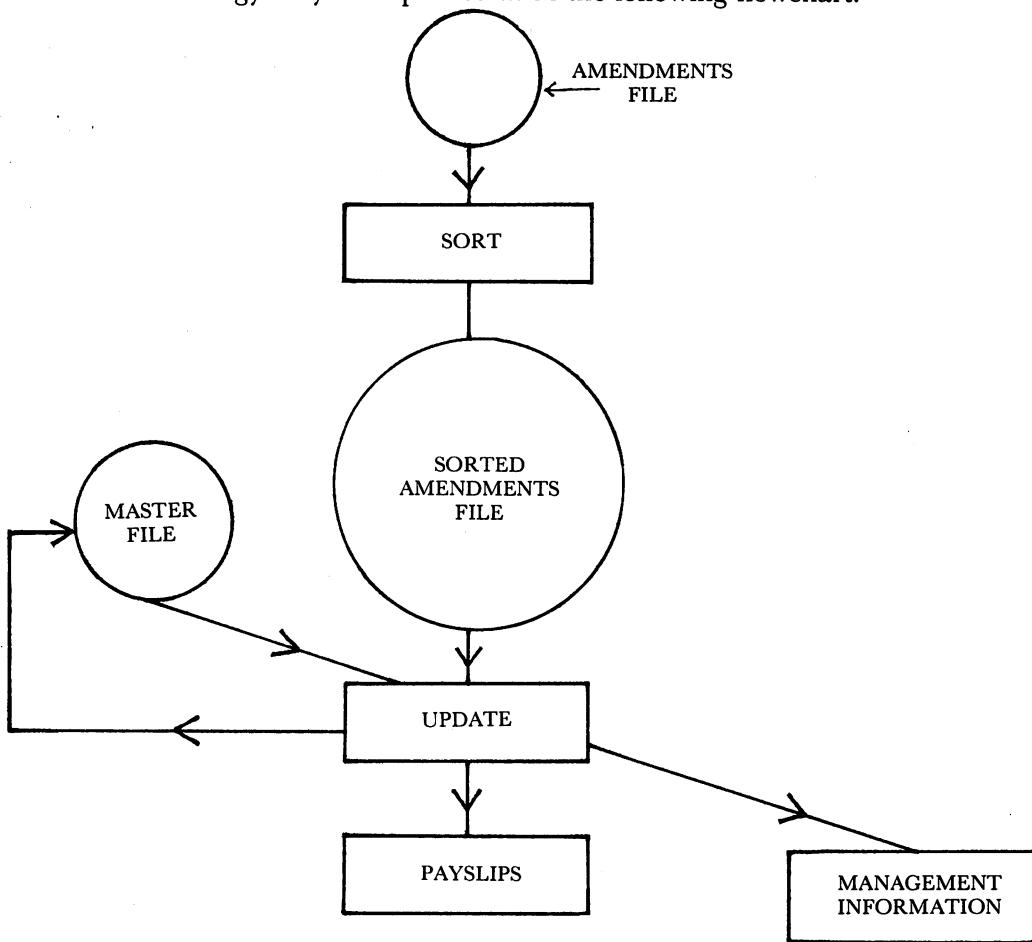
## **Payroll**

One of the first commercial applications of computers was payroll processing. An *employee master file* is kept, which contains various data on each employee, such as

- name
- employee number
- Income Tax information
- Pay Related Social Insurance information
- VHI payments
- pension schemes
- union dues, etc.

This file is updated weekly or monthly and the appropriate salary payments and deductions are calculated.

The overall strategy may be expressed as in the following flowchart:



## Word Processing

One of the most frustrating things which besets the life of every typist is the correcting of typing errors, often necessitating the re-typing of a whole page. Word processing systems can eliminate this frustration by magnetically recording the text, allowing it to be fully checked for accuracy and completeness on a screen display, before finally producing a hard copy.

Word processors, however, usually allow much more sophisticated work to be done. This may include

- (1) Text justification into lines, paragraphs and pages
- (2) Editing of text through the insertion, deletion and replacement of characters, words, or even whole segments

- (3) Production of standard letter forms
- (4) Information storage and retrieval
- (5) Easy checking, correcting and updating of files.

## **Process Automation**

Computers are being used increasingly in the automating of a wide variety of processes. No doubt you have seen television advertisements showing automated car plants, where industrial robots controlled by computers are involved in the complete assembly of cars.

Because of their speed, accuracy and insusceptibility to boredom, computers are well suited to the control of complex processes where

- (1) precisely-timed actions must be taken
- (2) very accurate measurements must be made
- (3) a great variety of instruments must be monitored
- (4) critical or potentially dangerous activities are involved

Some of the areas where process control programs have been used are

- Machine Tool Control
- Automated Warehouses
- Air traffic control
- City transport control
- Patient-monitoring systems

## **Law**

'Criminologists, criminal justice officials and others familiar with the problems of crime control have long emphasised that the lack of adequate, complete and timely information lies at the root of many of their problems' according to the United States president's Commission on Law Enforcement and Administration of Justice (1967).

The problems of the fair and timely administration of justice are becoming more acute since it has been discovered that, in many countries, the crime-rate is growing at a faster rate than industry. In the United States, only 25% of known crimes lead to arrest, and many petty crimes are not reported at all.

Computers are essential in the collecting, recording and collating of information on crimes, suspects, arrests, convictions, fingerprints, guns, stolen cars, stolen credit cards, etc. They are essential, too, in the rapid transmission of information from police centre to police centre and to mobile patrols.

Criminologists are interested in the discovery of pattern in criminal behaviour by file-searching, cross-referencing and analysis of fingerprints, voice prints, locations, modus operandi, etc. Huge data banks need to be maintained and serviced.

Apart from the problems of crime detection and criminal apprehension, there are problems due to the ponderous implementation of justice through the courts. Some of these problems are:

- \* long delays in processing cases
- \* difficulties and conflicts in scheduling clerks, solicitors, barristers and judges, in dealing with court cases
- \* large and increasing case inventories
- \* large volume of routine repetitive clerical tasks
- \* difficulty and delay in accessing case information
- \* non-coordination of data on prosecutions, court procedures, precedents, etc.

Computers have a great deal to offer in the solution of this huge information flow problem. Data can be stored in compact form on magnetic media, updated quickly and easily and displayed instantaneously on video screens. Scheduling and case flow management can be improved through computer analysis.

## **Medicine**

Medicine is another area where a great deal of information is gathered and often becomes difficult to coordinate and access. This information ranges from patient details for hospital administration to research data on recent discoveries in medical science.

To enable doctors and hospital administration staff to carry out their jobs properly, it is vital to have accurate data, including

Case histories

Accounting and general administration data

Information on patients' diets, allergies, blood groups, etc.

X-ray data

Laboratory tests

Operations, etc.

When such information is held in traditional paper files, difficulties may arise in a number of ways

- (a) It may take some time to locate the file in an emergency.
- (b) If information is handwritten, it may not always be completely legible.
- (c) If one doctor takes away a file, it may be unavailable for hours to other staff.

In addition to holding complete information readily available to all authorised staff, computers help the cause of medicine in many other ways:

- (1) Patient monitoring systems which register and control factors such as blood pressure, respiration rate, heart rate, body temperature, lung movement, etc.
  - (2) Instrument control such as in advanced X-ray scanners
  - (3) Automatic clinical laboratory testing where blood, urine and other samples may be tested and analysed. The results can be filed away permanently. Printed analyses can be produced for immediate inspection.
  - (4) Medical research
  - (5) Diagnostic systems where large data banks can be used to provide information which may help doctors to diagnose more accurately
  - (6) Patient-questioning systems, where patients may enter the details of their symptoms on a computer terminal. It has been found that patients are often more open and forthcoming at a terminal than they are with their own doctors.
- It is likely that computers will play an ever-increasing role in the provision of adequate health care for all our citizens in the future.

## **Science and Technology**

From the very beginning, computers have been used in the service of science and technology. A great deal of modern scientific research relies on computer processing of vast quantities of data. It would not have been possible to send men to the moon without the aid of computers. Research into improving food production, increasing grain yields, making better use of energy and other resources, makes extensive use of computer technology.

A few of the many areas in which computers play a vital role are

Numerical analysis of conventional and  
non-conventional mathematical problems

Engineering and architectural design

Modelling

Simulation

Weather Forecasting

Electronic communications

Robotics and Artificial Intelligence

Systems optimisation

### *Robotics*

Robotics is an area of Artificial Intelligence which studies methods of building robots to perform complex, dangerous or intelligent activities.

## *Artificial Intelligence*

Artificial intelligence is the study of computer programs and systems which simulate or emulate the intelligent behaviour of human beings in such areas as chess-playing, proving mathematical theorems, understanding natural language, etc.

## *Systems Optimisation*

Systems optimisation refers to attempts to build new systems or change existing systems so that they perform as well as they possibly can.

# **Arts and Humanities**

## **Archaeology and Art History**

The problem here, once again, is one of vast quantities of data to be stored, cross-referenced, and analysed.

Three major application areas may be identified:

- (1) Calculations and statistical analysis of collected data
- (2) Automatic classification
- (3) Data storage and retrieval

In the past, archaeological research has been hampered by the impossibility of collating all the data gathered. The speed of the computer enables all the data to be included in the analyses, and jobs which might take literally hundreds of years by hand, have been reduced to hours, or even minutes. The fact that more data can be taken into account often leads to the discovery of unsuspected relationships and to the provision of new clues to the solution of difficult problems.

## **Language and Literature**

Again three main areas of work may be indicated:

- (1) The construction of Concordances, Indexes and Dictionaries
- (2) Stylistic Analysis and Attribution studies
- (3) Text editing

A concordance is basically a listing of the key words in a work of literature, together with an indication of their context. It is invaluable in aiding scholars in studying documents. Concordances have been produced by hand, but they take many years of painstaking work. For example, a concordance to Wordsworth's works produced in 1910 took 57 000 hours and required about 50 people. A similar computer concordance would take a couple of hundred hours at most.

The construction of dictionaries and indexes to literary works similarly benefit from the untiring accuracy of the computer.

Statistical analysis of such features of author style as  
use of special words  
syllable frequencies  
metrical variations  
use of function words such as *if*, *or*, *indeed*, etc.  
rhythmic preferences

has often led to a deeper analysis of creative writing and to the resolution of difficult problems of authorship. Sometimes it has revealed influences of one writer on another, which had hitherto been unsuspected.

#### *Attribution studies*

Attribution studies are concerned with attributing works of scholarship or literature to the correct authors. The computer is invaluable in analysing works where the identity of the author is in doubt.

#### *Text-Editing*

Text-editing often involves the comparison of different versions of historical texts, the removal of anomalies and inconsistencies and the production of an accurate new edition. Such painstaking work is greatly facilitated by computer analysis.

### **History**

The computer is invaluable in facilitating the storing and retrieval of vast quantities of historical information. It enables details which had previously been left unexamined to be taken into account. For example, an analysis of over 6000 men and women who lent financial support to England's rise to trading power in the late 16th and early 17th centuries, revealed a wealth of detail on social class and political involvement.

### **Music and Art**

The fields of music and art have, perhaps, been two of the more bizarre areas of computer application. (Of course we ourselves dabbled a little bit in computer art (?) from time to time in this book!) Nevertheless, some serious work has been done on the less creative side of the fields, such as

- information storage and retrieval
- encoding musical forms
- analysis of rhythmic, melodic, and harmonic structures
- analysis of musical style
- detection of art forgeries
- tuning of musical instruments

(In Ireland a microprocessor system for helping with the complex task of tuning Uileann pipes, was developed by an engineer with Bord na Móna)

The production of musical and artistic compositions by computer always attracts a certain number of enthusiasts. Such compositions usually make use of the random number function.

## **Government and Public Services**

Government departments such as Finance and Social Welfare are increasingly heavy users of computers. This is not surprising in view of the large quantities of data which must be processed.

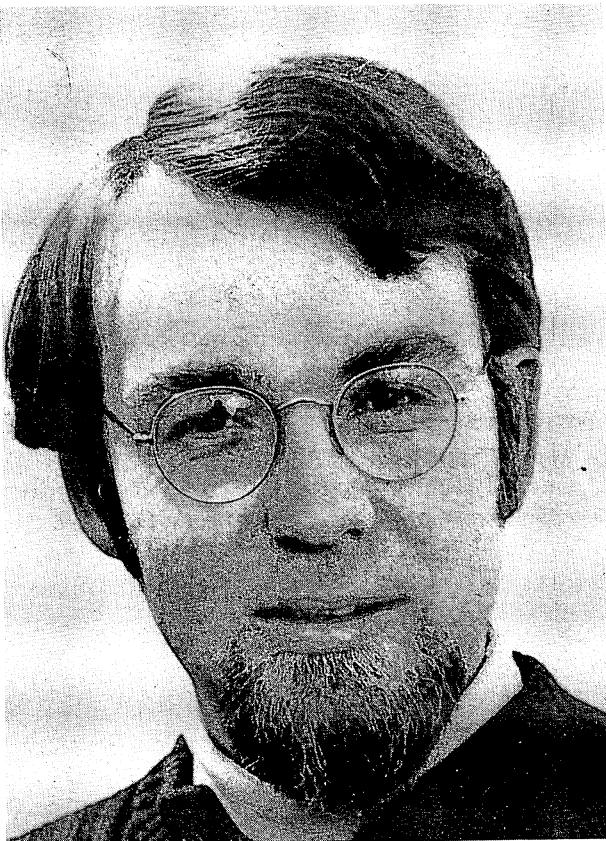
Just as in business applications in general, file-handling forms a major part of Civil Service data processing. It should be possible to provide better service in all aspects of Government through the use of computers. Perhaps in the future it may be possible to enlarge democratic participation in political decision-making, by conducting instant public referenda on major issues, by means of computer-linked home terminals.

## **Information Retrieval**

All applications of computers require the storing and retrieval of great quantities of information. Great success has been achieved in such information processing in the specialised areas we have been considering – Medicine, Law, Business, etc. But the general problem of information retrieval is fraught with very great difficulties:

- \* How can it be arranged for computers to 'read' ordinary printed material?
- \* How can computers 'understand' such material?
- \* How can computers summarise the contents of a document?
- \* What storage structures are best suited to representing textual material?
- \* How can we best interrogate an automated document collection, in searching for relevant material?
- \* How can the computer match the information it holds to the requests which people make?

One of the foremost world experts in Information Retrieval research works here in Ireland. He is C. Van Rijsbergen, who is Professor of Computer Science at University College, Dublin.



C. Van Rijsbergen  
Professor of Computer Science  
University College, Dublin

### **Summary**

In this chapter we have given but a very brief description of some of the important applications and problem areas where the modern electronic digital computer is used as an important tool. In fact, there are very few areas of human endeavour where the computer has no contribution to make.

The following topics were discussed:

Archaeology & Art History	Law
Art	Management
Banking	Medicine
Computer Aided Instruction	Music
Computer Managed Learning	Payroll
Commerce and Industry	Process Automation
Education	Simulation
Government and Public Services	Stockbroking
History	Science and Technology
Insurance	Timetabling
Information Retrieval	Word Processing
Language and Literature	

## **QUESTIONS and EXERCISES**

1. Write down a list of all the tasks performed by
  - (a) the computer in your school
  - (b) the computer in the nearest factory or business premises.
2. Write a COMAL program to
  - (a) create a list of important events in a short period of history
  - (b) allow a user to extract information on this period.
3. Write a COMAL program to simulate
  - (a) a falling stone
  - (b) the reflecting of a ray of light
  - (c) a flashing light
4. Assume that you are employed by a business firm to examine how a computer might help them. Write a small payroll program.
5. Write essays on the use of the computer in
  - (a) Banking
  - (b) Insurance
  - (c) PAYE
  - (d) Law
  - (e) Process Automation
  - (f) Music
  - (g) Art and Design
6. Write programs to draw designs of your own choice. See if you can 'animate' some of the designs.
7. Write a program to cause the computer to produce a poem.
8. Find out where the nearest factory which uses computer-controlled processing is and write an essay about it.

# 18 Social Implications

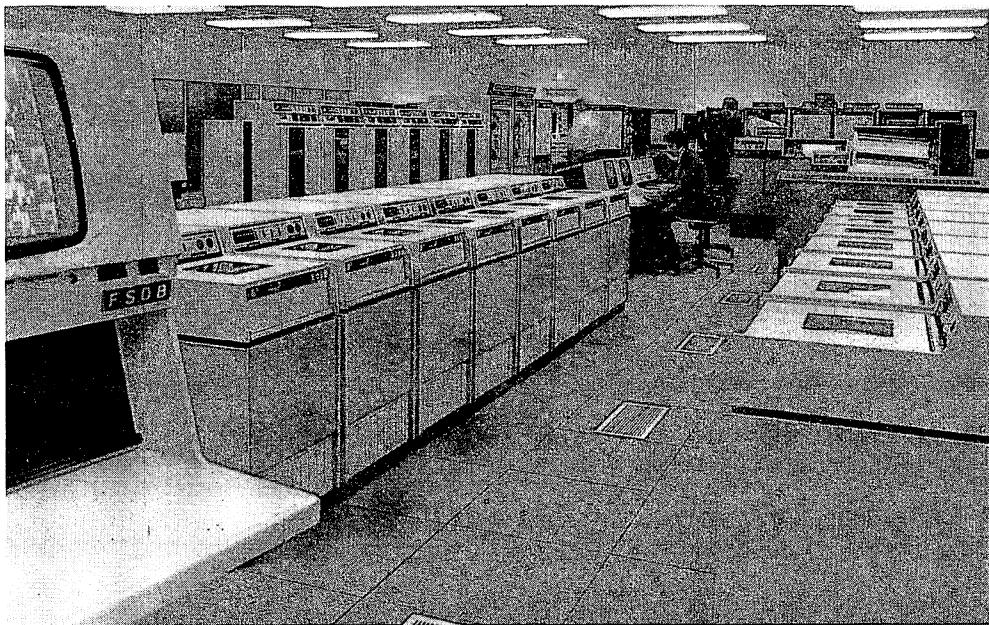
## Up with People

The ascent of man to his present scientific and cultural eminence can be attributed in some measure to his ingenious inventions throughout the centuries. Some of these inventions have been important in solving very specialised problems, e.g. the kidney dialysis machine. Others, though often less urgently important in themselves, have had a much more profound and universal effect upon the progress of mankind. Among the inventions which have changed the whole fabric and movement of society are

The Wheel  
The Clock  
The Printing Press  
The Telescope

The Steam Engine  
The Motor Car  
Television

No doubt you can think of others yourself.



A large computer system

Many authorities now feel that the electronic digital computer will take its place among those inventions which profoundly alter the structure of human society. We

have considered, in the last chapter, some of the vast range of applications which enlist the services of computers. In this chapter we will briefly consider some of the implications of this ever-increasing use of the computer.

## **Freedom, Individuality and Privacy**

There are two vital dimensions to each person's life, the private dimension and the social dimension. The rights and privileges of the first are usually tempered by the needs and obligations of the second. For example, the right to freedom of speech does not authorise an individual to slander another's character, or tell lies about him. There are few absolute rights.

Consider the right to privacy. Total privacy would be totally disastrous. Each individual must interact with the world outside him for such basic needs as air, food, clothes, shelter, etc. But even when these fundamental needs are taken care of, man is still not satisfied. His gregarious nature expresses itself in the desire for social intercourse of various kinds - conversation, companionship, love, etc. Further, in our modern highly complex society, other inroads are made into our private world by the requirement to supply information about ourselves to school authorities, employers and various government and social agencies. Very often, too, when we need to avail of certain government services e.g. social welfare, we are required to reveal even further information about ourselves. In view of all these requirements of self-revelation, we must ask - is there any such thing as the right to privacy?

Our intuition says yes!

Surely we are entitled to privacy in our homes, in our thoughts, in our solitary walks. This is hardly disputable.

The difficult problem is to strike a correct balance between the right to privacy and freedom, and the need to provide good government, fair treatment and scientific planning for all sections of society. The increasing use of computers has caused fresh consideration to be given to the problem of privacy, and in countries such as Britain and the U.S., lively debates have taken place on the issue. Let us examine some of the considerations involved.

## **Consent**

When information is required from a person, it is important that he consent to divulge this information. It is equally important that the information be used only for the purpose for which consent has been given. For example, if you give medical information to your doctor, it would be unjust if he were to pass this information on to somebody who might use it against you. In fact, of course, doctors are obliged by the Hippocratic Oath not to divulge such information. Similarly you would expect that when you give your address to the school authorities, it would not be passed on to any casual enquirer without your consent.

However, there are occasions when a person is obliged to divulge information, even when he does not readily consent. For instance, the tax authorities require each person in gainful employment to reveal his earnings each year. People are often reluctant to do this. Yet it would seem that it is the only fair way to share the burden of maintaining essential state services.

## **Confidentiality**

A person's consent to divulge information about himself is often given on the understanding that the information will be kept in confidence. If confidentiality is preserved and only authorised use is made of personal data, then the threat to privacy is diminished. It is important, for example, that census returns are treated in confidence and that only sworn officers are allowed to examine individual reports.

## **Anonymity**

A great deal of information is gathered nowadays by means of polls, e.g. the Gallup Poll of political opinion. People are much more likely to reveal their true opinions on goods, or services, or political policies, if they are convinced that their names are not known or used. Anonymity can be preserved in such cases as census returns, by removing the name or other identifiers before passing on the information to other authorities for analysis.

## **The Computer and Privacy**

You are probably asking - why all this emphasis on a human problem such as privacy, in a book about computers? The answer is simple.

Computers enable vast quantities of data to be organised, collated and analysed very quickly, very comprehensively and often very subtly.

It is true that even before computers were invented a great deal of data was collected about each citizen, e.g.

- Birth certificate
- Baptismal certificate
- School records
- Examination records
- Employment records
- Social Welfare records
- Tax records
- Bank records
- Mortgage records
- Motor Tax records
- Health records
- Marriage records, etc.

# How French protect the privacy of man-in-street against the databank

FRANCE's National Commission for Information Technology and Individual Liberties, CNIL, which will be three years old this month, is proud of its record in protecting the man in the street against encroachments by databanks on his rights and privacy.

The Commission, set up by Act of Parliament during the presidency of Valéry Giscard d'Estaing, continues to act without any constraints under his successor François Mitterrand. CNIL is publicly recognised as a valuable and effective institution.

Its task is summed up in the law under which it was established: "Information technology must be at the service of each citizen. It must develop in a framework of international co-operation. It must not damage the human identity, human rights, privacy, the lib-

ties of the NCC. NOT only are members of the BCS at loggerheads with each other on privacy, but there appears to be little accord between such DP powers as the IDPM and the

The catalogue of privacy proposals has been around since the formative years of data processing. Certainly the National Council of Civil Liberties under the leadership of Patricia Hewitt has been active in the computer field as far back as 1968 when it published a timely report which closed with the stirring words: The time for action is now.

Four years later the action call was heeded and the Younger Re-

is refused, the issue is referred to the Council highest ruling law.

Although number of express pronounced only for the chin File "This file instance mothers dered it not esta basis a criteria. "The con Comm of chil discrin

The way and most of them are left feeling distinctly queasy about the o men's failure to gras

The

headquarters agreement with France is negotiated shortly. Many other nations could challenge

## Storm brewing over govt data protection paper

by Kevan Pearson

THERE is much concern over what is widely viewed as the inadequacy of the data protection proposals outlined in a White Paper, published before Easter. The interested parties have now had a chance to digest the paper and most of them are left feeling distinctly queasy about the government's failure to grasp

Patricia Hewitt, general secretary of the National Council for Civil Liberties, said: "It is one of the most glaring loopholes in a white paper full of loopholes."

The government paper, save

## Japan acts on privacy

by Philip Hunter

JAPAN is expected to introduce a policy on computer privacy in the spring.

The Japanese government has acknowledged public fear of the invasion of privacy and is committed to implementing guidelines on privacy protection and international transfer of personal data, as

About half of the Japanese people questioned in a recent survey answered that they thought their computerised society resulted in excessive quantities of information, and more than half felt that privacy violations were increasing.

● Iceland has also established laws on data protection and these, like those of Norway, prohibit the export of private data collected at home.

All these records taken together could provide a very comprehensive profile of an individual. However, the data was fragmented and held in different locations. It was very difficult and time-consuming to try to combine all the information into one comprehensive picture.

The fragmentation of the data provided a measure of security against invasion of privacy. Nowadays such data is likely to be held on computer files and the danger of speedy access and collation is very much greater. This is why there is a good deal of concern about the amount of information which is held on computer files and why much thought is being given to

- (1) the accuracy of data
- (2) the security of data
- (3) the authorised use of data
- (4) the control of data.

## **Accuracy**

As you undoubtedly know by now, it is very easy to enter wrong information into a computer file.

If you have an error in a program, the computer may detect it and report an error message. However, if you have a mistake in the data used by the program, the mistake may never be detected and the program may appear to run correctly. This type of error can be very subtle and very damaging.

For example, suppose a person's record indicated that he had 10 convictions for drunken driving instead of 0 convictions. This could destroy his chances of employment, or promotion, or whatever. If the person did not know that his record contained this error, he would be greatly puzzled as to why he never received favourable consideration.

Data can be inaccurate and misleading in other ways than through mistyping. For example, a person may be accused of some crime and be found not guilty and acquitted. Suppose that the fact that he has been in court is recorded, but the fact that he is acquitted is not. He thus ends up with an undeserved criminal record. Such incomplete and misleading data can be very harmful.

## **Security**

Data of a highly confidential or sensitive nature should be secure from unauthorised access. It is not sufficient for a doctor or a banker to abstain from wilfully divulging information on a client. It is necessary to safeguard such information so that those who have no right to see it are prevented from doing so. In fact, in this respect the computer can be used as an aid to security through a system of passwords, codes and electronic locks which make it virtually impossible to gain access to files of data without proper authorisation. Data in a magnetic file can often be safer than data in an old-fashioned filing cabinet.

## **Authorisation**

We have already referred in several places to the need for proper authorisation to access and use personal data. Authorisation may range from complete authorisation to read and alter files, to limited authorisation to read, but not change, part of a file.

Protection from unauthorised access may be achieved by fragmenting the files, i.e. holding different parts of the data in separate files, so that someone who gains access to one part of the data, is locked out from other parts of the data, and is therefore unable to coordinate the information.

## **Control**

The need for accuracy, security and proper authorisation implies that careful control is exercised over the gathering, recording and filing of data. Information should be checked and double-checked, secure coding schemes and filing arrangements should be adopted, and personnel handling sensitive data should adhere to a strict ethical code. Files should be kept in secure locations and protected from damage by fire or other disaster.

Furthermore, each citizen should have a measure of control over data relating to him personally and be given opportunities to verify that the data is accurate, complete and up-to-date. He should be allowed to call for changes to the data as circumstances change and be informed as to the use which will be made of personal data.

## **Action**

The debate on privacy issues began in the U.S. as far back as 1965, when the Dunn report examined a proposal to establish a Federal Data Centre which would bring together many thousands of files of personal data. In the discussion, which has continued up to the present day, certain issues have been highlighted:

- (a) Information must be gathered if the quality of decision-making is not to be degraded.
- (b) Important data should be collected, preserved and centralised for economic and statistical research purposes.
- (c) Techniques should be developed to facilitate research, but prevent disclosure to the wrong people.
- (d) Organizational and legal safeguards should be drawn up, to prevent misuse of private information.
- (e) An independent body should be established, to monitor and police the work of data-gathering agencies.

In Britain the most important work so far has been done by the Younger committee. A report was published in 1972 which concluded that the computer, at that stage, was not a threat to privacy. It recommended the adoption of ten basic principles in dealing with the collection and use of data:

- (1) Information should be regarded as held for a specific purpose and not used, without appropriate authorisation, for other purposes.
- (2) Access to information should be confined to those authorised to have it for the purpose for which it was supplied.
- (3) The amount of information collected and held should be the minimum necessary for the achievement of the specified purpose.
- (4) In computerised systems handling information for statistical purposes, adequate provision should be made in their design and programs for separating identities from the rest of the data.
- (5) There should be arrangements whereby the subject could be told about the information held concerning him.
- (6) The level of security to be achieved by a system should be specified in advance by the user and should include precautions against the deliberate abuse or misuse of information.
- (7) A monitoring system should be provided to facilitate the detection of any violation of the security system.
- (8) In the design of information-systems, periods should be specified beyond which the information should not be retained.
- (9) Data held should be accurate. There should be machinery for the correction of inaccuracy and the updating of information.
- (10) Care should be taken in coding value judgments.

The Younger report was followed by a Government White Paper on Computers and Privacy and this led to the setting up of a committee under Sir Norman Lindop called the Data Protection Committee. Ironically, in spite of all the important work done in relation to privacy and data protection Britain is now (1982) one of a very few countries in Europe without privacy legislation. Indeed a 1982 Government White Paper was seen by many to be a backward step and has been severely criticised. Ireland, too, has as yet no laws in this area, but it has been reported that the Department of Justice is preparing legislation.

Most European countries have established statutes to protect the privacy of the individual and to control the use and dissemination of sensitive data. The first such legislation was established in the state of Hesse in the German Federal Republic in 1971. This was followed by the first national legislation in Sweden in 1973.

Two general principles are to be seen at work in the establishment of data protection laws:

- (1) the right of an individual to have access to data relating to himself.
- (2) the concept of licensing data banks containing personal information.

There is increasing pressure on countries which have not as yet adopted privacy legislation to formulate laws consistent with the general approach. One of the main concerns is that these countries may be used to hold and disseminate sensitive data in a way which would be forbidden in countries with proper data protection laws. In fact it is considered that both Britain and Ireland are already losing important business because of their failure to attend to the problem.

## **Data Banks and Communications**

Modern society thrives on efficient means of communication and citizens are bombarded by a continuous flow of information from books, newspapers, radio, television, telephones and computers.

The electronic media - telephones, television and computers - are playing a greater and greater role in the field of communications and this may bring about dramatic changes in styles of living and working. For example, electronic transmission of banking information through credit cards, etc., may bring about a cashless society.

Business people often have to make their frustrated way through traffic-choked cities to their offices, spend much of the day on the phone, and then make their frustrated way home again. With increased efficiency and sophistication of the telephone system, much of their work could be conducted from home. This could have the beneficial effect of reducing city traffic, thereby saving energy and reducing pollution, not to mention increasing the effectiveness and personal satisfaction of the workers. Devices such as the videophone and electronic document transmitters may make 'home offices' a very feasible and attractive proposition.

The fact that accounts, market reports and other business information are held on computer files is a powerful inducement to decentralisation of offices. Access to such information is faster and more efficient through a computer terminal than through a filing cabinet. The terminal may be linked via a telecommunications system to large data banks containing vast amounts of business, legal, educational, recreational and other information. We see the beginnings of these publicly-accessible data banks in the Ceefax and Oracle services provided by British television channels and in the PRESTEL service of the British G.P.O.

It is likely that, in the near future, paper telephone directories will cease to exist. They will be replaced by electronic directories accessed via a computer terminal with an elementary keyboard and small video screen. Experiments have already taken place in France. Disadvantages of the paper system are:

- (1) Thousands of tons of scarce paper are used.
- (2) Directories are often outdated before they reach subscribers.
- (3) Changed and new numbers have to wait until the next directory is published.

These disadvantages would be eliminated by an electronic system. In addition, the new system would cost less.

## **Education, Work and Leisure**

It is clear that educational methods may well be transformed by the existence of CAI, CML, large data banks, television services, etc. Educational content may also undergo dramatic changes, with a switch of emphasis from facts to principles and skills.

An important new emphasis will surely be on education for leisure. It is expected that in the future people will spend less time working for their living and will therefore

have more time for leisure. Thus an education which emphasises community service, care of the old, youth leadership, crafts, skill in games such as chess, etc. will probably be needed to enable people to use their free time enjoyably and creatively.

At present there is growing concern about increased unemployment due to the widespread use of the computer, particularly the microprocessor. So far, however, it appears that computers have not reduced the level of employment and may even have increased it. They have caused new jobs to be created to offset the loss of obsolete types of work. They have changed overall patterns of work. They have resulted in greater productivity.

The decrease in the number of hours worked per week has also not been as dramatic as first thought. In the U.S., where automation is more advanced than anywhere else, the average working week was reduced by only 3 hours between 1948 and 1970.

In the final analysis, it must be admitted that increased automation and computerization is likely to bring about dramatic changes in the level and structure of employment in the future. The effects of these changes could be desirable or painful, depending on the intelligence, common sense and good will with which we respond to them. Governments, employers and trades unions do not appear to have given the problem sufficient attention yet. Among the important questions which must be asked are:

- (1) Is work in the traditional sense becoming an increasingly artificial means of sharing wealth?
- (2) Is full employment really desirable?
- (3) Should people be given a basic living wage irrespective of whether they work or not? and by whom?
- (4) How can we develop means of sharing resources and wealth more equitably throughout the world?
- (5) Is work psychologically necessary for man?
- (6) If paid work plays a decreasing part in our lives, how can we use our increased leisure to the best advantage?
- (7) Is increased automation always desirable?
- (8) Is increased automation the only effective way to reduce waste and utilise resources efficiently and fairly?

## **Working with Computers**

In this section we briefly describe some of the types of work which have grown up around computers.

### **Systems Analyst**

The systems analyst's job is to make a detailed study of a particular job in order -

- (a) to assess whether it is suitable to use the computer for that application,
- (b) to design a computer system to solve the problem.

For large applications, a team of analysts is usually employed. The analysts may actually work for the company involved, or they may be employed on a contract basis from outside the company. Among the tasks which may be performed are the following:

- (1) studying the needs of the company - the existing procedures used in handling the application - the attitudes, skills and training requirements of the relevant personnel
- (2) establishing all the facts involved
- (3) studying input and output requirements - how input will be gathered, checked and formatted - scope, frequency and format of output
- (4) deciding on contents and format of files
- (5) drawing up overall specifications of the solution for the guidance of the programming staff
- (6) setting out appropriate documentation and instructional material for staff in general
- (7) conducting and supervising the testing of the operational system
- (8) monitoring the accuracy and effectiveness of the system
- (9) possibly advising on the choice of equipment.

## **Programmer**

The programmer's job is to use the systems analysts' specifications to plan, write and test the programs required to solve the problem being studied. Again, for large applications, a team of programmers will set to work. Each programmer will tackle a specific portion of the overall task and the separate procedures will then be combined and coordinated to solve the overall problem.

The work of the programmers will probably be supervised and coordinated by a **chief programmer** who will work in close consultation with the systems analysts.

The chief programmer may

- (1) develop detailed programming specifications from the overall plans of the systems analysts
- (2) advise the systems analysts on the feasibility and practicality of the programming specifications
- (3) draw up schedules
- (4) coordinate the work of the programming team
- (5) ensure that good programming standards are adhered to
- (6) examine the clarity and adequacy of the documentation.

The **applications programmers** will

- (1) prepare the detailed logic of the programs
- (2) code the solution in a suitable programming language
- (3) run and test the programs
- (4) debug (i.e. detect and remove 'bugs' or errors) and correct the programs

- (5) draw up the necessary instructions for the operations staff
- (6) look after the maintenance of working programs

In many situations the main part of a programmer's job may be the maintenance of existing program packages. He will thus check that the package is working correctly and keep it in working order, perhaps making small adaptations from time to time if necessary.

In a small company, or for a small application, a programmer may act as systems analyst and programmer, and take jobs right through from the analysis stage to the testing, debugging and maintaining stages.

Computer manufacturers, software houses and other establishments employ **systems programmers**, whose job is to write the vital software such as compilers, operating systems, etc. which is necessary for the proper functioning of a computer system. The work of the systems programmer is highly skilled and usually demands an intimate knowledge of computer architecture.

## **Computer Operator**

The job of the operator is to operate the actual computing machinery and to supervise the running of the computer installation. A large computer installation will employ a number of operators who may work around the clock in shifts. The tasks of the operator may include

- (1) switching on and getting equipment started
- (2) getting the systems programs into action
- (3) notifying the systems programmers, or engineers, if a serious fault develops
- (4) mounting the appropriate magnetic disk packs or tapes, as required by users
- (5) facilitating the work throughput of batch systems, by loading punched cards, paper tape, etc. and collecting printouts
- (6) responding to the operating system through the operator console
- (7) taking care of equipment
- (8) keeping a log of the performance of the system.

## **Data Preparation Staff**

The work here is mainly typing or keyboard operation. Up to recently, and even yet in many installations, it involves punching and verifying cards or paper tape. Nowadays it more usually means using a VDU terminal in a key-to-disk or key-to-tape system. The advantages of the modern system are

- (1) greater control and accuracy
- (2) easier error-correcting
- (3) speed
- (4) flexibility

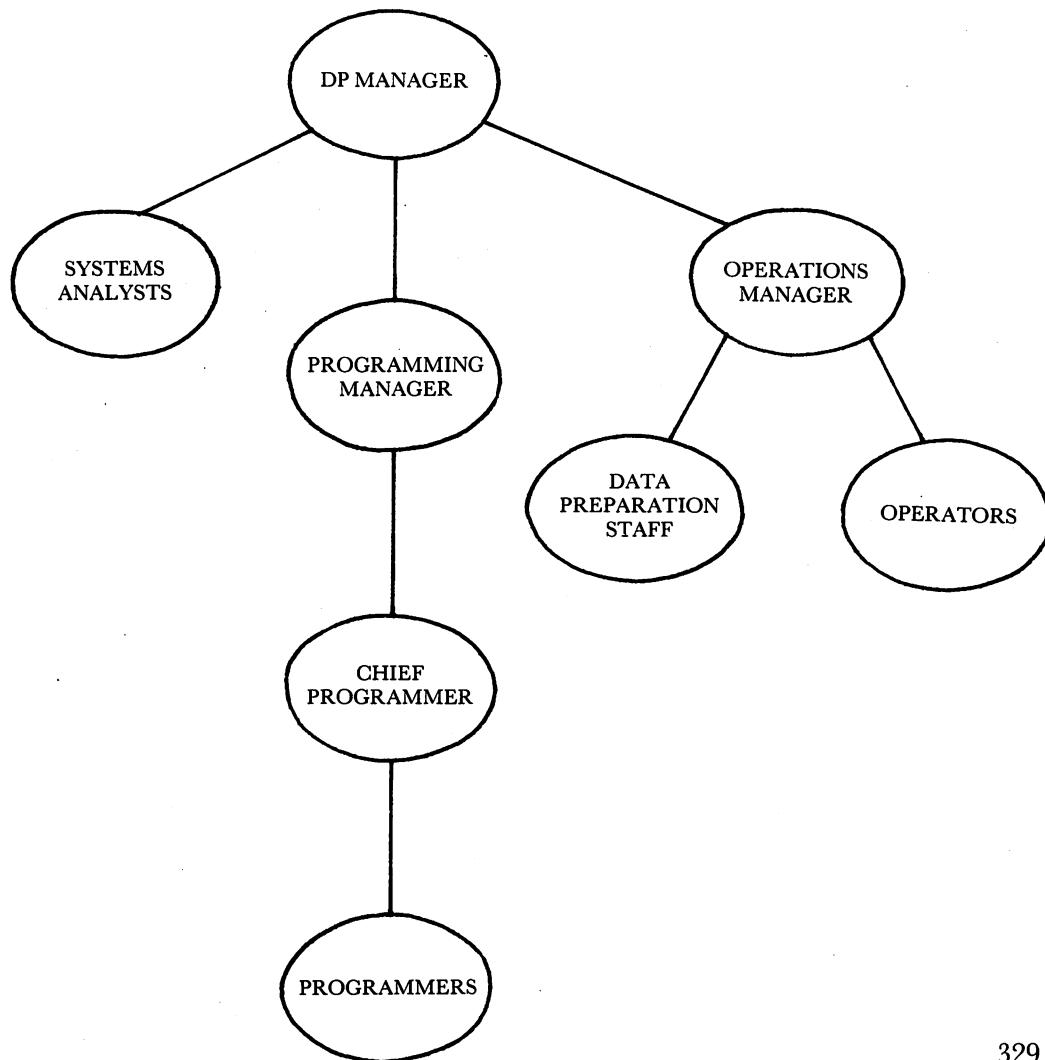
Staff may also be involved in the control and distribution of data and information documents, collecting and checking input and output, despatching output to the correct departments, mailing statements, etc.

## **Data Processing Manager**

The DP Manager is head of the computer installation and is responsible for

- (1) the general solution of the data processing problems of an organization
- (2) introducing new information systems
- (3) advising top management
- (4) planning for future data processing needs
- (5) coordinating the activities of all the DP staff.

### **Structure of a large DP department**



## **Librarian**

A librarian may be employed by a large installation to look after

- (1) Equipment and Programming Manuals
- (2) Documentation
- (3) Magnetic Files

The librarian's task may include

- (1) indexing, cataloguing and physically storing documents and files
- (2) issuing documents and files as required
- (3) keeping duplicates
- (4) replacing deteriorating files
- (5) responding to staff needs for manuals, etc.

Other jobs related to computing occur in engineering, selling, education and training, technical writing and management.

### **Summary**

accuracy of data  
action on privacy  
authorisation  
communications  
confidentiality  
consent  
control of data  
data banks  
data preparation staff

DP manager  
education, work and leisure  
freedom, individuality and privacy  
librarian  
operator  
programmer  
security of data  
systems analyst

### **QUESTIONS and EXERCISES**

1. Find out where your nearest large computer installation is and ask the different staff about their work. Write an assay on each person's job.
2. You are a systems analyst called in by the school to study the problem of using the computer to help with school examinations. Draw up outline specifications of programs to solve the problem.
3. The best solution to the problem of privacy is to create an open society where people are tolerant of each other's idiosyncracies. Discuss this statement.
4. Find out what issues relating to computers have been discussed in the Dáil.
5. What educational changes would you make, to enable people to cope with a 20-hour working week?

# Appendix 1

## COMAL Pre-defined Functions

### **ABS**

Calculates the absolute value of a number or arithmetic expression,  
e.g.  $\text{ABS}(2 * 3 - 10) = 4$

### **ATN**

Calculates the inverse tan (arctan) of a number or arithmetic expression, giving the answer in radians,  
e.g.  $\text{ATN}(1) = 0.7853982$

### **BSTR\$**

Calculates the binary representation of an arithmetic expression or number, rounded if necessary; only deals with numbers 0 to 255,  
e.g.  $\text{BSTR\$(255)} = 11111111$   
 $\text{BSTR\$(8.7)} = \text{BSTR\$(9)} = 00001001$

### **BVAL**

Converts a string of exactly eight bits to its integer value; opposite of BSTR\$,  
e.g.  $\text{BVAL}("00001001") = 9$

### **CHR\$**

Converts a number or arithmetic expression to a single character whose ASCII value equals the given number,  
e.g.  $\text{CHR\$(65)} = A$

### **COS**

Calculates the cosine of an angle expressed in radians  
e.g.  $\text{COS}(I) = .5403023$

### **ERRTEXT\$**

Takes a number or arithmetic expression as argument and reports the corresponding COMAL system error message,  
e.g.  $\text{ERRTEXT\$(6)} = \text{ERROR IN COMMAND}$

### **EXP**

Calculates  $e$  to the power of a given number or arithmetic expression, where  $e = 2.718282$  is the base of natural logs,  
e.g.  $\text{EXP}(3) = 20.08553$

### **FRAC**

Extracts the decimal part of a real number,  
e.g.  $\text{FRAC}(10.246) = 0.2460003$

**INT**

Calculates the largest integer equal to or less than a given number or arithmetic expression,

e.g.             $\text{INT}(7.98) = 7$   
                 $\text{INT}(-7.98) = -8$

**IVAL**

Converts an integer which is in string form to its actual integer value,  
e.g.             $\text{IVAL}("73") = 73$

**LEN**

Calculates the number of symbols contained in a string variable,

e.g.            If  $S\$ = \text{"ORANGE"}$  then  
                 $\text{LEN}(S\$) = 6$

**LOG**

Calculates the logarithm to the base  $e$  (i.e. the natural logarithm) of a number or arithmetic expression,

e.g.             $\text{LOG}(10) = 2.302585$

**ORD**

Converts the first symbol in a string into its ASCII value,

e.g.             $\text{ORD}(\text{"ORANGE"}) = 79$

**PEEK**

Returns the contents of a memory location,

e.g.             $\text{PEEK}(4793) = 253$  (temporarily)

**POKE**

Puts a value into a memory location. The value must be between 0 and 255 inclusive

e.g.             $\text{POKE } 100, 33$  puts 33 into location 100

**POS**

Finds out if a substring is contained in a string and if so where it is placed,

e.g.             $\text{POS}(\text{"AN"}, \text{"ORANGE"}) = 3$   
                 $\text{POS}(\text{"ON"}, \text{"ORANGE"}) = 0$

**RND**

Produces a random number,

e.g.             $\text{RND}(1,6) \rightarrow 4$

**ROUND**

Rounds a real number to the nearest integer,

e.g.             $\text{ROUND}(7.78) = 8$   
                 $\text{ROUND}(-7.78) = -8$

**SGN**

Produces the sign of an arithmetic expression or number, 1 for values greater than 0, 0 for 0, -1 for values less than 0,  
e.g.            SGN(3.84) = 1

**SIN**

Calculates the sine of an angle expressed in radians,  
e.g.            SIN(1) = .8414711

**SPC\$**

Produces a string of spaces where the number of spaces is determined by the argument given to SPC\$,  
e.g.            SPC\$(5) → "       "

**SQR**

Calculates the square root of a number or arithmetic expression,  
e.g.            SQR(9) = 3

**STR\$**

Converts a number or arithmetic expression into a string,  
e.g.            STR\$(79) = "79"

**TAB**

Tabulates to a specified position before next PRINT command,  
e.g.            PRINT TAB(5), "HELLO"

**TAN**

Calculates the tangent of an angle expressed in radians,  
e.g.            TAN(1) = 1.557408

**TRUNC**

Chops off the decimal part of a real number,  
e.g.            TRUNC(-3.86) = -3

**VAL**

Converts a string to a real number,  
e.g.            VAL("17.813") = 17.813

# **Appendix 2**

COMAL has a number of commands and features which have not been covered in the main text for various reasons. Some of these features are considered to be incompatible with good practice in structured programming (e.g. GOTO, GOSUB), while others are somewhat advanced for an introductory text. For the interested student these features are briefly discussed in this appendix. Many of the commands have variations which are not mentioned here but should be explored by the keen student.

## **CALL**

Calls into action a machine code program starting at a specified address in memory,

e.g.           CALL 4132

links to a set of instructions starting at 4132 and sets them in motion. Use of this instruction demands a good knowledge of the machine architecture and machine-level programming.

## **CHAIN**

Causes a program which has been saved on disk to be brought into working store and run. The incoming program replaces the previous program in memory,

e.g.           500 CHAIN "NEWPROG"

## **ENTER**

Similar to LOAD in that it causes a program or file stored on disk to be brought into the work space in main memory. However, unlike LOAD, it does not cause overwriting of the program already in memory except where the line numbers are the same. Thus it allows the combining of two programs in memory and is therefore very useful. An important point to remember, however, is that the program or file to be entered must have been saved on disk by the LIST command, not the SAVE command,

e.g.           LIST SAMPLE

will cause the program in the work space to be stored as an ASCII file on disk under the name SAMPLE. Later this program can be combined in memory with another program already in the work space, by the command

ENTER SAMPLE

## **GETUNIT**

Tells the user which secondary storage device is the current default device,

e.g.           GETUNIT might return DK0:

## **GOSUB . . . RETURN**

A much cruder version than EXEC of calling a sub-program and returning from it. GOSUB is followed by a line number, e.g. GOSUB 500. Unlike EXEC (e.g. EXEC TESTPRIME) the subroutine called by GOSUB has no name. In addition GOSUB has no parameter passing mechanism and is generally less useful than EXEC. The RETURN command is required to cause a return from the called subroutine to the main program.

## **GOTO**

Used to interrupt the normal sequence of program execution and cause the program to continue from another point,

e.g. GOTO 500 transfers the program to line 500 for its next statement.

GOTO can be used with a named label rather than a line number, as in the following example

```
10 DIM NAME$ OF 20
20 INPUT "WHAT'S YOUR NAME": NAME$
30 GOTO 70
40 LABEL FINIS
50 PRINT "BYE NOW"
60 GOTO 90
70 PRINT "HELLO", NAME$
80 GOTO FINIS
90 END
```

## **LOOP . . . EXIT . . . ENDLOOP**

A method of constructing a loop which is terminated when a specified condition is met within the loop,

e.g.

```
10 SUM := 0
20 COUNT := 0
30 LOOP
40 INPUT VALUE
50 IF VALUE < 0 THEN EXIT
60 SUM := SUM + VALUE
70 COUNT := COUNT + 1
80 ENDLOOP
90 PRINT "MEAN = ", SUM/COUNT
100 END
```

The lines between LOOP and ENDLOOP are continually executed until a negative value is read. The loop is then terminated (see line 50) and line 90 is executed.

## **MAT**

Allows a value to be assigned to each element of an array by a single instruction,

e.g. 10 DIM NAMES\$ (50) OF 20

```
20 MAT NAMES$ := "EMPTY"
```

The string "EMPTY" is placed in each of the 50 entries in the array NAMES\$.

**PAGE**

Causes the paper on a line printer which is being used for output to be advanced to the top of the next page.

**RENAME**

Used to change the name of a file stored on disk,  
e.g.           RENAME FILE 1, FILE 2 causes FILE 1 to be renamed FILE 2.

**RESTORE**

Used to restore the data 'pointer' to the beginning (usually) or some intermediate point of DATA lists, to allow re-reading of data.

**SELECT OUTPUT**

Normally the VDU screen is taken as the default device for displaying information output by the PRINT and PRINT USING statements. By using the SELECT OUTPUT command we can change the default to a different device, e.g. a printer or a disk drive,

e.g.           50   SELECT OUTPUT LP:  
causes output to be sent to a line printer.

**SIZE**

Causes information about your work space in memory to be displayed, namely the amount of working storage used, the amount of space left and the amount of memory used for variables.

**UNIT**

To specify which secondary storage device is to be the default device,  
e.g.           UNIT DK1:

**VARPTR**

To reveal the address in memory where a variable is stored,  
e.g.           10   APPLE := 47.6  
               20   PRINT VARPTR (APPLE)

# **Appendix 3**

## **A Basic Toolkit for METANIC COMAL-80 on the APPLE II microcomputer**

Trying to cope with a whole range of commands and novel features can be a bit confusing for the beginner. This appendix attempts to put together a small list of the most useful features and to take the student through a typical work session.

### **Important Commands**

AUTO - for automatic line numbering  
LIST - to display your program  
RUN - to execute your program  
EDIT - to edit your program  
SAVE - to save your program on disk  
LOAD - to bring a copy of a program from disk into working memory  
NEW - to clear work space

Don't forget the

ESC key - to get out of trouble or to end AUTO, etc.

### **To Work**

Typically you will want to enter a program, run it and save it on disk. Proceed as follows:

Put the COMAL A or COMAL B disk in disk drive DK0: (possibly marked A:) and switch on. The red light on the drive will come on and the drive will start whirring. After a while the light will go off and the following will appear on the screen

APPLE II CP/M  
56K VER 2.20  
(c) 1980 MICROSOFT  
A>

Type in COMAL-80 and press RETURN. The disk drive will again be active for a while and then the following will appear on the screen:

COMAL - 80 V1.8  
BY ARNE CHRISTENSEN  
COPYRIGHT (C) 1980, 81, 82 METANIC APS  
ERROR TEXTS (Y/N)?

Enter Y (no need for RETURN this time).

An asterisk followed by a flashing cursor will appear. You are now in the COMAL system.

Enter the command NEW (followed by RETURN).

Enter the command AUTO (followed by RETURN).

The number 10 will appear on the screen and you may enter the first line of your program, followed by RETURN. Numbers 20, 30, etc. will appear in succession as you enter each instruction and terminate each one by pressing RETURN.

When you are finished entering your program press ESC. Line numbers will no longer appear.

Type RUN to have the program executed.

Now place a prepared disk (see below) in disk drive DK1: (possibly marked B:) and type INIT. INIT is required to inform the system that you have inserted a new disk in one of the disk drives.

When disk activity ceases type SAVE DK1: followed by the name you wish to give your program, e.g. SAVE DK1: SAMPLE. When you press RETURN, disk activity will commence and your program will be saved.

## **Preparing a disk**

### **FORMAT**

Place the CP/M 56K (or the COMAL A) disk in disk drive A: and switch on. When A> appears on the screen type FORMAT B: and press RETURN. The message

APPLE II CP/M

16 SECTOR DISK FORMATTER

(C) 1980 MICROSOFT

INSERT DISKETTE TO BE FORMATTED IN DRIVE B:

PRESS RETURN TO BEGIN

will appear. Place a blank diskette in drive B:, close the door and press RETURN.

When the disk has been formatted the following message appears:

FORMAT COMPLETE

FORMAT DISK IN WHICH DRIVE:

If you wish to format another disk, type B: and follow the same procedure as before. If you do not want to format another disk just press RETURN.

### **COPY**

Leaving your newly formatted disk in drive B: and the system disk in drive A:, type  
COPY B: = A:/S

and press RETURN. When the copying is finished you will be invited to make another copy. If you do not wish to do so type N.

There are a number of variations of the COPY command which you may like to explore by referring to the manuals which are supplied with your system. This appendix merely aims to get you up and running!

# Appendix 4

## Dealing with Errors

As we have discovered over and over again, making mistakes is part and parcel of programming, and learning how to correct these errors is often a creative and enlightening process. Sometimes, however, it is a bit frustrating and in this appendix we try to indicate some of the more common errors and how they might arise.

In general, there are two types of error — syntax errors and logical errors. The COMAL system usually detects syntax errors when you are entering a program. If you type in a line with an error in it, the system will stop accepting new lines (temporarily) and will repeat the offending line. The cursor will be positioned somewhere near where the error occurs, and an error message displayed. Sometimes the error message is helpful. At other times it may be uninformative or even misleading and you have to scan the line carefully to detect the mistake.

Some errors will not be detected until you try to run the program, e.g. failure to close off a CASE statement with ENDCASE, or failure to give a variable a value before using it.

## Common errors and their remedies

### Illegal Characters

It is very easy to mistype a character when entering an instruction. Various errors arise:

Example: N 1 := 47

Possible Message: SYNTAX ERROR or :=/:+/- EXPECTED

Cause: Blank in variable name

Correct: N1 := 47

Example: 10 INPUT "ENTER YOUR NAME" 8N\$

Possible Message: SYNTAX ERROR

Cause: 8 instead of : before N\$

Correct: 10 INPUT "ENTER YOUR NAME" :N\$

Example: 10 PRINT JOE : JACK : JILL

Possible Message: SYNTAX ERROR

Cause: Use of : instead of , or ; between print items

Correct: 10 PRINT JOE, JACK, JILL  
or 10 PRINT JOE; JACK; JILL

- Example: N := 124 + 1\$267  
 Possible Message: FORMAT ERROR or NO \$/# HERE  
 Cause: \$ in middle of number  
 Correct: N := 124 + 14267
- Example: N := 124 + 1, 267  
 Possible Message: SYNTAX ERROR or END-OF-LINE EXPECTED  
 Cause: , in middle of number  
 Correct: N := 124 + 1267
- Example: N :=124 + 1%  
 Possible Message: FORMAT ERROR or ILLEGAL CHARACTER  
 Cause: Use of % sign with number  
 Correct: N := 124 + 1
- Example: N := 124 + 12 \* 13  
 Possible Message: SYNTAX ERROR or :=/:+/- EXPECTED  
 Cause: Use of ; instead of :  
 Correct: N := 124 + 12 \* 13
- Example: N% = 124 + 12 \* 13  
 Possible Message: FORMAT ERROR or ILLEGAL CHARACTER  
 Cause: Use of % instead of :  
 Correct: N := 124 + 12 \* 13
- Example: N (= 124 + 12 \* 13  
 Possible Message: OPERAND EXPECTED  
 Cause: Use of ( instead of :  
 Correct: N := 124 + 12 \* 13
- Example: N := 124 + 12; 13  
 Possible Message: SYNTAX ERROR or VARIABLE EXPECTED  
 Cause: Use of ; instead of e.g. \*  
 Correct: N := 124 + 12 \* 13
- Example: 100 DATA 12 13 20  
 Possible Message: SYNTAX ERROR or END-OF-LINE EXPECTED  
 Cause: Blanks separating data  
 Correct: 100 DATA 12, 13, 20
- Example: PRINT "JOE" \* "FRED"  
 Possible Message: TYPE CONFLICT  
 Cause: Use of numeric operator with strings  
 Correct: PRINT "JOE" + "FRED"

## **Use of Wrong Type**

In general, we should distinguish between integers, reals, strings and logical values.

Example: N := "HELLO"

Possible Message: TYPE CONFLICT or WRONG TYPE

Cause: Omission of \$ after N

Correct: N\$ := "HELLO"

Example: N\$ := "AB" = "A" + "B"

Possible Message: TYPE CONFLICT or WRONG TYPE

Cause: Assigning a logical value, true or false (something = something),  
to a string variable

Correct: N := "AB" = "A" + "B"

The next two examples do not give rise to errors on the Commodore-64, but may do so  
on the Apple.

Example: N# := 12.67

Possible Message: WRONG TYPE

Cause: Mixing integer and real numbers in unacceptable fashion

Correct: N# := TRUNC(12.67)

Example: N := 124 + 1 & 267

Possible Message: TYPE CONFLICT

Cause: Use of & (a string operator) with numbers

Correct: N := 124 + 1 + 267

## **Mistyping O for 0 or vice versa**

Example: N := 12O + 12 \* 13

Possible Message: SYNTAX ERROR or END-OF-LINE EXPECTED

Cause: Use of 'oh' instead of zero in 120

Correct: N := 120 + 12 \* 13

## **Mistyping of 1 for I or vice versa**

Example: 1 := 124 + 12 \* 13

Possible Message: NOT A STATEMENT or VARIABLE EXPECTED

Cause: Use of 1 instead of I to left of :=

Correct: I := 124 + 12 \* 13

## **Unbalanced Quote Marks**

Example: PRINT HELLO"

Possible Message: SYNTAX ERROR or STRING NOT TERMINATED

Cause: No quotes before H of HELLO

Correct: PRINT "HELLO"

### **Important Omissions**

- Example: INPUT "ENTER YOUR NAME " N\$  
Possible Message: SYNTAX ERROR or ';' EXPECTED  
Cause: Omission of : before N\$  
Correct: INPUT "ENTER YOUR NAME" :N\$
- Example: FORA := 1 TO 10 DO  
Possible Message: SYNTAX ERROR or END-OF-LINE EXPECTED  
Cause: No separating space between FOR and A  
Correct: FOR A := 1 TO 10 DO
- Example: 10 THIS IS A PROGRAM TO ...  
Possible Message: SYNTAX ERROR or :=/+/- EXPECTED  
Cause: Omission of // or REM  
Correct: 10 // THIS IS A PROGRAM TO ...
- Example: PRINT 4 + 5 \* 4 + 2)  
Possible Message: SYNTAX ERROR  
Cause: Omission of opening bracket  
Correct: PRINT 4 + 5 \* (4 + 2)
- Example: PRINT 4 + 5 \* (4 + 2  
Possible Message: BRACKET ERROR or ')' EXPECTED  
Cause: Omission of closing bracket  
Correct: PRINT 4+ 5 \* (4 + 2)

### **Wrong Dimensioning**

- Example: DIM N OF 20  
Possible Message: SYNTAX ERROR  
Cause: Use of string format for DIM instead of number array format  
Correct: DIM N(20)
- Example: DIM N\$(20)  
Possible Message: SYNTAX ERROR or 'OF' EXPECTED  
Cause: Use of number instead of string format  
Correct: DIM N\$ OF 20

### **Incorrect use of key words**

- Example: READ := 20  
Possible Message: EXPRESSION EXPECTED or VARIABLE EXPECTED  
Cause: Use of key word READ as variable  
Correct: REED := 20
- Example: DIM LIST (20)  
Possible Message: SYNTAX ERROR or A NAME EXPECTED  
Cause: Use of key word LIST as variable  
Correct: DIM LISST (20)

### **Misspelling of key word**

Example: PRANT "HELLO"

Possible Message: SYNTAX ERROR *or* :=/:+/- EXPECTED

Cause: Misspelling PRINT

Correct: PRINT "HELLO"

Example: PRINT RAND (1,6)

Possible Message: COMMAND, ARRAY, SUBSTRING *OR* PROCEDURE  
ERROR

*or* UNDEFINED VARIABLE *OR* FUNCTION VALUE

Message will not appear until program is run

Cause: Misspelling RND as RAND

Correct: PRINT RND (1,6)

### **Wrong-way-round Assignment**

Example: A + B := SUM

Possible Message: SYNTAX ERROR *or* :=/:+/- EXPECTED

Cause: A + B should be on right hand side

Correct: SUM := A + B

### **Using too high line numbers**

Example: 99999 A := B + C

Possible Message: NOT A STATEMENT *or* OUT OF STORAGE

Cause: Line number too high

Correct: 9999 A := B + C

## **Run time errors**

Some errors will not be detected until the program is run.

### **Errors in Program Structure**

These take a number of forms

(a) Omission of ENDIF, ENDCASE, ENDPROC, ENDFOR (*or* NEXT)

(b) Wrong nesting of FOR loops,

e.g. FOR A := 1 TO 10 DO

FOR B := 1 TO 7 DO

NEXT A

NEXT B

*Correct version*

```
FOR A := 1 TO 10 DO  
FOR B := 1 TO 7 DO
```

```
    .  
    NEXT B  
    NEXT A
```

- (c) Using ELSE with a one line IF statement,  
e.g. IF A < B THEN PRINT A  
 ELSE  
 PRINT B  
 ENDIF

*Correct version*

```
IF A < B THEN  
  PRINT A  
ELSE  
  PRINT B  
ENDIF
```

## **Out of Data**

This occurs when a READ instruction fails to find an accompanying piece of data.  
It usually happens when some, but not enough, values are supplied in DATA lines.

## **Error in Indexing**

This usually occurs with an empty string,  
e.g. 10 DIM S\$ OF 10  
 20 INPUT "ENTER ANY STRING": S\$  
 30 PRINT S\$(1)

There is an error here if S\$ is empty, i.e. if user simply presses RETURN in response to INPUT request.

## **Undefined Variable**

e.g. 10 A := 20  
 20 B := A + C

There is an error here, because C has no value.

## **Final Point**

Do not confuse DEL and DELETE.  
DEL is used to delete lines of a program in memory.  
DELETE is used to delete files from a disk.

# Appendix 5

## Editing with Apple COMAL

It is just a little more difficult to edit with Apple COMAL than with a straightforward screen editor. Nevertheless you soon get used to it and it becomes quite routine. To illustrate the technique, let us return to the problem posed on page 13, namely, change the comma to a semicolon in the line

40 PRINT "HAPPY BIRTHDAY", NAME \$

- (a) Enter the command EDIT 40. You will find that the cursor positions itself at the right hand end of line 40.
- (b) Keep pressing the back arrow ( $\leftarrow$ ) key until the cursor is over the comma. You may speed up the movement of the cursor by pressing the REPT (REPEAT) key simultaneously with the back arrow key.
- (c) Type a semicolon. The semicolon will overwrite and replace the comma.
- (d) Press RETURN.

If you now list 40 you will find that line 40 contains a semicolon instead of a comma and everything else is as before.

You may insert extra characters into a line or delete characters from a line by typing **EDIT** and taking action as follows.

- (a) To *delete* a character, position the cursor over the character and type CTRL/S, i.e. press the CTRL key and the S key simultaneously. The character involved will disappear and the characters to the right will move left one place to fill the gap.
- (b) To *insert* a character, position the cursor where you want to insert and press CTRL/A, i.e. CTRL and A keys simultaneously. The symbols to the right of the cursor will move one place to the right, leaving a gap where you may insert a new symbol. Simply type the desired symbol and press RETURN.

You may of course insert or delete several characters in one attempt by repeating the actions outlined above.

You should spend some time practising the editing operations, as you will find that facility in using EDIT is extremely helpful as you proceed through the course. (Even *you* are bound to make a few mistakes now and again!)

If you enter the command EDIT on its own (i.e. without a line number) the COMAL system assumes you wish to edit the whole program and will begin by placing the first line of the program on the screen with cursor positioned at its right hand end. You may make any changes you wish. When you press RETURN the system will display the next line, and so on. If you do not wish to change a line just press RETURN and the next line will be displayed. When you want to finish editing press the ESC key.

# Appendix 6

## Notes on the Commodore-64 and BBC versions of COMAL

One of the major advantages of COMAL compared with many other computer languages is its strong degree of standardization. This means that most programs will run on different systems without any change and only very minor changes are needed in the rest. The major differences (at the time of writing) arise in the implementation of graphics. Commodore-64 has perhaps the most complete graphics system but many of its simple features can be achieved (indirectly) on the BBC. This appendix describes briefly some of the principal differences between the Commodore-64 and BBC versions and the Apple version.

### COMMODORE-64

Some extra words are reserved words and may not be used as variable names,

- e.g.            STATUS (see Program 23)  
                BACK (see Program 61)  
                RIGHT (see Program 83)  
                PASS (see Program 107)

Instructions should not extend beyond about two screen lines. This means you may have to be careful about INPUT lines with messages. If necessary, you should use a combination of PRINT and INPUT.

- TAB := is replaced by ZONE  
e.g.            TAB := 6 is replaced by ZONE 6 (see Program 4)

The use of AUTO is terminated by pressing RETURN twice rather than ESC.

### Strings

In specifying substrings of a string, a comma is replaced by a colon,

- e.g.            NAME\$ (1,5) is replaced by NAME\$ (1:5)  
                (see Program 62, etc.)

It is possible to assign to a substring,

- e.g.            ITEM\$ (8,12) := "BREAD"

This is a great convenience!

## Files

In the usual one-disk-drive Commodore-64 system, the drive is numbered 0. This may be specified in file names,

e.g. OPEN FILE 2, "0 : STOCK", WRITE (see Program 112)

In opening a random file, the comma is omitted between the word RANDOM and the record size,

e.g. OPEN FILE 3, "LIBRARY", RANDOM 40 (see Program 116)

## FOR loops

To achieve uniformity with other structures, FOR loops are ended with ENDFOR,

e.g. FOR C := 1 TO 5 DO  
      PRINT "HELLO"  
      ENDFOR C

However, you may type in NEXT C and the system will automatically change it to ENDFOR C.

## Function Definitions

There are a few differences to be observed when defining functions. An example will illustrate these (see Program 84):

```
FUNC SUM (X)
      RETURN X + 12
ENDFUNC SUM
instead of
DEF FNSUM (X)
      FNSUM := X + 12
ENDDEF FNSUM
```

### Note

- (1) The heading is changed: DEF is replaced by FUNC, and the name of the function does not start with FN.
- (2) The ending is changed: ENDDEF is replaced by ENDFUNC.
- (3) Instead of assigning a value to the function name, that value is returned to the calling program by using RETURN.

## **Missing Functions**

Some commonly used functions may not be available. It is usually possible to write your own. In case you are feeling a bit lazy, here are some examples:

### **ROUND**

For ROUND(N) use INT(N + 0.5).

### **CLEAR**

In order to clear the screen in text mode, use

PRINT CHR\$(147)

### **CURSOR**

The following procedure can achieve the effect of the CURSOR function:

```
PROC CURSOR (X,Y) CLOSED
    PRINT CHR$(19) // BRING CURSOR HOME
    FOR P := 1 TO Y DO
        PRINT
    ENDFOR P
    PRINT TAB(X),
ENDPROC CURSOR
```

### **VAL**

The following function will simulate the function VAL:

```
FUNC VAL (STRING$) CLOSED
    L := 0
    LENGTH := LEN(STRING$)
    WHILE STRING$(L)<>"." AND L < LENGTH DO
        L := L+ 1
    ENDWHILE
    IF L < LENGTH THEN
        MARK := L- 1
    ELSE
        MARK := L
    ENDIF
    V := 0
    FOR P := 1 TO MARK DO
        V := 10 * V + ORD (STRING$(P)) - ORD ("0")
    ENDFOR P
    E := 0
```

```

FOR P := L + 1 TO LENGTH DO
    E := 1
    V := V + (ORD(STRING$(P)) - ORD ("0"))/10↑E
ENDFOR P
RETURN V
ENDFUNC VAL

```

### **STR\$**

In order to achieve the effect of the STR\$ function, we have to resort to a little cunning. The simplest thing is to use a procedure instead of a function and to open the screen as a file. The number to be converted to a string is printed on the screen and then read back as a string. Thus

```

PROC STR (NUMBER, REF STRING$)
    PRINT NUMBER, CHR$(145)//MOVE BACK UP ONE LINE
    OPEN FILE 99, " ", UNIT 3, READ
    INPUT FILE 99: STRING$
    CLOSE FILE 99
ENDPROC STR

```

## **BBC**

Many of the powerful facilities of BBC BASIC are available from COMAL, e.g. MODE, FX, GET, GET\$, INKEY, INKEY\$, POINT, SOUND, ENVELOPE, VDU, etc. The interested student should study the BBC manuals.

GOTO line number is not allowed; GOTO a label is, e.g. GOTO START

Strings have a default value of 40 characters. They need not be dimensioned unless they will have more than 40.

### **Some omissions**

```

GOSUB
LOOP... EXIT ... ENDLOOP
CALL
TRAP ERR
TRAP ESC

```

functions such as BSTR\$, BVAL, FRAC, IVAL,  
ROUND, SPC\$, TRUNC

CAT  
DOWNT0  
MAT  
PAGE  
RANDOM  
System variables: ERR, ERRTEXT\$, ESC

## Graphics

The BBC's high-resolution colour graphics are fully available from COMAL. Although the turtle graphics commands are not implemented, it is easy to emulate the turtle graphics by simple procedures. In fact most of the work can be done by just two procedures, e.g. FORWARD and RIGHT. The following two procedures might serve:

```
PROC RIGHT (ANGLE)
    ORIENTATION := ORIENTATION-ANGLE
ENDPROC RIGHT

PROC FORWARD (DISTANCE)
    X := X + DISTANCE * COS (C * ORIENTATION)
    Y := Y + DISTANCE * SIN (C * ORIENTATION)
    DRAW (X,Y)
ENDPROC FORWARD
```

To support these procedures, ORIENTATION would need to be given a starting value (say 90) and be available for global use. Similarly for X and Y, with starting values of say 600 and 500 respectively. C will be defined as  $\pi/180$  for converting from degrees to radians.

# Index

- abacus 118  
ABS 129, 172, 331  
accumulator 111  
ACE 123  
action 26, 27, 37, 38, 49, 51,  
  62, 93, 150, 178, 186, 188,  
  201, 214, 323  
action on privacy 323  
actual parameter 186, 188,  
  189, 206, 215, 249, 250  
ADA 119  
address 105, 111, 334, 336  
Aiken 122  
algorithm 3, 86, 181, 190,  
  210, 254, 256  
ALU 110  
ambiguity 3  
analytical engine 119, 120  
AND 92, 141  
anonymity 320  
application programs 113,  
  327  
archaeology 313  
argument 163–68, 170, 184,  
  186, 192, 193  
arithmetic priority 21, 22  
arithmetic symbols 17  
array 131–37, 147, 148, 151,  
  155, 158, 219, etc.  
art 256, 313, 314  
ASCII 97, 103, 104, 331, 332  
assembler 114, 127  
assign 19, 37  
asterisk 17, 18, 20, 21, 22, etc.  
Atanasoff 122  
ATN 331  
authorisation 322, 323  
AUTO 10, 14, 337, 338, 346  
automatic 10, 119, 122, 123,  
  180, 185, 206  
  
Babbage 119  
BACK 265, 346  
BACKGROUND 269, 285  
backing store 103, 106  
band 108  
  
banking 305, 306  
BBC 349–50  
binary 103, 113  
binary search 220, 221, 223  
bits 103, 105, 106, 108  
blocks 38, 106, 108, 131  
Boole 120, 140  
BORDER 269  
BRACKET ERROR 342  
Briggs 119  
BSTR\$ 331  
bubble sort 224–27  
buses 110, 111  
bug 327  
BVAL 331  
byte 105, 248, 289  
  
CAI 301  
calculator 118, 119  
call 184, 186, 188, 191, 192,  
  193, 201, 205, 206, 210,  
  212, 215, 216, 219, 223, 246  
carbon ribbon 102  
card reader 97, 98, 115  
cascade 42  
CASE 49, 51, 54, 55, 56, 339  
cassette tape 106  
CAT 26  
chain printer 102  
character 4, 11, 13, 26, 54, 59,  
  96–101, 116, etc.  
cheques 269  
choice 36, 39, 42, 45, 88  
CHR\$ 289–90, 294, 348  
clarity 3, 11, 134, 194, 255,  
  260  
CLEAR 12, 164, 264, 348  
close 213, 214, 217, 219  
CLOSED 213, 217  
closing a file 236  
CML 301  
code 97, 98, 103, 113, 114,  
  123, 127, 322, 323  
COLOUR 269, 294, 297  
column 98, 101, 147, 149,  
  151, 153  
  
colossus 123  
COMAL 3, 4, 6, 10, 11, 14,  
  15, etc.  
comma 6, 7, 12, 13, 21, 28, 30,  
  55, 80, 157, 163, 216, 239  
comment 20, 27, 254–56  
commerce and industry 305  
Commodore-64 13, 346–49  
compiler 113, 114, 124, 127,  
  328  
communications 325  
compound statement 71  
CON 24  
concatenation 31  
condition 61, 85–88, 92, 141,  
  160  
conducting highways 110  
confidentiality 115, 320  
consent 319  
constant 8, 10, 180, 183, 249  
control unit 110  
control variable 71, 73, 74  
copy 206, 211, 216, 305, 338  
COS 115, 169, 170, 172, 184,  
  191, 331  
counter 73, 79, 81, 219  
CP/M 115  
CPU 109–111, 115, 124  
CRSR 13  
CTRL key 13, 14, 345  
cursor 12, 13, 14, 116, 166,  
  168, 264, 337, 339, 345  
CURSOR 263, 348  
  
daisywheel printer 101  
data 2, 7, 8, 11, 12, 59, 60, 62,  
  77, etc.  
DATA 12, 135, 141, 142, 152,  
  160, etc.  
data banks 311, 312, 324, 325  
data base 117  
DATACOLLISION 299  
data preparation staff 328  
data processing 2, 329

Data Protection Committee 324  
debugging 15, 116, 254, 255, 256, 327, 328  
decrement 139  
DEF 184, 347, 350  
DEFINE 290  
define 185-89, 193, 201  
default 236  
DEL 14, 27, 344  
DELAY 272  
DELETE 27, 344  
design 117, 120, 123  
Difference Engine 119  
DIM 11, 130, 131, 137, 158, 248  
direct access device 107  
direct file 247  
disk 24, 26, 27, 106, 107, 115  
disk band 108  
disk drive 25, 107, 115, 236, 242, 303, 336, 337  
disk pack 108  
disk sector 108  
DIV 18, 22, 221  
document 255  
dollar sign 11, 13, 37, 55, 63, 90, 138, 142-44, 158, etc.  
DOS 115  
DOWNTO 73  
DP manager 329  
drive 347  
drive shaft 108  
drum printer 102  
dry run 133, 255  
dummy argument 186  
dynamic 219  
  
Eckert 122, 123  
EDIT 13, 14, 300, 345  
editor 116  
education work and leisure 301-16, 325-26  
EDVAC 123  
electronic 122, 312, 318, 322, 325  
ELIF 44-49, 55, 56  
encode 103  
END 5, 10  
ENDCASE 51, 339  
ENDDEF 184, 347  
  
ENDFOR 347  
ENDFUNC 347, 349  
ENDIF 38, 44, 45, 92, 138  
END-OF-LINE EXPECTED 340-42  
ENDPROC 201, 348  
ENDWHILE 88  
ENIAC 122, 123  
EOD 141, 142  
EOF 239, 250  
erase 27, 216  
ERR 252  
errors 5, 13, 14, 17, 45, 46, 48, 114, 116, 339-44, etc.  
ERRTEXT\$ 331  
ESC key 5, 10, 14, 300, 301, 345  
EXEC 201, 202, 206, 335  
execute 17, 45, 58, 63, 71, 72, 88, 113, 129, 180, 198, etc.  
EXP 172, 173, 331  
  
Falcon 120  
factor 202, 203  
factorials 63, 64  
FALSE 141, 142, 229, 239  
ferrite core 124  
field 235, 239  
field width 6  
FILES 347  
file 1, 234, 235, 239-52, etc.  
FILL 298  
first generation computers 123  
floppy disk 107  
FN 184, 191-93  
FOR 347  
FOR ... NEXT 70, 71, 73, 78, 80, 85, 133, 139, 142, 165, 170, 226, 229, 256  
formal parameter 186, 188, 189, 206, 211, 249, 275  
FORMAT ERROR 340  
formatting 116, 327  
FORWARD 264  
fourth generation computers 126  
FRAC 331  
FRAME 298  
freedom, individuality and privacy 319, 320  
  
FULLSCREEN 298  
FUNC 348  
function 30, 65, 116, 138, 162-64, 167, 172, 173, 175, 179, 180, 184-94, 205, 208, 209, 213, 223, etc.  
  
gap 106  
GLOBAL 188, 189, 213  
Government and Public Services 315  
GRAPHICS 264  
graphics 100, 117, 256, etc.  
Graphics mode 264, 272  
graph-plotter 101  
grid 147, 287-88, 295  
  
hard copy 101, 272  
hardware 96-112, 113, 126  
Harvard Mark I 122  
heat printer 102  
HIDESPRITE 299  
HIDETURTLE 266, 287  
high-resolution 289  
holes 97, 98, 120  
Hollerith 121  
HOME 264, 282  
hopper 98  
  
IBM 121-25  
ICL 308  
IDENTIFY 290  
identifier 9, 10, 11, 90, 134, 153, 158, 186, 215, 237, 239  
IF ... THEN... ELSE 36-38, 44, 138  
ILLEGAL CHARACTER 340  
immediate access store 105  
immediate execution 4, 17, 18, 28  
IN 210  
increment 65, 71, 74, 133, 138, 139, 151  
indent 38, 40  
index 130, 147, 153, 251  
index register 111, 124  
information processing 2, 109, 311  
information retrieval 111, 117, 235, 313-16

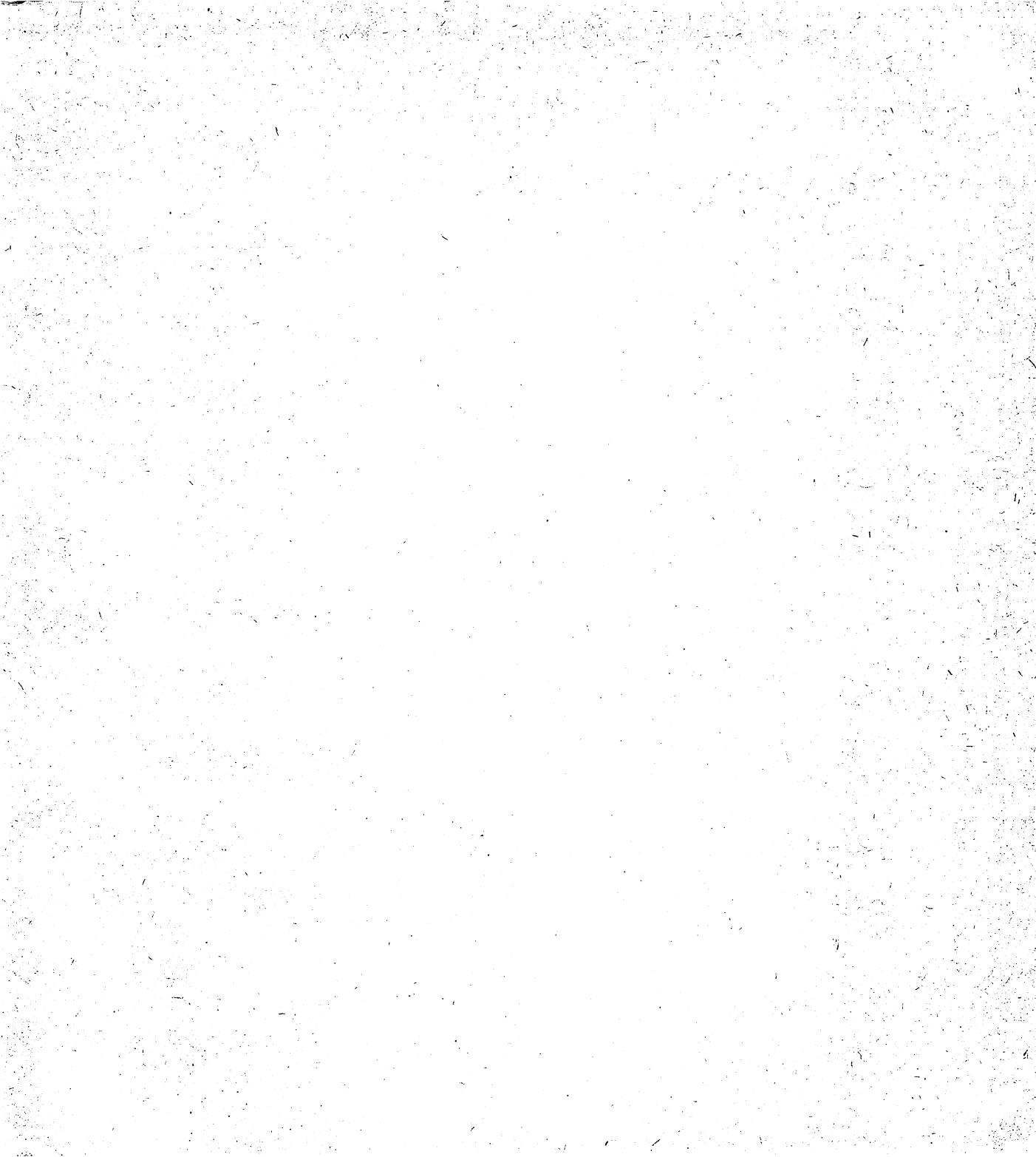
INIT 301  
 initialise 71, 138, 139, 157  
 input 2, 9, 11, 19, 21, 45, 48,  
   55, 67  
 INPUT 2, 8, 9, 11, 12, 19, 20,  
   46, 59, etc.  
 instruction 2, 3, 5, 6, 8, 12, 14,  
   15, etc.  
 instruction register 111  
 insurance 305, 307  
 INT 168, 332  
 integrated circuits 125  
 interpreter 113, 114  
 inverted comma 7, 9, 19, 30,  
   etc.  
 iteration 58, 60–66, 69, 71,  
   78, 82, 85, 92, 93, etc.  
 IVAL 332  
 -Jacquard 121  
 keyboard 3, 11, 38, 54, 96–98,  
   100, 113, 115  
 language and literature 313  
 law 310  
 LEFT 265  
 Leibniz 118  
 LEN 138, 332  
 librarian 330  
 licensing 324  
 light pen 100  
 light-sensitive cells 98  
 Lindop 324  
 line-printer 101, 115, 336  
 LIST 12, 334, 337  
 LOAD 26, 127, 334, 337  
 LOCAL 189, 212, 214  
 location 9, 19, 105, 106, 111,  
   131, 133, 134, 137, 211, 212  
 LOG 332  
 logical 140, 141, 229, 256  
 logical errors 14, 15, 116, 339  
 logical variable 272  
 loop 61, 62, 63, 64, 65, 67, 70,  
   71, 72, 74, 78–82, etc.  
 Lovelace, Ada, 119  
 LSI 126  
 Ludgate 120  
 magnetic cores 105, 124  
 magnetic disk 106, 107, 116,  
   235, 328  
 magnetic tape 106, 124, 235,  
   307, 328  
 main-frame 107  
 management 243, 305, 307  
 MAT 335  
 market research 305  
 matrix printer 101, 102  
 Mauchly 122, 123  
 MBR 111  
 mean 58, 59, 60, 128, 129,  
   130, 133  
 medicine 274  
 medium 235, 311, 325  
 memory 8, 9, 19, 24, 26, 27,  
   97, 98, 100, 103, 105, 106,  
   110, 111, etc.  
 Memory Address Register  
   111  
 Memory Buffer Register 111  
 merging 117  
 message 19, 27  
 MICR 100, 306  
 micro chip 126  
 microfiche 102, 106, 306  
 microfilm 102  
 microprocessor 110, 126, 326  
 microprogramming 125  
 mini 125  
 MOD 18, 22, 93  
 monitor 115  
 MOS 105  
 movement 286  
 MOVETO 283  
 multiple selection 44, 49, 55  
 multiprogramming 115, 116  
 music 314  
 Napier 119  
 nesting 75, 78, 80, 82, 149,  
   151, 152, 157, 343–44  
 networks 303  
 NEW 8, 12, 337  
 NEXT 347  
 NOT 141  
 NOT A STATEMENT 343  
 OCR 99  
 one-dimensional 132, 145,  
   157, 216  
 open 213, 214, 236  
 OPERAND EXPECTED  
   340  
 operator 18, 21, 22, 75, 93,  
   141, 162, 328  
 operating system 114  
 OR 141  
 ORD 332  
 order 21  
 OTHERWISE 51–54  
 OUT OF STORAGE 343  
 output 2, 4, 5, 6, 19, 28, 29, 30,  
   46, 96, 100, 108, etc.  
 package 117  
 palindrome 139  
 paper tape 98, 99, 102, 106,  
   235, 328  
 parabola 293  
 parameter 163, 186, 188, 189,  
   204, 205, 206, 211, 215,  
   216, 219, 223, 226, 246,  
   249, 272, 281, etc.  
 Pascal 118  
 pass 224, 225, 226, 229  
 PASS 346  
 PASS machine 306  
 payroll 1, 305, 308  
 PC 111  
 PEEK 332  
 PENCOLOR 269  
 PENDOWN 298  
 PENUP 298  
 peripheral 102, 115  
 photo-electric cells 98  
 PLOT 286  
 PLOTEXT 298  
 plotting 264  
 POKE 332  
 polyester 106  
 polynomial 74  
 POS 332  
 position 172  
 precedence 22  
 pre-defined functions 185,  
   186, 331  
 primary memory 103  
 PRINT 4, 5, 6, 7, 9, 17, 19, 20,  
   28, etc.  
 printer 101, 113, 115, 165,  
   303  
 printout 328

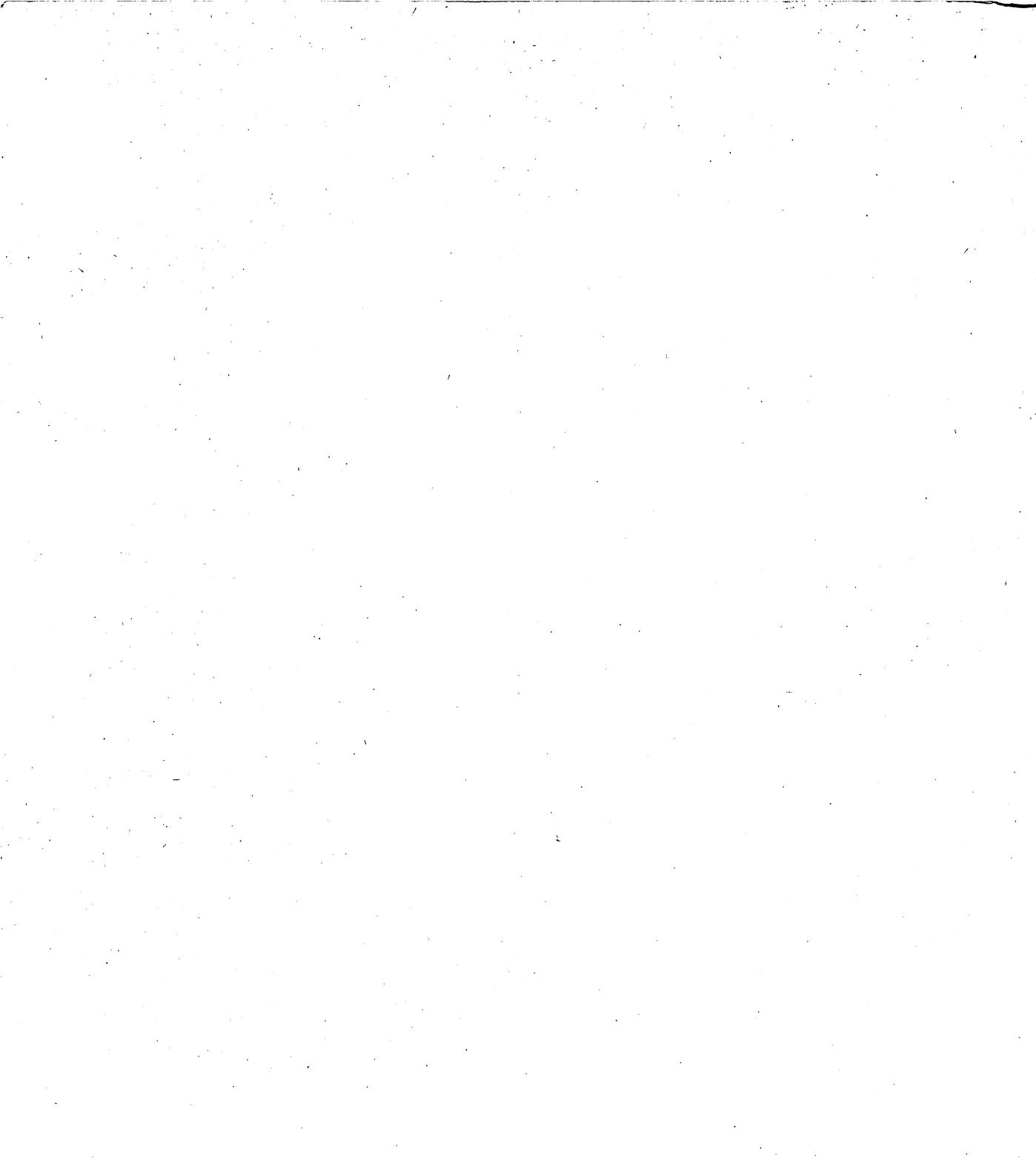
PRINT USING 239, 336  
PRIORITY 299  
priority in arithmetic 21, 22  
PROC 201, etc.  
procedure 197-219, 223, 225,  
  229, 230, 264, 272, 273,  
  281-82, etc.  
process 2, 3, 96, 162, 178, 243,  
  244  
process automation 310  
Program Counter Register  
  111  
programmer 2, 32, 115, 119,  
  126, 255, 327, 328  
programming 2, 4, 36, 327  
protect 116, 323, 324  
pulses 97, 100, 103  
punched cards 97, 98, 102,  
  106, 120, 122, 235, 328  
question mark 8, 9, 18, 98  
queue 115  
quote marks 7, 9, 19, 30, 164,  
  etc.  
radian 167, 168, 170, 331, 333  
RAM 105, 106  
RANDOM 247, 347  
random 65, 66, 175, 178, 179,  
  180, 247, 249, 250, 297  
READ 11, 12, 135  
record 101, 235, 237, 247,  
  248, 251, etc.  
recursion 191, 192, 193, 194,  
  223, 224, 255, 273  
recursive procedure 273, 277  
REF 211, 212, 214  
reference parameter 212,  
  215, 219, 246, 249, 250  
refinement 29  
registers 111  
remark 18, 255  
RENUM 13  
REPEAT ... UNTIL 61, 65,  
  67, 78, 85, 86, 88, 130, 144,  
  178, 229  
repetition 271  
REPT 345  
REPT key 13  
reserve 131  
reserved words 346  
retrieve 26, 105, 235, 248, 249  
RETURN 345-47  
return key 4, 10, 13, 14, etc.  
RIGHT 265, 266  
RND 65, 115, 175, 180, 184,  
  205, 332  
robotics 310, 312  
rogue value 59, 62, 129, 141  
ROM 105, 106, 113  
ROUND 30, 332, 348  
RUN 5, 58, 337, 338  
row 97, 147, 149, 151, 153,  
  154, 157  
SAVE 26, 334, 337  
schedule 115  
Scheutz 120  
Schichard 118  
Science and Technology 312  
screen 4, 9, 10, 12, 17, 18, 19,  
  81, 96, 100, 116, etc.  
screen editor 116  
search 160, 200, 216, 220,  
  223, 244, 250  
secondary memory 103, 106  
second generation computers  
  124, 125  
sector 108  
security 322  
selection 36, 38, 44, 45, 65, 91,  
  93, 136  
semiconductor 105, 126  
semicolon 6, 13, 28, 149, 151  
sensitive data 322  
sequence 36, 37, 58, 85, 93,  
  131, 200, 249  
sequential file 247  
SET XY 298  
SETGRAPHIC 264  
SETHEADING 283  
SGN 333  
SHIFT key 4  
SHOWSPRITE 299  
simplicity 194, 251, 255  
simulation 301, 302, 308, 312  
SIN 115, 167, 168, 170, 172,  
  173, 184, 191, 205, 333  
software 113-17  
soroban 118  
sorting 117, 218, 219, 220,  
  224, 227, 230  
SPC\$ 333  
spiral 275, 278  
SPLITSCREEN 298  
sprite 287-98  
SPRITECOLLISION 299  
SPRITEPOS 290  
SPRITESIZE 292  
SQR 333  
static 219  
STATUS 346  
STEP 73, 74, 76, 77, 83  
stockbroker 305, 307  
stock control 117, 237-46  
STOP 24  
store 121, 123, 235, 313- 315  
STR\$ 296, 349  
string 7, 10, 30, 31, 50, 55, 63,  
  80, 128, 131, 137-40, 157,  
  165, 198, 208, 209, 229, etc.  
STRING NOT TERMINATED 341  
structure diagram 24, 29, 49,  
  60, 65, 72, 79, 85, 136, 197,  
  198, 208, 210, 215, 230, etc.  
subroutine 124, 298  
subscript 130, 131, 147, 153  
substring 142, 144, 158, 198,  
  210, 346  
supervise 114, 115  
swap 153, 154, 155, 224, 225,  
  227, 229  
symbol 4, 90, 98, 103, 207,  
  208, 248  
SYNTAX ERROR 340-43  
syntax errors 14, 340-43  
systems analyst 326, 327  
system commands 6  
system software 113, 114, 124  
TAB 6, 7, 163, 165, 166, 170,  
  333, 346  
tabulating 121  
TAN 296  
teletype 99  
temporary variable 229  
terminal 114, 115, 305, 315,  
  325  
terminate 178  
test 22, 86, 87, 88, 92, 117,  
  172, 178, 207, 208, 255, 328  
text editor 116, 313

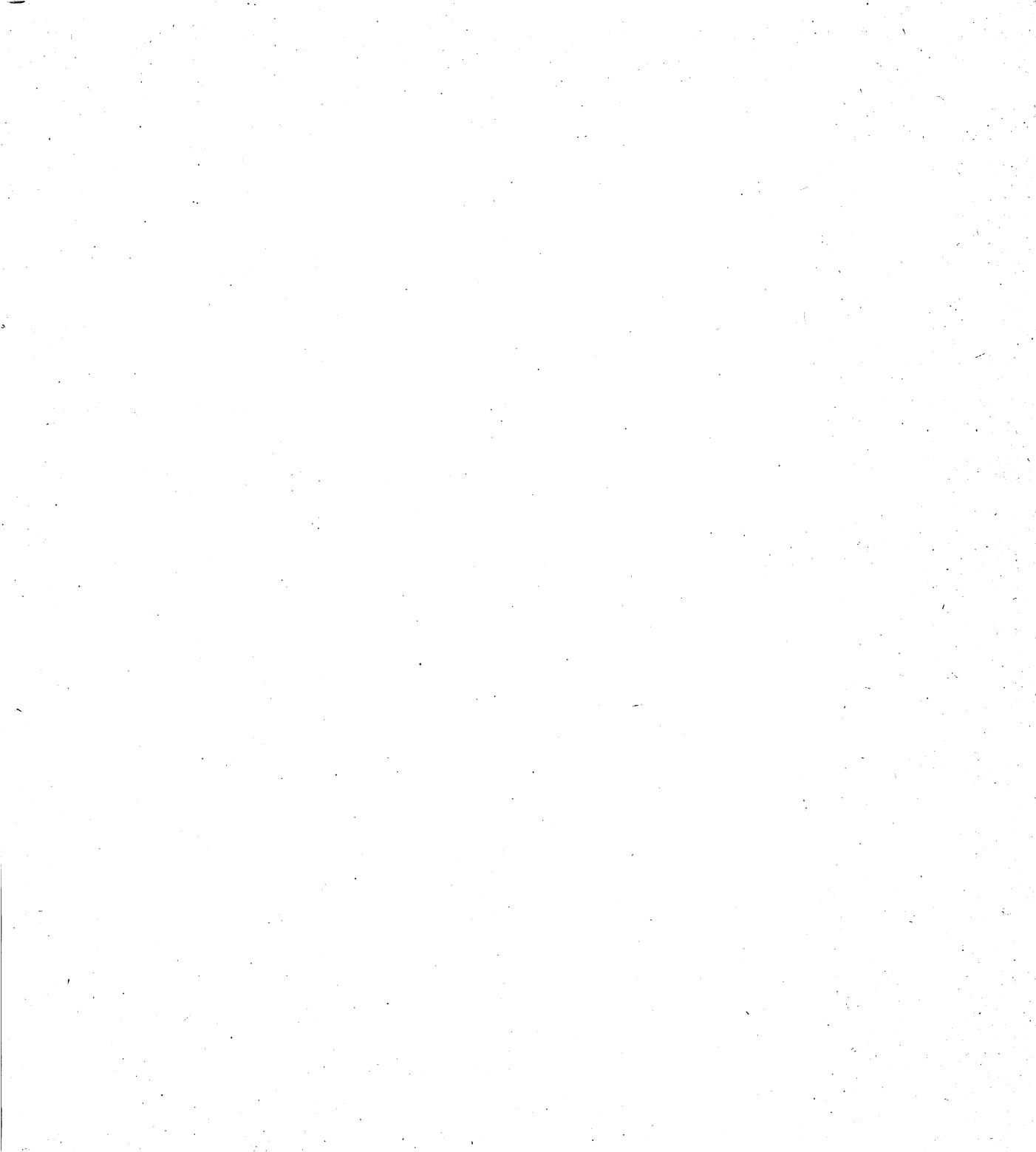
text mode 264  
third generation computers 125  
timetabling 117, 301, 303  
translation program 113  
transistor 124, 127  
transform 162, 163  
TRAP ERR 252  
traversal 70, 81, 82  
TRUE 141, 142, 229  
TRUNC 333  
Turing 121, 123  
turtle 264  
turtle graphics 264  
two-dimensional array 147, 157  
TYPE CONFLICT 340-41  
  
UNDEFINED VARIABLE 343-44

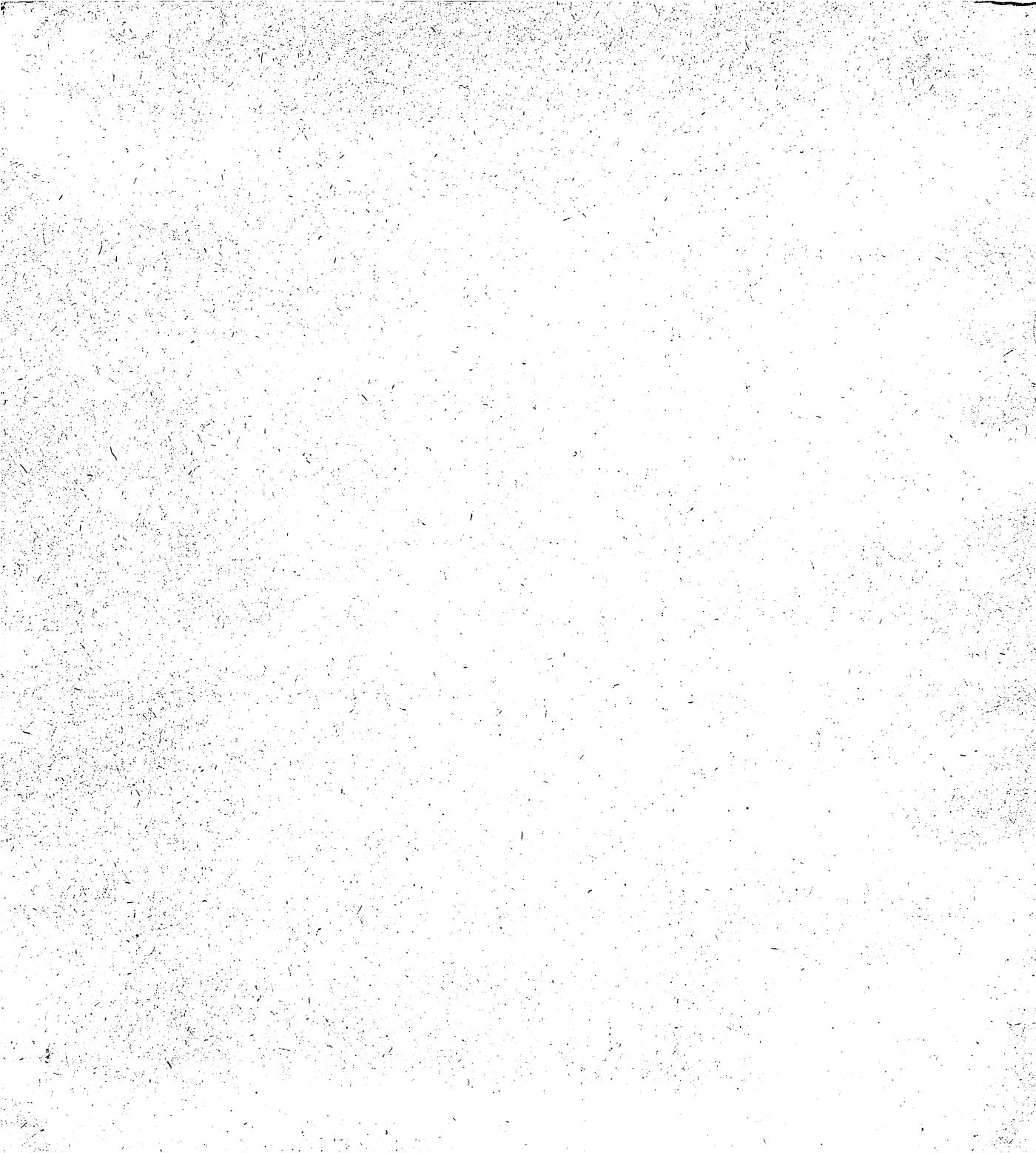
user-defined function 185, 187, 191  
utility 115, 116  
  
VAL 208, 333, 348  
value parameter 205, 212, 216, 249  
valves 122-24  
Van Rijsbergen 316  
variable 8, 9, 10, 37, 60, 63, 78, 83, 90, 130, 133, 137, 140, 142, 150, 165, etc.  
VARIABLE EXPECTED 340  
VDU 4, 96, 100, etc.  
Von Neumann 123  
  
WHEN 51-55  
WHILE 87-89, 92, 93, 138, 178  
  
Wilkes 124  
word 105  
word processing 305, 309  
work-space 8, 12, 334, 336, 337  
WRITE 347  
writing to a file 236  
WRONG TYPE 341  
  
Xerographic printer 102  
  
Younger 323, 324  
  
zone 6, 7, 163  
Zuse 121













**The Educational Company**