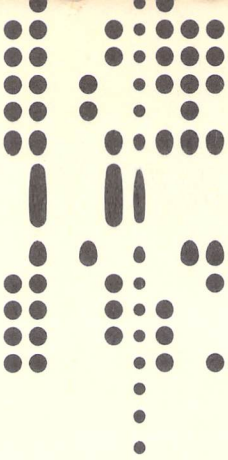
Ellis Horwood Publishers
COMPUTERS AND THEIR APPLICATIONS 17

Structured Programming With Comal

ROY ATHERTON



Computers and Their Applications **17**

Series Editor:
BRIAN MEEK, Director, Computer Unit,
Queen Elizabeth College, University of London

STRUCTURED PROGRAMMING WITH COMAL

ROY ATHERTON, Director, Computer Education Centre,
Bulmershe College, Reading

This book draws together, "under one roof", the simple style and ease of the programming language BASIC with the elegantly sophisticated structures and clear appearance of the more recent COMAL. The definitive spirit which these languages share is their virtue of simplicity in manipulation.

The author offers a suitable introduction to programming and problem analysis either for the beginner, or to those who know BASIC. He organises his material into a learning order, tested over the years by both students and teachers, comparable to a spiral in which topics are treated piece-by-piece, introducing new material at the appropriate moment. Whilst it is not a set of lecture notes, it compromises by carefully introducing new material in each chapter, thereby not exhausting the reader; information on these new topics is built upon as the book progresses, thus saving the reader the common problem of referring back.

Roy Atherton's treatment is logical, maintains development and advancement at a steady level, avoiding unnecessarily rigorous explanations and syntactic detail. He approaches syntax rules and structures in a creative, "free-wheeling" and imaginative way to be made rigorous and comprehensive at a later stage.

Readership: All teachers and students of computing science in colleges, universities and schools, non-specialist users of computers in universities, students in sixth forms, personal computer users, small computer users in business, academic institutions, industry, government, etc.

STRUCTURED PROGRAMMING WITH COMAL

THE ELLIS HORWOOD SERIES IN COMPUTERS AND THEIR APPLICATIONS

Series Editor: BRIAN MEEK

Director of the Computer Unit, Queen Elizabeth College, University of London

The series aims to provide up-to-date and readable texts on the theory and practice of computing, with particular though not exclusive emphasis on computer applications. Preference is given in planning the series to new or developing areas, or to new approaches in established areas.

The books will usually be at the level of introductory or advanced undergraduate courses. In most cases they will be suitable as course texts, with their use in industrial and commercial fields always kept in mind. Together they will provide a valuable nucleus for a computing science library.

INTERACTIVE COMPUTER GRAPHICS IN SCIENCE TEACHING
Edited by J. McKENZIE, University College, London, L. ELTON, University of Surrey, R. LEWIS, Chelsea College, London.
INTRODUCTORY ALGOL 68 PROGRAMMING
D. F. BRAILSFORD and A. N. WALKER, University of Nottingham.
GUIDE TO GOOD PROGRAMMING PRACTICE
Edited by B. L. MEEK, Queen Elizabeth College, London and P. HEATH, Plymouth Polytechnic.
CLUSTER ANALYSIS ALGORITHMS: For Data Reduction and Classification of Objects
H. SPÄTH, Professor of Mathematics, Oldenburg University.
DYNAMIC REGRESSION: Theory and Algorithms
L. J. SLATER, Department of Applied Engineering, Cambridge University and H. M. PESARAN, Trinity College, Cambridge
FOUNDATIONS OF PROGRAMMING WITH PASCAL
LAWRIE MOORE, Birkbeck College, London.
PROGRAMMING LANGUAGE STANDARDISATION
Edited by B. L. MEEK, Queen Elizabeth College, London and I. D. HILL, Clinical Research Centre, Harrow.
THE DARTMOUTH TIME SHARING SYSTEM
G. M. BULL, The Hatfield Polytechnic
RECURSIVE FUNCTIONS IN COMPUTER SCIENCE
R. PETER, formerly Eötvos Lorand University of Budapest.
FUNDAMENTALS OF COMPUTER LOGIC
D. HUTCHISON, University of Strathclyde.
THE MICROCHIP AS AN APPROPRIATE TECHNOLOGY
Dr. A. BURNS, The Computing Laboratory, Bradford University
SYSTEMS ANALYSIS AND DESIGN FOR COMPUTER APPLICATION
D. MILLINGTON, University of Strathclyde.
COMPUTING USING BASIC: An Interactive Approach
TONIA COPE, Oxford University Computing Teaching Centre.
RECURSIVE DESCENT COMPILING
A. J. T. DAVIE and R. MORRISON, University of St. Andrews, Scotland.
PASCAL IMPLEMENTATION: The P4 Compiler and Compiler and Assembler/Interpreter
S. PEMBERTON and M. DANIELS, Brighton Polytechnic
MICROCOMPUTERS IN EDUCATION
Edited by I. C. H. SMITH, Queen Elizabeth College, University of London
AN INTRODUCTION TO PROGRAMMING LANGUAGE TRANSITION
R. E. BERRY, University of Lancaster
ADA: A PROGRAMMER'S CONVERSION COURSE
M. J. STRATFORD-COLLINS, U.S.A.
STRUCTURED PROGRAMMING WITH COMAL
R. ATHERTON, Bulmershe College of Higher Education
SOFTWARE ENGINEERING
K. GEWALD, G. HAAKE and W. PFADLER, Siemens AG, Munich

STRUCTURED PROGRAMMING WITH COMAL

ROY ATHERTON, B.Sc., M.Tech., M.B.C.S.
Director of Computer Education Centre
Bulmershe College of Higher Education
Reading

ELLIS HORWOOD LIMITED
Publishers · Chichester

Halsted Press: a division of
JOHN WILEY & SONS
New York · Brisbane · Chichester · Toronto

First published in 1982
and Reprinted in 1985 by

ELLIS HORWOOD LIMITED
Market Cross House, Cooper Street, Chichester, West Sussex, PO19 1EB, England

The publisher's colophon is reproduced from James Gillison's drawing of the ancient Market Cross, Chichester.

Distributors:

Australia, New Zealand, South-east Asia:
Jacaranda-Wiley Ltd., Jacaranda Press,
JOHN WILEY & SONS INC.,
G.P.O. Box 859, Brisbane, Queensland 40001, Australia

Canada:
JOHN WILEY & SONS CANADA LIMITED
22 Worcester Road, Rexdale, Ontario, Canada.

Europe, Africa:
JOHN WILEY & SONS LIMITED
Baffins Lane, Chichester, West Sussex, England.

North and South America and the rest of the world:
Halsted Press: a division of
JOHN WILEY & SONS
605 Third Avenue, New York, N.Y. 10016, U.S.A.

British Library Cataloguing in Publication Data
Atherton, Roy
Structured programming with COMAL. –
(The Ellis Horwood series in computers and their applications)
1. COMAL (Computer program language)
2. Structured programming
I. Title
001.64'24 QA76.73.C/

ISBN 0-470-27318-6 (Halsted Press – Library Edn.)
ISBN 0-470-27359-3 (Halsted Press – Student Edn.)

Typeset in Press Roman by Ellis Horwood Ltd.
Printed in Great Britain by R. J. Acford, Chichester.

Table of Contents

Author's Preface

The justification for yet another book about structured programming needs to be strong. There are already a surprising number of good books on this theme treating a variety of languages, though most use Pascal. Two arguments seem to be paramount, and both stem from the ideal of ultimate simplicity. First, the essential structures of modern problem analysis and programming are reduced to what seem to be ultimately simple forms: a single, very carefully chosen word to open and close each structure or section of a structure, no begins or ends, and minimal punctuation.

Second, the book compromises with BASIC. It recognises the virtues of this old and popular language: clear notation, simple syntax, easy operating environment, wide availability. The book follows Borge Christensen's view that there is more to be gained from building on the strengths of BASIC than totally rejecting it for the serious deficiencies which have become painfully apparent in the 1970s.

Thus the two qualities of COMAL, the simple structured features and the compromise with BASIC both derive from the idea of making computing as simple as possible for a growing majority of non-specialist users. These ideas were worked out, tested and refined in Danish schools in the 1970s, but there is a universal relevance about simplicity which scientists and artists have pursued across the centuries. It is therefore reasonable to expect, and there is already some evidence for the view, that anyone, young or old, in business, industry, government or education, could benefit from exploring modern structural ideas in this context.

There is a price to pay for sticking with BASIC but after examining some COMAL programs as they are presented here the reader may agree that the price is a small one. Even if one takes one of the most advanced versions such as HP BASIC, which has all the COMAL structures, the difference in appearance of two functionally identical programs is striking. It is achieved by (system-forced) indenting, keywords in upper case and others in lower case and the omission of line numbers. Programs on a screen or printer are not quite like this but the

methods are entirely appropriate for learning and the preparation of programs. The systems provide line numbers and typing can be done entirely in one case without using shift keys.

The effect is that the appearance of programs is both appealing and an aid to perceiving their function. Here the author is indebted to David Barron and Peter Grogono who write Pascal in such excellent style. It is a little surprising that COMAL which is technically little more than a set of extensions to BASIC looks so much better as to seem a quite different language. In spirit and philosophy, of course, it is different sharing with BASIC only the virtue of simplicity in its various manipulations.

The content of the book is what is judged to be a suitable introduction to problem analysis and programming for a complete beginner or for someone who knows BASIC. The organisation of the material is based essentially on a learning order which has been tested over several years with student teachers and others. A good learning scheme is sometimes like a spiral in which topics are treated a bit at a time, returning to previous topics when the time seems right. Thus knowledge of a particular area builds up a bit at a time along with other topics.

On the other hand a book is not a set of lecture notes and it will be used as a sequential medium and as a reference or direct access medium. A compromise has been reached in which some topics are introduced but not exhausted in one chapter. They are 'topped up' later, particularly in Chapter 8. Thus it is sensible for the reader to follow the chapters sequentially without too much referring back.

At each stage enough material is presented to make logical sense and to maintain the development but not too much detail of any particular topic at any stage. For the same reason, while it is hoped that no incorrect statements have been made, rigorous and comprehensive syntactic detail is not necessarily provided in the main treatment of any topic. Even in mathematics, that most rigorous of subjects, proofs and solutions are usually discovered in creative, free-wheeling, imaginative ways and made correct and watertight at a later stage.

The main sequence is determined, after some essential preliminary work in Chapters 1 and 2, by the concepts underlying the major control structures. Within this main sequence the need for such things as logical expressions and the simpler data structures is motivated and treated. First ideas about structure diagrams and top down analysis are also introduced in this way before being given a more extensive treatment in later chapters.

Tony Hoare once observed that he never uses the REPEAT loop because the WHILE loop will do all such functions with greater safety. Rather than teach his (postgraduate) students the slightly simpler REPEAT loop first, he deals only with the WHILE loop on the grounds that students should practice using the WHILE loop on all the simpler cases which would otherwise be 'used up' by the REPEAT loop. He even has doubts about introducing the FOR loop first for the same reason. A glance at the contents list will show that the writer does not base

this book on such arguments, though there is some force in them. However there are instances, in less fundamental and important matters where a simple technique is eschewed in favour of a more complex one on exactly these grounds. Students sometimes need to learn the more versatile techniques on simpler problems where it is judged appropriate and where there is little danger of its interfering with motivation and progress. This book aims to take a student as far as he can travel along the good programming road rather than stand or fall on reaching pre-set, fairly high, standards, but there are some matters, such as searching and sorting, which are so rich in technique that they are treated as early as possible in simple ways using techniques which will carry over to the more difficult (and more useful) cases.

In particular, certain optional notation in COMAL is avoided in the early chapters even though it is briefer. The more experienced student may discover such things later and use or ignore them as he judges fit.

A problem grading system has been introduced (see Appendix). Gradings are based on types of structures needed, complexity and program length. While it is only a rough guide this grading, together with the careful selection of problems to reflect the material of the chapter and certain important ideas and techniques, does imply that readers would benefit from working through the examples more or less in their order of presentation. If that is not possible then an examination of the solutions would be rewarding.

There is some disagreement in computing literature about the names of various sort processes. The author has adopted the names and classification given by Niklaus Wirth in his excellent introduction to sorting: Chapter 2 of *Algorithms + Data Structures = Programs.*

Notes on COMAL syntax are given in Appendix II but readers who require more precise details can refer to the original COMAL-80 specification. The material in the book is largely compatible with Commodore PET and RC Piccolo COMAL. It is also compatible with Metanic COMAL but certain constructions must be done differently. These are detailed in Appendix II. A version of structured BASIC called SBAS for RML 380Z systems is close enough to COMAL to merit serious attention and details of this are given in Appendix IV.

A debt has already been acknowledged to Christensen, Barron, Grogono and Wirth. In addition the writer gratefully acknowledges the benefit he has derived from the works of Tony Hoare, Ken Bowles and, of course, Edsger Dijkstra. There are others, some of whom are mentioned in the body of this book, but the above-mentioned stand out, in the writer's experience as not only great computer scientists, in their various ways, but fine teachers and presenters of material.

The spread of what is, in philosophy and effect, a new language involves an immeasurable effort by enthusiasts and considerable judgement and investment by computer companies. Erling Schmidt, chairman of the Danish Computer Education Society, and his many school and college colleagues have been quite

unstinting in their support and aid to people, in UK and other countries, who wished to evaluate COMAL and see its effects in Denmark. Thanks are also due to Ken McGibbon and John Miskelly of Regnecentralen who invested considerable effort in bringing COMAL to UK schools, and to Nick Green of Commodore who brought the first COMAL 80 into UK.

Many interesting hours have been spent discussing aspects of COMAL and other languages with Chris Robinson of Computer Communications Ltd., and Max Bramer of the Open University, who also read the drafts which inevitably evoked much criticism and discussion. Documentation provided by Len Lindsay of the USA COMAL Users Group has also been helpful. Hugh Williams of Trent Polytechnic made some valuable comments about structured programming in BASIC.

Mrs. Sheila Hopton and Mrs. Stella Savage typed patiently and efficiently the many papers and drafts which eventually became a book. Mrs. Sylvia Pearce checked many of the programs. I thank all these friends and colleagues sincerely but the organisation of material, opinions and errors are, of course, entirely my own.

Roy Atherton, November, 1981.

Have something to say and say it as clearly as you can. That is the only secret of style.

Mathew Arnold

To Green, Welsh, Williams, Ginsburg, Stratford, Slater,
seekers after truth and anything else that came their way.

CHAPTER 1

Experience of Comal

I hear and I forget. I see and I remember. I do and I understand.

Old Chinese Proverb

1.1 INTRODUCTION

It appears that there are two main difficulties which can prevent potentially competent or good programmers from recognising and developing their skills. The first is simply lack of motivation or intellectual courage to make a start – the feeling that it requires rather exceptional skills. As far as getting started and making a little progress is concerned this is not true. If a beginner has some sensible guidance and a friendly system it should be easy to overcome this first obstacle.

The second problem is that when the basic concepts have been understood there is still a great deal to be learned and practised before one can examine a real problem and make sensible judgements about its feasibility of solution in terms of available equipment, how to do it and whether it is worth the effort. The book is intended to provide a foundation of understanding and experience which may be the beginning of such skills. Many, but not everyone, can expect to achieve such success. But almost anyone can realistically expect to make a sensible start and discover that the fundamental techniques are not difficult. This chapter is intended to be a possible first stage in that process.

There is a certain amount of detail to be absorbed in the early stages of computing and it can become boring. For example, there are about fourteen ways in which the operating environment of COMAL helps the user. Such facilities are a most important element in the 'user friendliness' aspect of COMAL. When it is appropriate such features are invaluable. For example, having typed a program it would be very unfortunate if one had to type it all again the next day. The command SAVE enables the program to be stored, more or less permanently on a disc or other medium, ready for equally easy retrieval on another occasion.

The major treatment of such things is postponed until after the reader has had the opportunity of using some of them and can more easily see the value of the others. Generally speaking, the reader is presented with just what is needed at any time to maintain some motivation and keep the development going. This guideline is modified so that unrewarding fragmentation of topics is avoided.

This chapter treats material in such a way that the reader might gain experience of several aspects of computing without any attempt at *comprehensive* understanding of any of them. It is hoped that in this way a feeling of familiarity and ease with a COMAL system can be gained together with a rough general view of how things proceed. When confidence has been acquired the first problem is overcome and the reader will have reached a position from which steady progress to whatever level is desirable and possible can be expected.

No two computer systems are identical and there may be points of detail in particular systems which do not quite match what is written. However, such problems, if they exist, should be minimal because only the most common elementary matters are treated and they have been tested on several different systems.

In Chapter 2 the same ideas are discussed and extended in a framework which aims to classify and exhibit relationships making full use of the conceptual acquisitions of this first chapter.

1.2 GETTING READY

There are two main types of internal computer memory:

(1) ROM Read Only Memory which cannot be altered by the programmer.

(2) RAM Random Access Memory which is not well-named. Its most important difference from ROM is that data can be written into it as well as read from it under the control of a program.

Some systems provide COMAL, which is in essence a large controlling and translating program, in ROM so that one only needs to switch on the computer and start. Others provide COMAL on a magnetic disc or other type of permanent backing store in which case it is necessary to cause the COMAL system to be read from the disc into the main RAM. The procedures for doing this should be simple and details will be provided by the supplier. This chapter assumes that the user has available a computer with COMAL up and running, a keyboard and a visual display screen (Fig. 1.1).

It is worth spending a moment or two examining the keyboard and noting certain keys:

SHIFT This enables upper case letters to be typed or other "shift" symbols.

CAPS LOCK This key 'locks' the keyboard into upper or lower case letters. It does not usually affect other 'shift' symbols.

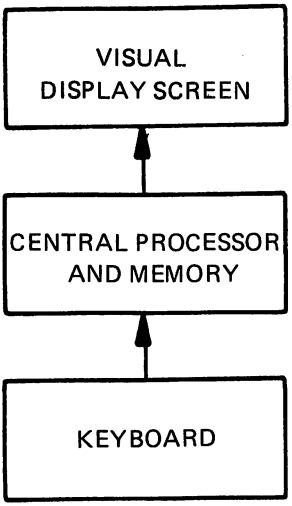


Fig. 1.1 – Minimal computer system.

RETURN or ↵ is usually the 'end-of-message' signal. The computer usually waits for this signal before taking any action in consequence of what has been typed.

ESCAPE can be used to escape from certain situations which have become unwanted. A line of typed information may be wrong and not worth correcting or a program may be doing unexpected things and need to be stopped.

← Move cursor to left.

→ Move cursor to right.

→| Move text to right to create space.

|← Move text to left to delete.

↑ Move Cursor up.

↓ Move cursor down.

CLEAR clears screen

HOME moves cursor to top left position.

DEL moves cursor left deleting character.

* symbol for multiplication

/ symbol for division

∧ symbol for raising to a power, for example, $3 \wedge 2 = 3 * 3 = 9$.

< less than

> greater than

REPEAT causes repetition. On some systems repetition is achieved by simply holding down the relevant key.

There may be other particularly useful keys in some systems and not all of the above are present in all systems.

In the rest of the chapter all material to be typed by the user will be enclosed in a rectangle which will be followed by the computer's response.

1.3 OUTPUT: THE PRINT STATEMENT

Type the following line in which ↵ means the *RETURN* key. Do not omit the quote marks.

```
PRINT "Jane" ↵
```

Jane

Now type the same line with a line number.

```
10 PRINT "Jane" ↵
```

This time there is no response. The computer has simply accepted the line as a correct COMAL statement and stored it. The word RUN will cause the 'program' to be executed.

```
RUN ↵
```

PRINT is a key word of COMAL.
"Jane" is a **character string constant** in which the quote marks indicate the start and end of the string but are not strictly part of it.

A statement without a line number is executed immediately by the computer when the *RETURN* key is pressed. The presence of a line number indicates that the statement is part of a program. The computer checks its syntax and if the line is correct it is stored.

1.4 NUMBERS

A simplified but useful view of a computer operating under the control of COMAL is to imagine a set of 'pigeon holes' which may contain numbers or other data. We enlarge the concept of Fig. 1.1 slightly to show part of the computer memory (RAM) allocated for use by the programmer (Fig. 1.2).

A single statement will name a particular pigeon hole and place a number in it. For example,

```
width := 7
```

has the effect of naming a particular pigeon hole (chosen by the system) and placing in it the number 7. Because the value placed in the pigeon hole named *width* can change as a program is running we say that *width* is the name of a variable or simply that *width* is a variable.

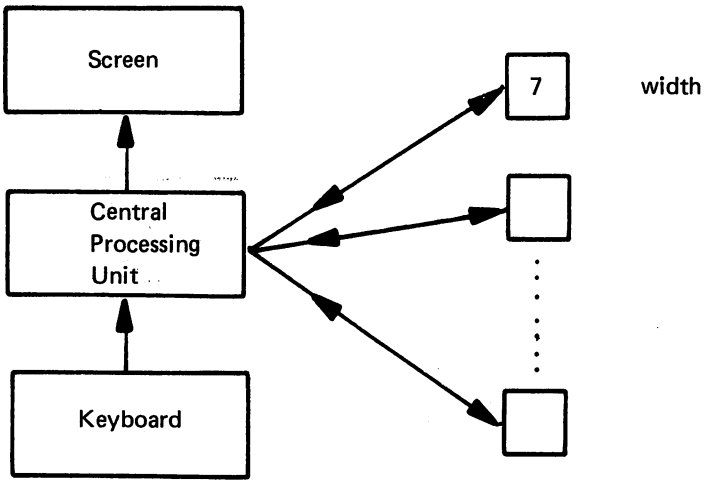


Fig. 1.2 – Computer memory under programmer's control.

Ensure that the program memory is clear by typing:

```
NEW ↵
```

and then type the program:

```
10      width  :=   7 ↵
20      height :=   2 ↵
30      area   :=   width * height ↵
40      PRINT area ↵
RUN ↵
```

14

We now have three variables: *width, height* and *area.* They are **real variables** because the **values** they can take are a very wide range of real numbers, whole or fractional, positive or negative.

Notice the difference between:

```
PRINT "Jane"
```

and

```
PRINT area
```

In the first case a constant string of characters, "Jane" was printed. In the second case the value of a real variable, *area*, was printed. The presence or absence of quote marks determines what happens.

The assignment statement

```
area := width * height
```

is worthy of careful consideration. Firstly it represents a dynamic process:

> "Evaluate the right hand side and make this the value of the variable on the left hand side."

It is not the same as the mathematical equation:

```
area = width * height
```

although the system may accept this for reasons of compatibility with BASIC. An assignment statement is best imagined as:

```
area ← width * height
```

or in the general form:

```
⟨variable⟩ ← ⟨expression⟩
```

Not only will this constantly remind the reader of the important fact that a computer program, which looks static on paper, represents a dynamic process which must be understood as such, but also there will be no confusion caused by such statements as

```
count := count + 1
```

1.5 CHARACTER STRINGS

Just as we can store numbers in pigeon holes which we can name as appropriate, we can also store strings of characters but there are two differences apart from the obvious fact that we would not attempt the same operations on strings as we would on numbers. First, a **string variable** is **tagged** as such by a $ at the end. For example all the following are legitimate string variable names.

```
first$
second$
word$
answer$
```

The second difference arises from the way data is stored internally. The use of a variable name *width* or *area* would cause internal storage to be reserved in a particular way which is the same for all real variables. But string variables can be required to hold values varying from a single character up to whatever the

particular system allows. This may be 72, 256 or almost anything. The programmer must tell the system what he wishes to specify as the maximum length of any string variable he uses by means of a DIM (for dimension) statement.

For example, type

```
NEW ↵
10  DIM first$ OF 4, second$ OF 7, third$ OF 12 ↵
20  first$   := "Good" ↵
30  second$  := "Morning" ↵
40  third$   := first$ + " " + second$ ↵
50  PRINT    third$ ↵
RUN ↵
```

The output will be:

```
Good Morning
```

The DIM statement declares three string variables of different maximum lengths. See Fig. 1.3. If more than one space is placed between the quote marks in line 40 some data will be lost because *third*$ cannot hold more than twelve characters including spaces. Note the use of + in line 40 which simply means 'join to' in this context.

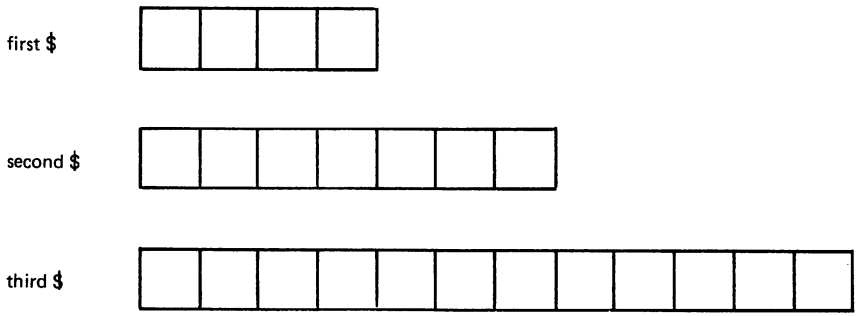


Fig. 1.3 – Computer memory allocated for string variables.

1.6 THE OPERATING ENVIRONMENT

1.6.1 Introduction

The user who has worked carfully through the material presented so far has probably begun to feel some minor but unnecessary frustrations which are more to do with getting things to work than the vocabulary or syntax of COMAL. When Kemeny and Kurtz invented BASIC in the 1960s one of its most striking, and at the time revolutionary, features was its easy operating environment.

Programs need to be designed and written but they also need to be tested, edited, stored and used. While these operation are not strictly functions of a programming language, BASIC demonstrates that, for practical purposes, they can be treated as such. The resulting complete computing system has proved to be very effective in getting simpler types of computing jobs done quickly.

COMAL, which makes concern for the user its first priority, takes the easy operating environment of BASIC and improves upon it. This section covers most of the features that a system is likely to offer under this heading. The user is advised to try each facility using the last program as an experimental aid. For the sake of completeness certain commands already used will be included in the following subsections. It is understood that all commands are terminated with the *RETURN* key and the symbol ↵ will no longer be used.

1.6.2 Program Entry

AUTO causes the system to provide line numbers automatically usually in the sequence 10,20,30 . . .
The *ESCAPE* key or the *RETURN* key will cause exit from this mode of program entry.
AUTO 150 will cause the numbering to start at line 150.
AUTO 150, 2 will cause the system to provide the sequence 150, 152, 154 . . .

NEW clears the program memory ready for the entry of a new program.

System-forced Indenting is also a most valuable aid. When indenting of program statements becomes appropriate the system will relieve the user of this clerical chore.

Some COMAL systems will further improve presentation by forcing keywords into upper case and other program words into lower case or vice-versa.

1.6.3 Testing and Using Programs

RUN causes execution of a stored program in line number order. RUN 150 causes execution from line 150.

CON continues execution from where a program was stopped without clearing the data area.
CON 150 will cause execution from line 150.

CHAIN followed by a program name causes the named program to be loaded from disc or other device and executed. This is normally a program statement.

1.6.4 Editing Programs

LIST causes a program to be listed with proper indenting and statements in line number order (as long as their correct line numbers are attached).

EDIT 150 causes line 150 to be displayed ready for editing. In some systems the 'cursor up' and 'cursor down' keys can be used instead. For variations in the use of edit command consult

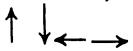
specific manuals. Note that in COMAL line numbers can be edited thus enabling lines or blocks of code to be physically shifted.

DEL 150 will delete line 150 from the program.
DEL 150-200 will cause deletion of lines 150 to 200 inclusive.
In some systems the DEL can be omitted.

RENUM or RENUMBER causes a renumbering of lines in a program. The result is the conventional sequence 10,20,30 . . .

RENUM 150 renumbers the program so that the first line is 150.

RENUM 150,2 renumbers the program producing a sequence 150, 152, 154 . . .

 These keys move the cursor in the direction indicated either one step at a time or continuously.

 These keys move text in a line to right or left.

RUBOUT or DEL causes backspacing of the cursor and deletion of one character.

1.6.5 Storing and Retrieving Programs

SAVE ⟨filename⟩ will cause a program to be saved on a magnetic disc or other device. The form of ⟨filename⟩ might be one of several possibilities, for example,

```
PROG
"PROG"
"1:PROG"
```

The manual for a system should be consulted.

LOAD ⟨filename⟩ will cause a program to be retrieved from a disc or other device. The form of ⟨filename⟩ will be the same as that used in the SAVE command.

CAT or LOOKUP may be followed by a disc-drive identifier or by a disc name. This causes a catalogue of the programs on the disc to be displayed.

ENTER ⟨filename⟩ causes a program to be added to the existing program in main memory. This enables programs or program segments to be merged without re-typing. It may be necessary to alter one program first using the RENUMBER or RENUM command to avoid a possible clash of line numbers.

1.6.6 Other System Commands

SELECT OUTPUT PRINTER or OUTPUT PRINTER or SELECT OUTPUT "LP" causes output or listing of program on printer rather than screen.

In some systems the default option of screen display resumes immediately after the program or listing ceases. In others it has to be reinstated by a command SELECT OUTPUT "DS" or something similar.

SIZE returns the size of the program currently in main memory and the size of free memory space.

BYE causes exit from COMAL to the system's next level of control.

ZONE (18) would set the field of a PRINT statement in which elements are separated by commas, at 18 positions.

The commands CHAIN and SELECT OUTPUT PRINTER or OUTPUT PRINTER can be used as program statements.

In subsequent sections or chapters line numbers will not be used. Nor will such words as RUN, NEW or the symbol for *RETURN*. In order to keep the presentation of material simple and uncluttered such details will be left to the reader. Indenting and upper case letters for keywords will also be shown but the user need not worry about these things when entering a program.

1.7 RANDOM NUMBERS AND RANDOM CHARACTERS

The statement

```
PRINT RND(1)
```

will cause the printing of a random number in the range 0 to 1. Strictly speaking the number is not random. Statistical tests could reveal patterns. For our purposes and many others this is a useful function and its lack of statistical validity need not be a cause for undue concern.

Sometimes random number generators are so arranged that, in a given program the same sequence of 'random' numbers will always be produced. This is sometimes useful in developing simulation programs. It is easier to test program behaviour if random numbers are not varying from run to run. However it is usually better to have different results on every run and this can be achieved by placing the word RANDOMIZE once near the beginning of a program.

Ranges of random whole numbers (integers) can be produced easily in COMAL, for example the throwing of a single die could be simulated with

```
PRINT RND(1,6)
```

The throwing of a pair of dice, as in the game "Monopoly" could be simulated with:

```
RANDOMIZE
die1   := RND(1, 6)
die2   := RND(1, 6)
score  := die1 + die2
PRINT score
```

or more economically:

```
RANDOMIZE
PRINT RND(1,6)+RND(1,6)
```

Random alphabetic characters can also be produced by combining the RND and CHR functions. The CHR function returns the character corresponding to the code used as the operand. For example

```
PRINT  CHR(65);  CHR(90)
```

would cause A Z to be printed. In PET COMAL add 128 to these code numbers.

The program

```
RANDOMIZE
PRINT CHR(RND(65,90))
```

would produce an upper case letter in the range A to Z.

PROBLEMS

Write down the output, or possible output, you would expect from the following computer programs assuming they have been properly entered with line numbers and RUN has been typed (Problems 1.1-1.5).

1.1
```
susan := 24
PRINT "susan"
PRINT susan
```

1.2
```
DIM susan$ OF 4
susan$ := "jane"
PRINT "susan"
PRINT susan$
PRINT susan$ + " " + "susan"
```

1.3
```
hours := 40
rate := 3
wage := hours * rate
PRINT hours; rate; wage
```

1.4
```
sum := 50
num := 5
share := sum/num
PRINT share
```

1.5
```
PRINT RND(1,9)
PRINT CHR(RND(65,90))
```

1.6 Why is a DIM statement necessary for a string variable but not necessary for a numeric variable?

1.7 What might be the consequence of not typing NEW before entering a program?

1.8 Give examples of keywords which are concerned with each of the following:

(a) editing of programs
(b) execution of programs
(c) COMAL program statements

1.9 What is the effect of using the RND function without RANDOMIZE at the start of the program?

1.10 If you were designing a computer language from scratch what alternatives might you prefer to the form:

```
variable := expression
```

for an assignment statement.

1.11 In what ways does the COMAL system help the user in matters related to:

(a) line numbers?
(b) the use of the symbol :=?
(c) indenting of program statements?

Chapter 2

Simple Data Representation and Movement

People are always transmitting or receiving data, even when they just hold hands.

W. Tagg at conference.

2.1 DATA AND INFORMATION

Strictly speaking, data is a plural form (of datum) but its usage as a collective noun is as common that it seems pedantic and pointless to do other than follow the trend. Chamber's Twentieth Century dictionary defines data as "Facts given from which others may be inferred". In computing we need a more precise meaning and it may be helpful to recall the old joke about the all-knowing computer conversing with the new managing director.

MD: Where is my father now?
Computer: Your father is fishing off Brighton pier.
MD: Wrong. My father died four years ago.
Computer: Your mother's husband died four years ago.
Your father is fishing off Brighton pier.

The word "father" is the same sequence of symbols or sounds to both computer and the managing director but even after the routine bit of processing to establish that "my father" means the managing director's father there are still two different interpretations. The data is the same but the information is different. Generally speaking people want information but computers receive, process and output data.

Internally the data which computers receive, process and output consists of little more than binary patterns – magnetic dots, electrical states or impulses, electronic switches, holes or their absence in paper tape, etc. At the points of contact with people – keyboards, visual display screens, printers – the data is converted from or to a wider set of symbols such as numeric digits, letters and other symbols but this data is not different in principle from the internal binary patterns. The letter A may be encoded internally as 1000001 but letters are only slightly richer in information than binary digits. When letters are grouped in

certain ways they can become words in some natural human language and may be well on the way towards the form which enables information to be extracted.

We need only occasionally consider in any detail the internal computer representations of data. We shall only occasionally concern ourselves with the way data is represented ultimately as binary patterns. Mostly we will consider the level of computer representation which reflects the concepts used in problem solving. At this level there are the simpler concepts such as numbers or strings of characters, and there are more complex ways of representing data. These more complex data structures will be introduced in later chapters.

2.2 SIMPLE DATA TYPES

2.2.1 Variables and Identifiers

A variable can take values that are within its data type. For example we can define a variable *width*, say, of type real number or real. We are really saying that it is possible for *width* to hold or take one of a large number of possible values which must be real numbers. Most BASICs offer two simple data types: reals and strings. Like other extensions of BASIC, COMAL offers two additional ones which will be explained in this chapter.

A variable is an abstract concept but we can imagine it quite sensibly as a pigeon hole capable of storing certain values. A **variable name** in COMAL is not abstract. It consists of up to sixteen characters the first of which must be a letter. The other characters may be letters or digits and some systems allow one other character. All the following are valid in at least one COMAL system. Identifiers cannot be the same as keywords of COMAL. There is no internal distinction between upper and lower case letters.

```
X
area
die2
month'total
year'end'total
```

Names like these are also called "identifiers": they are used to identify things other than variables, as we shall see.

2.2.2 Real Variables

A real variable can take a range of values and, in any particular system, it will be rounded to a certain degree of accuracy. This accuracy is measured by the number of significant digits. If a system offers seven-figure accuracy large or small numbers such as 123456789 or 0.123456789 could not be represented in the normal way. Seven-figure accuracy is preserved without extra storage being required and such numbers might be printed as

```
1.234568E+08  and  1.234568E-01
```

"E+08" means move the decimal point, eight places to the right. "E–01" means move the decimal point one place to the left.

2.2.3 Integer Variables

Real numbers are rather complex data types and their internal storage and manipulation reflects this complexity. Whole numbers or integers are simpler. They take up less internal memory and can be manipulated faster than real numbers. They are, of course, more fundamental and it seems odd that they should get the special type of identifier to distinguish them from real numbers, rather than the other way round. This reflects the history of BASIC and COMAL's relationship with it.

An **integer variable** is tagged with a % sign thus:

```
num%
count%
days%
month%
```

Some systems use a # sign instead. The advantages of using integer variables when fractions are definitely not needed are appreciable:

(1) There is no chance of small errors. However complex the computations an integer variable can only take exact whole number values.
(2) They take less internal storage.
(3) They can be processed more quickly.

In COMAL the disadvantage is that the rather ugly % sign must be attached. Where complex computations are not involved, and neither memory space nor speed is critical real variables will be used simply because they look better. But it should be remembered that such matters may be critical in computer work and integer variables can be advantageous.

2.2.4 String Variables

We have already seen that string variables must be **declared** in a DIM statement, we may assign values to them and we may join the values of two string variables. There is an unusually large amount of variation in the notation and facilities for string handling in different BASICs. However, there are certain operations with strings which can be done in most BASIC or COMAL systems. There are variations in the way such operations are treated and not all can be done in all systems.

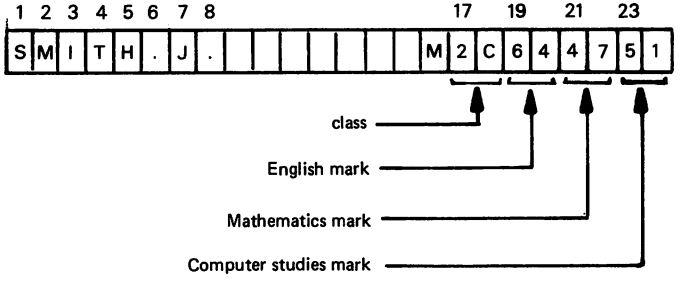
In order to exemplify these operations in a fairly realistic manner we will suppose that there exists a string variable *rec*$ which is used for passing information about school pupils to a file. The string variable and its associated record is defined below and in Fig. 2.1.

DIM rec$ OF 24

The record is to consist of six fields occupying the character positions of rec$ as shown.

Field	*Character Positions*	*Example*
Name	1 – 15	SMITH, J.
Sex	16	M
Class	17 – 18	2C
English mark.	19 – 20	64
Mathematics mark	21 – 22	47
Computer Studies mark	23 – 24	51

rec$



Note

The digits, dots and letters are all of string data type. The mathematics mark of 47 could be extracted as a substring but it could not be processed as a number without first being converted to numeric (real or integer) data type.

Fig. 2.1 – A record format.

JOIN TWO STRINGS

In constructing the last part of the string we might use an assignment statement after suitable preliminary work, including the declaration of *mark$* as a six-character string and the others as two-character strings.

```
mark$ := eng$ + math$ + comp$
```

PLACE A SUBSTRING IN A STRING

We may wish to place the above string into substring positions 19 – 24 of the string *rec$*.

```
rec$(19:6) := mark$
```

means assign the six characters of *mark$* into six positions in *rec$* starting at the nineteenth.

COPY A SUBSTRING FROM A STRING

It may be necessary to read the marks data from the record and we can do this by reversing the above assignment:

```
mark$ := rec$(19:6)
```

or to get the English mark:

```
eng$ := rec$(19:2)
```

CONVERTING STRING DATA TO NUMERIC DATA

The two characters "64" representing the English mark do not comprise a number in the full sense that we can apply numeric operations to it. For example we may wish to scale the English marks, increasing them all by one eighth. We cannot write

```
eng$ * 1.125
```

even though the value of eng$ is "64". The system simply cannot perform arithmetic operations on string data. Some systems enable the conversion to be made easily by writing:

```
1.125 * VAL(eng$)
```

In other systems a small procedure (see chapter 5) is necessary. A method is given in the solution to Problem 5.14.

COMPARING STRINGS

In sorting or searching processes we may wish to compare two strings or the values of two string variables. In order to do this we use the **relational operators**

```
 =  is equal to
 <  is less than
 >  is greater than
<=  is less than or equal to
>=  is greater than or equal to
<>  is not equal to
```

"A" is less than "B" because the internal numeric code of "A" is less than that of "B" (see Appendix III). All of the following **conditional** or **Boolean** (see next section) **expressions** are TRUE.

```
"ALF"    <  "BEN"
"ADOLF"  <  "BEN"
"ADA"    <  "ADOLF"
"ALF"    <= "BEN"
"ALF"    <= "ALF"
"ALF"    <> "ADA"
"PET"    <  "PETE"
```

CONVERTING NUMERIC DATA TO STRING DATA

The reverse of the above process is sometimes required. Suppose that the number 64 is the value of a numeric variable *english.* Some systems will allow the conversion easily by writing:

```
eng$ := STR(english)
```

Other systems would need a small procedure (see chapter 5) for this.

LENGTH OF A STRING

Although a string variable has a maximum length defined by its declaration, the actual length of the value of a variable can be anything between zero and the defined maximum. At any time the actual length can be found by writing:

```
LEN(string$)
```

where *string*$ can be any string variable.

SEARCHING A STRING FOR A SUBSTRING

We may wish to search a string for a particular substring. For example, we could search *rec*$ to see whether "2C" occurs anywhere. Some systems would allow one to write:

```
start := "2C" IN rec$
```

If 2C occurs in *rec*$ the position of its first character would be placed in *start.* Thus *start* may take the value 17. If 2C did not occur anywhere in the string *start* would take the value zero.

Another method of searching a string is given in Example 4.3.

String-handling is the beginning of one approach to data-processing. It is an important topic capable of considerable development leading naturally into the use of files. The objective of this section has been to introduce the fundamental concepts which will be used in further work throughout the book.

2.2.5 Boolean (Logical) variables

Boolean variables are named after George Boole who developed some principles and techniques of mathematical logic in the nineteenth century. Mathematical logic deals with things which are either TRUE or FALSE. We could, for example, simulate ground being hard or soft by writing:

```
soft := RND(0,1)
IF soft=1 THEN (ground is soft)
```

The expression

```
soft=1
```

is TRUE if soft has the value 1

and otherwise FALSE. The IF . . . THEN statement is discussed in detail in Chapter 4 but it is clear that the form

IF (condition) THEN (action)

requires that (condition) should be TRUE or FALSE. Any numeric variable placed in the position of (condition) is treated by the system as FALSE when zero, otherwise TRUE.

Thus we can write

```
soft := RND(0,1)
IF soft THEN (ground is soft)
```

The variable *soft* is treated as a Boolean or Logical variable. Although it can take any numerical value it is treated as though its only possible values are FALSE (zero) or TRUE (non-zero). The advantages of Boolean variables are not obvious. We could manage without them but it will be shown, in due course, that their use can make programs more readable. The little effort it takes to understand them is worth taking.

2.3 THE STORED PROGRAM

The idea of a stored program is now commonplace. A set of instructions for the computer is stored alongside the data, if any, on which they will operate. This concept was inherent in the early nineteenth century computer designs of Charles Babbage and it has not changed. However, it is worth re-stating certain important points about this central theme.

Clearly computers would not be effective if programs were not stored. They would be like calculators which achieve results slowly because of the need for constant human participation. The idea of a stored program in that the human should decide what is to be done in detail and list his instructions for the computer to start executing at a given signal. But a simple linear execution of a list of instructions would be over in a flash. The programmer must build into his program both repetition and decision-making if the power of computing machinery is to be exploited.

A programmer's position can be compared to that of a football manager. The manager must somehow inculcate a set of principles and rules in such a way that the team plays good football when they are on the field and he no longer has direct control. The principles and rules can be written as static text but they guide a very dynamic activity. A football manager might summarise his instructions as follows.

> Now then lads, I want you to go on the field for each half, play hard and leave the field tired after each half. Remember that we've agreed not to attempt dramatic but unlikely shots from outside the penalty area. While the ball is in play, if they have it, get into your defensive positions but, if we have it, keep passing safely until you can attack.

When we're attacking if you have the ball keep moving forward or pass, otherwise run into an open space where you might receive a pass. Keep cool in front of goal and only shoot from inside the penalty area when there is a real chance.

Programming languages are still much more formal than this but a rough translation using keywords of COMAL might be as shown in Fig. 2.2.

```
REMARK Football Manager's Instructions to his Players.
FOR half := one TO two DO
 REPEAT
  WHILE the ball is in play DO
   IF opponents have ball THEN
    play in defensive positions
   ELSE
    REPEAT
     pass safely to team-mate
    UNTIL good attacking opportunity occurs
    EXEC attack
   ENDIF
  ENDWHILE
 UNTIL half time OR final whistle
 leave the field and go to dressing room
NEXT half

PROC attack
 REPEAT
  IF you have the ball THEN
   keep moving forward or pass
  ELSE
   run into space to receive a pass
  ENDIF
 UNTIL ball is in penalty area AND shot is possible
 shoot
ENDPROC
```

Fig. 2.2 – A football manager's program.

The precise significance of each keyword, and of the indenting, will become apparent in due course but what is clear is that a highly dynamic process (the players' actions in a game) involving much reptition and decision making is directed or described by something which is essentially static (the text in Fig. 2.2) and storable.

A programmer's text, although static and linear, can be made to reflect what is a highly dynamic and non-linear process in the computer. Or he can work as though in a fog with a much less readable and meaningful text eventually getting what seems to be a correct result by much trial and error. This book is dedicated to methods which help the programmer to retain control of the dynamic processes by writing readable programs in professionally constrained ways.

The above program reacts to data which is generated in some way during the course of the game (who has the ball? where is it? is there an opportunity to attack? etc.) but a common situation in computing is analagous to other types of processing. For example a carpet making machine is programmed with a pattern and receives wool and hessian or nylon and foam rubber as input. The output is a carpet (see Fig. 2.3).

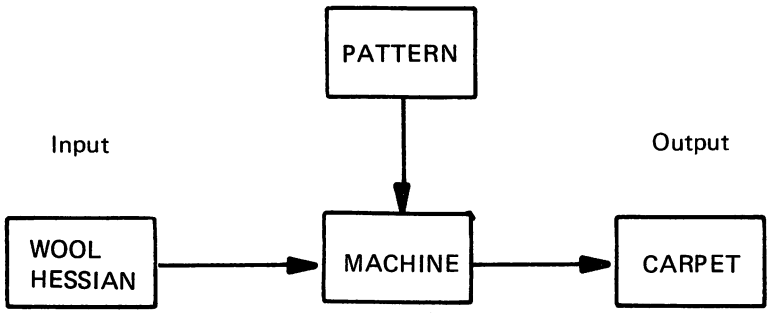


Fig. 2.3 – Carpet making.

A computer's input and output consists entirely of data as shown in Fig. 2.4.

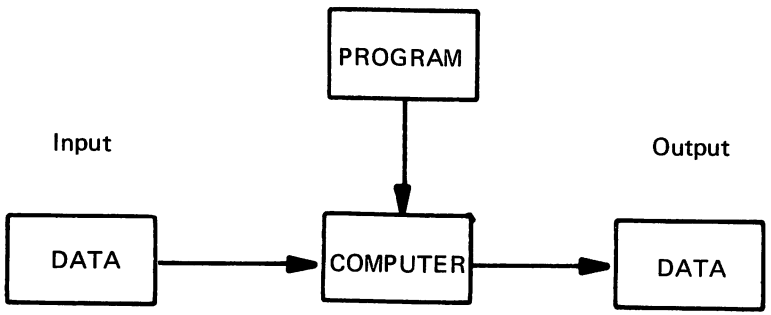


Fig. 2.4 – Data processing.

The data on which the program operates should be separated in some sense from the program itself. This enables one to change the data without necessarily rewriting any of the program. In BASIC and COMAL the data is sometimes physically included in the program but the conceptual separation should be preserved by placing the data in a particular place such as the end of the program.

It is sometimes very fruitful to regard a computer program as a process for converting what is available or what can be made available as input into what is required as output. Such a suggestion might seem obvious common sense but a good deal of current practice particularly in the fields of simulation and the design and processing of questionnaires could benefit from asking the simple questions;

What information do I need?
How can I get just this information and no more?
What do I need as input?

Finally, the form in which data is handled is of fundamental importance. The processing intentions can determine the most effective and efficient ways of structuring the data for input. Structured programming can only be fully effective if data is properly treated as well.

2.4 INPUT

2.4.1 READ and DATA statements

We can assign values to numeric variable with assignment statements but this is a cumbersome way to handle more than a few items of data. It also contravenes the principle of separating program and data. BASIC and COMAL provide one of the simplest and most effective methods of handling small and medium quantities of data. We can place the three examination scores (Fig. 2.1) in a DATA statement:

```
DATA 64,47,51
```

This statement is unlike most COMAL statements in that it is not executable. It is really a small record or file of data conceptually external to the program though in fact it can go anywhere in a COMAL program. As it stands nothing can be done with the data. It is necessary to READ it into suitable variables:

```
READ english, maths, compst
```

Note that the variables must match the data in order and number.

We could read a complete record into various *fields* with two statements:

```
DATA "SMITH.J.","M","2C",64,47,51
READ name$,sex$,class$,english,maths,compst
```

The variables in the READ statement must match the items in the DATA

statement in order, number and *data type*. Note that the three marks are numbers, not strings, in this example.

2.4.2 INPUT Statements

In some circumstances it is necessary for a programmer to allow for information to be given to a program at **run-time**. That is, to make the program execution pause while the user enters something from the keyboard. Often a game or computer assisted learning program will give the user the option of having rules or instructions displayed. The INPUT statement takes information from the keyboard in the same way that a READ statement takes information from a DATA statement.

```
PRINT "Do you want instructions?"
INPUT answer$
```

After printing the message, "Do you want instructions?", the computer waits for a reply terminated with the *RETURN* key. The program then continues according to the nature of the reply which has become the value of *answer*$.

Because it is almost always necessary for an INPUT statement to be associated with a message provision is made for this without the use of a PRINT statement:

```
INPUT "Do you want instructions?" : answer$
```

The INPUT statement is useful but it should not be used where a DATA statement will do the job. The DATA statement is usually quicker and safer in the long run.

2.5 OUTPUT

2.5.1 Text Output

The PRINT keyword may be followed by a number of elements separated by commas or semicolons. The use of commas splits the print line into fields or zones which may be set by a ZONE statement.

```
ZONE := 8
PRINT "A","B","C"
```

will give the printout

```
A       B       C
```

A is in position 1, B in position 9 and C in position 17. Thus the zones are eight characters wide. The default option for ZONE is zero. Some systems do not have the ZONE statement and it is effectively fixed at about eight.

The PRINT elements can be constant strings or numbers or variables or expressions. If semicolons are used instead of commas they usually cause one or

two spaces between numeric elements and none or one between string elements. A semicolon at the end of a line suppresses the carriage return/linefeed and causes any further printing to continue on the same line.

Text output is normally directed to the screen but it can be directed to a printer using the OUTPUT or SELECT OUTPUT commands.

2.5.2 Graphical Output

There is probably more variety in graphics notation and facilities between different systems than in any other area of programming. In its simplest form graphical output means **cursor control** – the ability to print characters, special or standard, in any screen character position. Semi-graphics or low-resolution graphics, often to the teletext standard, mean that each character position is capable of being split into six small 'squares'. In this way a 24-line X 80-column screen becomes a 72 X 160 coordinate system. A 24-line X 40-column screen would become a 72 X 80 system. For only modest extra expense it is usually possible to move to medium or high-resolution graphics which might give resolutions of 200 X 300 or better. The terms are vague and one cannot give precise definitions. There are some cheap colour graphics systems but good quality colour graphics and very high resolution ordinary graphics tend to be expensive.

In this and other sections we shall make use of the simplest form of graphics – cursor control. We shall also stick to the across-down coordinate system because it is common and because it relates sensibly to the handling of lines of text for screen or printer. Variations in systems should be easy to accommodate to the principles and methods used here.

Typically a screen will consist of about 24 lines and 80 columns. Less than 80 columns is unsatisfactory because most printers work to this standard. Books often have 72 characters in a line and generally one needs about 80 columns for satisfactory layout of text output. The scheme is given in Fig. 2.5.

In order to place an asterisk at position (37,11) it is necessary to move the cursor to that position and print the desired character. A variety of notations exist for cursor control and some systems combine cursor control and printing into one instruction something like:

```
PLOT(37,11,"*")
```

We shall adopt a procedure notation for cursor control because almost any system could be made to work in this notation. The PRINT instructions are standard. Our notation for the above effect is:

```
EXEC cursor(37,11)
PRINT "*"
```

The precise notation for a particular system will soon be discovered from a manual or sample program. It is the idea, in this case, which is much more important.

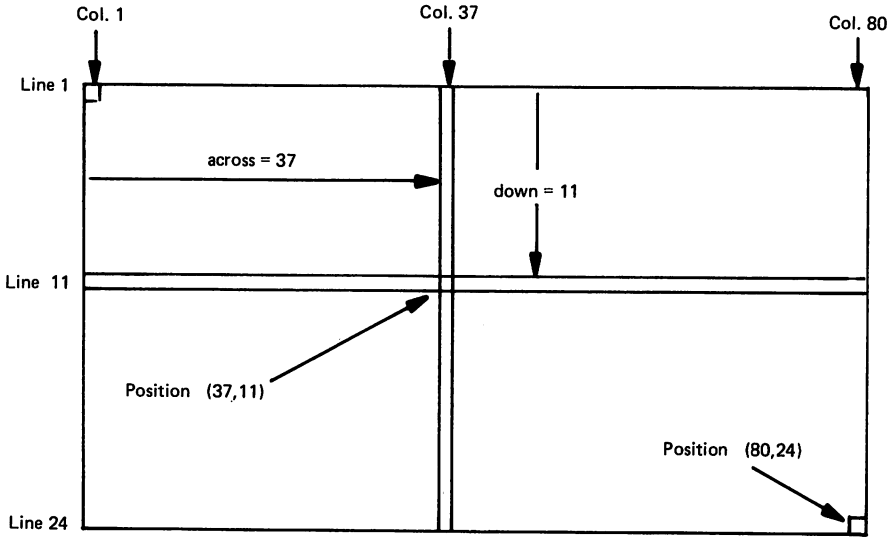


Fig. 2.5 – Screen character positions.

PROBLEMS

2.1 A ship is sailing in a fog near a coast. A man is placed at the prow of the ship to keep a look out for dangerous rocks. Eventually he raises his right arm horizontally as a signal to the helmsman. Why is this data (right arm raised) useless by itself? What must the helmsman know in order to interpret the data as useful information?

2.2 Give two examples of valid variable names of type:

(a) Real
(b) Integer
(c) Boolean
(d) String

and associate with each variable name a possible matching value.

2.3 In cases where whole numbers only are needed what are the advantages of using an integer variable?

2.4 What are the two possible meanings of + in COMAL?

2.5 Given that the statement

```
text$ := "A herb for a lamb chop."
```

has been executed, write down statements which will assign the sub-strings "herb" and "lamb chop" to the variables plant$ and meat$. Write down the DIM statement which would enable the use of the variables text$, plant$ and meat$.

2.6 If string variable, history$ has the value "65" and the variable geography$ has the value "50" what would be the effect of the following statements?

(a) total := history$ + geography$
(b) total$:= history$ + geography$

2.7 Given that the statement:

```
num$ := "123456789"
```

has been executed what would be the output from the following program segment?

```
ZONE := 8
one$ := num$(1:2)
two$ := num$(5:2)
three := VAL(one$)+VAL(two$)
four := LEN(num$)
five := "56" IN num$
PRINT one$, two$, three, four, five
```

2.8 Give a value TRUE or FALSE to the following statements, or give the value NEITHER if the statement is neither true nor false.

(a) Blood is red
(b) Grass is green
(c) Circles are square
(d) This statement is false

2.9 Give a value TRUE or FALSE to the following Boolean expressions. Write NEITHER in the case where it is not a valid Boolean expression.

(a) 2 > 1
(b) 2 < 1
(c) 2 = 1
(d) 2 >= 1
(e) 2 <= 1
(f) 2 <> 1
(g) 9 > 10
(h) LEN("nine") > LEN("ten")
(i) nine$:= "ten"
(j) "ZOE" > "PAM"

2.10 Why is it difficult for the appearance of a computer program to reflect what happens when it is executed by a computer?

2.11 What would be the output from the following program segment

```
DIM word$ OF 6
READ word$, num
PRINT word$; num
DATA "Twenty"; 20
```

2.12 What effect does an INPUT statement have on a running program?

CHAPTER 3

Repetition

Full thirty times hath Phoebus' cart gone round
Neptune's salt wash and Tellus' orbed ground,
And thirty dozen moons with borrow'd sheen
About the world have twelve times thirties been . . .

– Player King in Hamlet.

3.1 REPEATED OPERATIONS–1 FOR LOOPS

The power of computers could not be exploited effectively if every computer action was separately programmed. A programmer needs to arrange for at least some of the stored instructions to be executed repetitively. It is this magnification of effort which produces an acceptable relationship between human work and computer work. In Chapter 4 we will see how to make the computer choose between different possible courses of action and, whilst such techniques are essential, by themselves they produce less work than the programmer has directed. Chapter 5 is concerned with the problem of handling complexity by breaking down into smaller jobs and although the methods which will be described can add enormously to a programmer's efficiency this is not achieved by any great multiplication of his effort though there will be some.

Repetition, like the stored program concept, is another big step towards making computers the powerful tools that they are. As we shall see, associated with this idea are certain ways of storing data and it is really the combination of good algorithms with appropriate data structures that produces the desired effects. Nicklaus Wirth made the point very succinctly with the rather unusual and now famous title of his book, *Algorithms + Data Structures = Programs.*

Suppose, for example, that a foreman wishes to instruct a workman to dig four post holdes. There might be other associated jobs such as mix concrete, place concrete support, bolt post in position, cover unused cement and so on. For each sequence of repeated operations it is necessary to specify clearly what the sequence is and the number of times it is to be repeated. Since a computer program is itself a sequence of instruction, if the foreman behaved like one he would have to specify:

(1) The start of the sequence
(2) The end of the sequence
(3) The number of repetitions

A natural way to do this might be:

```
REPEAT 4 TIMES
   Dig a post hole.
END OF SEQUENCE
```

Partly for historical reasons, and partly because experience has shown its value, the actual syntax, using a PRINT statement to simulate the instruction, would be:

```
FOR post := 1 TO 4 DO
   PRINT "Dig a post hole."
NEXT post
```

The variable, *post*, is called the control variable or loop counter because its function is the control of the repetition rather than the process itself. The output from this program would be:

```
Dig a post hole.
Dig a post hole.
Dig a post hole.
Dig a post hole.
```

Although the main function of the variable, *post*, is to control the loop it can be used in the repetitive sequence. One would write:

```
FOR post := 1 TO 4 DO
   PRINT "Dig post hole number"; post
NEXT post
```

and the output would be:

```
Dig post hole number 1
Dig post hole number 2
Dig post hole number 3
Dig post hole number 4
```

The FOR statement:
(1) Causes the setting up of a counter, *post*, and defines its initial and final values: 1 and 4.
(2) Implicitly defines the start of the repeated sequence as the next program line.

The NEXT statement:
(1) Causes the counter to be incremented and if necessary a further repetition.

(2) Implicitly defines the end of the sequence.

Notice the indenting of the PRINT statement. This is an instance of an important feature of COMAL. It has two advantages:

(1) Readability of the program is enhanced.

(2) In more complex work if successive indenting is not fully reversed by the end of the program an error is revealed.

In the rest of this booklet indenting will be shown but the user need not provide it. After typing in a program the user need only type LIST and the system should display a properly indented version.

PROBLEMS

3.1 The number of repetitions may be as few or as many as necessary. Use the EDIT facility to alter the 4 in line 10: the FOR statement. Note especially the effects of making it 1 or 0.

3.2 In a football match simulation there are two halves and the overall structure of such a program might be:

```
FOR half := 1 TO 2 DO
  (Complex sequence of instructions)
NEXT half
```

Write and test a three line program to simulate this structure using one line such as PRINT "Half a game" to represent the complex sequence of instructions.

3.3 Test the RND function with the program which simulates dice throwing.

```
FOR die := 1 to 100 DO
  PRINT "Number"; die; "is"; RND(1,6);
NEXT die
```

Note that the 'random' numbers generated are in the range one to six. How would you make the results tabulate neatly one under the other?

3.2 REPEATED OPERATIONS —2 NESTED FOR LOOPS

Using the graphics facilities described in the previous chapter we can now see how to draw a simple rectangle on a screen.

Example 3.1

Draw a rectangle, on the screeen, consisting of five rows of seven stars with the top left hand star at position 21,6 (Fig. 3.1).

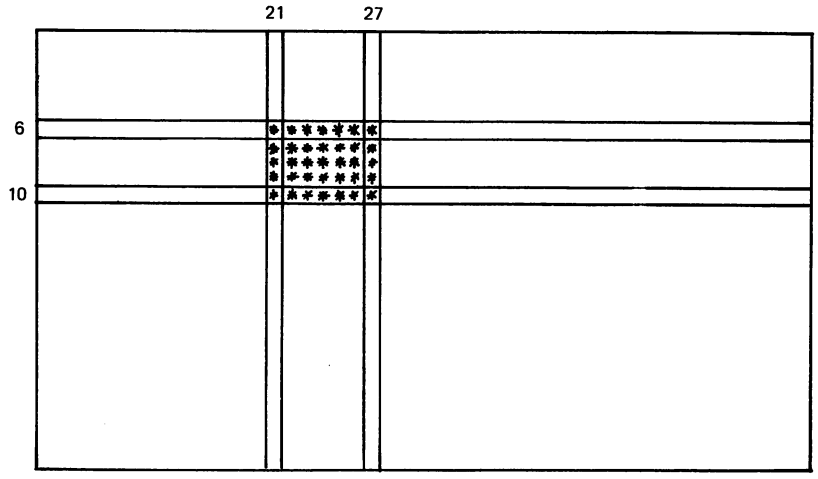


Fig. 3.1 – Stars on a screen.

To draw one line of stars we could write:

```
FOR across := 21 TO 27 DO
   EXEC cursor(across,6)
   PRINT "*"
NEXT across
```

Notice that the values taken by the variable *across* are in succession 21, 22, 23, 24, 25, 26 and 27 while the down coordinate remains constant at 6. The programme can be re-written without changing its effect:

```
down := 6
FOR across := 21 TO 27 DO
   EXEC cursor(across, down)
   PRINT "*"
NEXT across
```

It is now easy to see how we can make the rectangle by drawing five lines allowing *down* to vary from 6 to 10.

```
FOR down := 6 TO 10 DO
   FOR across := 21 TO 27 DO
      EXEC cursor(across, down)
      PRINT "*"
   NEXT across
NEXT down
```

We say that the inner loop is **nested** in the outer loop.

PROBLEMS

3.4 Alter the above program to draw a rectangle whose top left hand corner is position 30, 10 and size 20 units across, 5 down.

3.5 Alter the program resulting from exercise 1 so that instead of drawing 5 rows 20 units across it draws 20 columns 5 units high. Time the running of each program. If there is a significant difference in the times it may be useful to seek reasons.

3.3 REPEATED OPERATIONS —3 REPEAT . . . UNTIL

In a FOR loop, exit from the repeated sequence occurs when a counter reaches a pre-determined value. Sometimes a programmer must provide for exit from a loop when a condition becomes true instead of relying on a counter value.

Example 3.2

Simulate the rolling of a die until a six occurs.

We cannot use a FOR loop because it is not certain how many times the die will roll before a six appears. The structure we need has the form:

```
REPEAT
   (sequence of instructions)
UNTIL (condition)
```

The required solution is:

```
REPEAT
   die := RND(1,6)
   PRINT die;
UNTIL die = 6
```

It is instructive to compare the two types of loop used so far.

	OPENING KEYWORD	*CLOSING KEYWORD*	*EXIT ON*
FOR loop	FOR	NEXT	Counter
REPEAT loop	REPEAT	UNTIL	Condition

There is no difference in principle between these two types of loop. Both have a single entry point and a single exit point and both cause repetition. Yet the FOR loop was widely available in computer languages before the REPEAT loop. One can only speculate about the reasons for this. It may be partly to do with the need to economise on the then expensive memory space taken up by system software but the writer's view is that it has more to do with the gradual realisation over the years of the 1960s and 1970s that it is sensible to give the programmer exactly the structures that he needs so that he can write better (more easily readable) programs more easily and develop sensible skills of problem analysis.

3.4 PROGRAM STRUCTURE

The Control Demon

It may be helpful to regard a computer program as a set of instructions stored in the computer in a dormant state. If we imagine that an instruction can only be activated by a 'control demon' which causes the instruction to be executed and then to revert to its dormant state, we can make some rules for the behaviour of the control demon, sometimes referred to simply as 'control'.

RULES FOR THE CONTROL DEMON

(1) Control normally proceeds from top to bottom obeying each instruction in sequence.
(2) This normal sequence can only be broken by certain keywords such as FOR, NEXT, REPEAT, UNTIL.
(3) Two or more such words form what is called a **control structure.** Two is the minimum number because every structure must be opened and closed. These two words together with what lies between them may be called a **structural element.**
(4) The control demon can only enter a structural element at the opening keyword and he may only leave at the closing keyword.

An example will clarify the behaviour of the control demon in nested FOR loops.

Example 3.3

Write down the output you would expect from the following program and describe the behaviour of the control demon as the program executes.

```
FOR line := 1 TO 2 DO
   PRINT "Line starts"
   FOR star := 1 TO 3 DO
      PRINT "*";
   NEXT star
   PRINT "Line ends"
NEXT line
```

OUTPUT

```
Line starts
*** Line ends
Line starts
*** Line ends
```

CONTROL

The behaviour of the control demon is illustrated by Fig. 3.2.

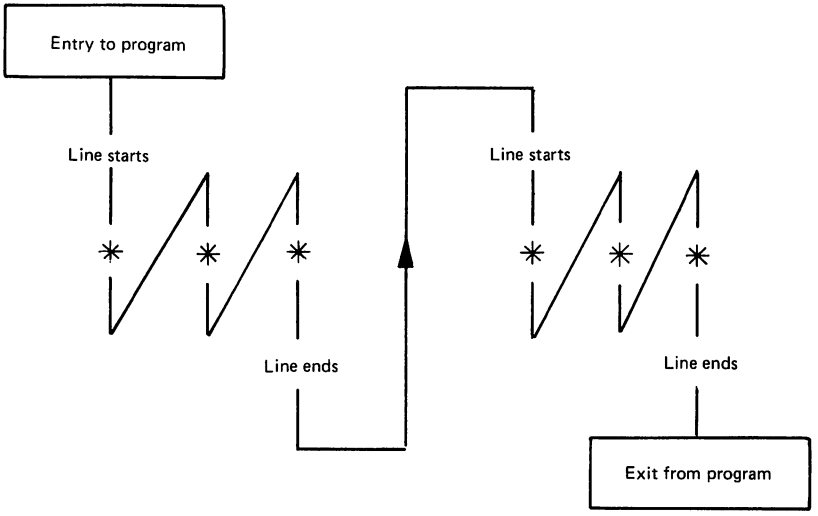


Fig. 3.2 – Control Demon's path in nested FOR loops.

Some definitions

A computer program consists of **program elements.** A program element may be a single statement, a sequence of statements or a set of statements starting with the opening keyword of a structure and ending with the corresponding closing keyword of the structure. The last is called a **structural element.**

A program statement is said to be **subordinate** to a structure if it lies within the structure.

A structural element which is subordinate to another structure is said to be **nested** within it.

With few exceptions, largely beyond the scope of this book, the main essential relations between program elements are sequential, subordinate and procedure calls which will be discussed later.

Example 3.4

Simulate throws of a single die until a six is scored. Repeat this game ten times and find the average number of throws per game.

METHOD

A counter will be placed in the inner REPEAT loop to record the total number of throws. On exit from the outer FOR loop the value of the counter will be divided by ten to give the required result.

```
PROGRAM
        count := 0
        FOR game := 1 TO 10 DO
          REPEAT
            die := RND(1,6)
            PRINT die;
            count := count+1
          UNTIL die = 6
          PRINT "End of game"
        NEXT game
        PRINT "Average is"; count/10
```

The structure of the program can be illustrated as in Fig. 3.3.

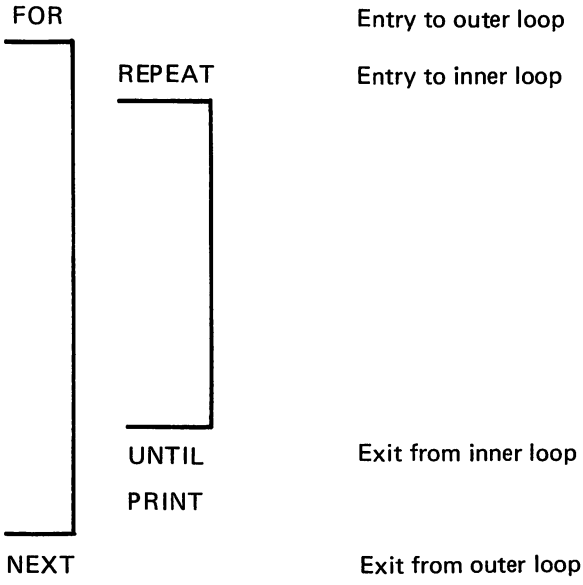


Fig. 3.3 – A REPEAT loop nested in a FOR loop.

The inner loop is nested completely within the outer loop which means that:

(1) Control must enter the outer loop before it can get into the inner loop.
(2) Control must exit from the inner loop before it can exit from the outer loop.

It would make no sense if "NEXT game" were placed before "UNTIL die = 6". The importance of these ideas is hard to underrate and they apply to all structures. It means that the solution of a problem consists substantially of answers to the following questions.

(1) What are the necessary structural elements?
(2) Which follows which?
(3) Which is nested within which?

Some legitimate program structures are shown in Fig. 3.4.

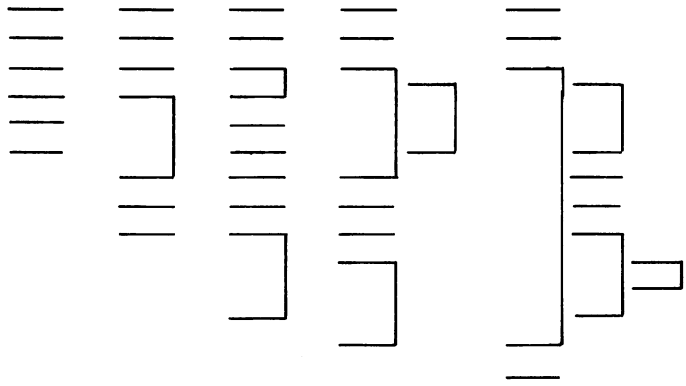


Fig. 3.4 – Legitimate program structures.

It is possible to create other types of structures in COMAL in unconstrained variety but, unless there are special reasons, it is unwise because such programs can easily become difficult to read, less predictable in their behaviour and educationally obstructive.

Although it is necessary to understand how the computer reacts to a particular structure – how control flows into, round, and out of it – a good programmer should try to see loops and other structures as whole elements and concentrate on these and the relations between them rather than worrying about the detailed flow of control. The reason for this is simply that problems should be analysed in terms of the elements and relationships. A good problem analyst will almost certainly be a good programmer because it is relatively easy to get details of syntax and presentation right once a correct structure has been established.

How different individuals perceive analyses of problems may be slightly mysterious, but a method known as top down analysis goes particularly well with structured programming and it is worth examining.

3.5 TOP DOWN ANALYSIS AND STEPWISE REFINEMENT

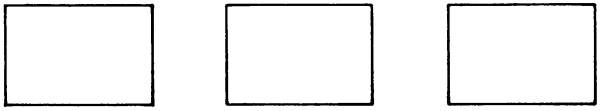
The process by which the program of Example 3.4 was constructed is not clear. An experienced programmer can just write down the program for a trivial problem but there is a systematic method by which more complex problems might usefully be tackled or which less experienced programmers might use to gain insights and develop skills in problem analysis.

We shall repeat the dice problem and show how a **top-down analysis** might proceed in this very simple case.

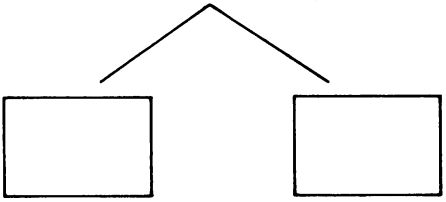
Simulate throws of a single die until a six is scored. Repeat this game ten times and find the average number of throws per game.

In analysing this problem we will develop a **structure diagram.** These are fully discussed in Chapter 6 but for the present it is sufficient to understand three ideas.

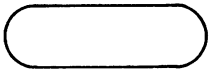
1. Statements in sequence



2. Statements subordinate to a structure (or program title).



3. Symbol for repetition.



Stage 1

We decide that something (a dice game) is to be repeated ten times (Fig. 3.5).

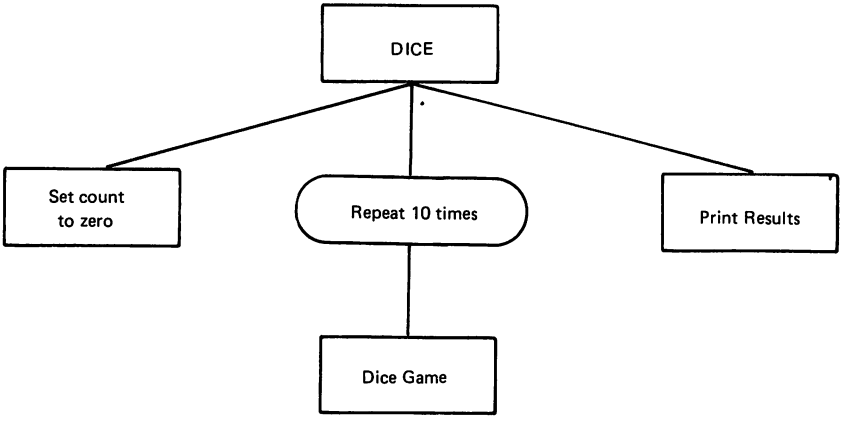


Fig. 3.5 – Stage 1.

Stage 2

These two outer boxes cannot be analysed much further. They are ready for translation into code, but the "Dice Game" box can be further refined as in Fig. 3.6.

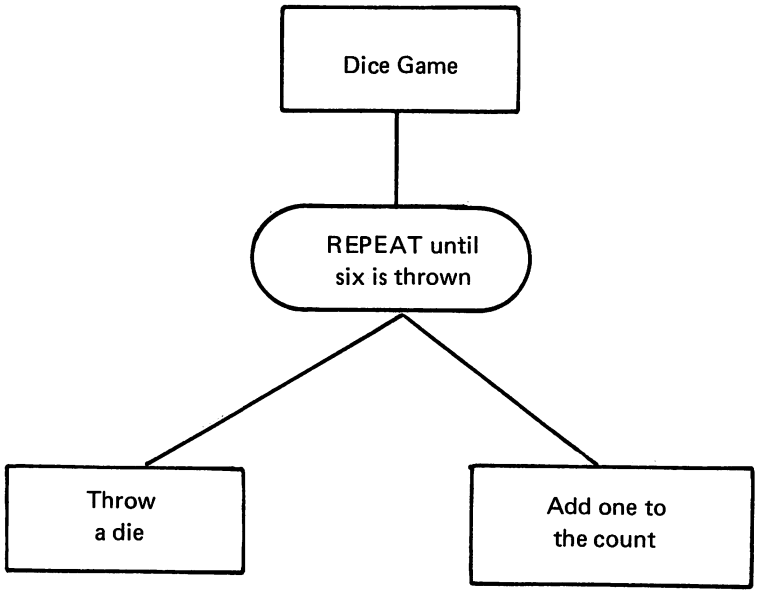


Fig. 3.6 – Stage 2.

Stage 3

The only box which can be further refined is "Throw a die". This is done in Fig. 3.7.

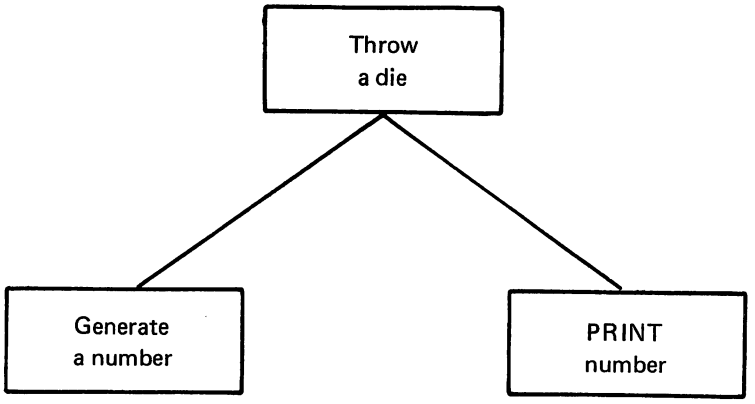


Fig. 3.7 – Stage 3.

Stage 4

No further refinement is necessary and we can put the whole structure together to form the structure diagram in Fig. 3.8.

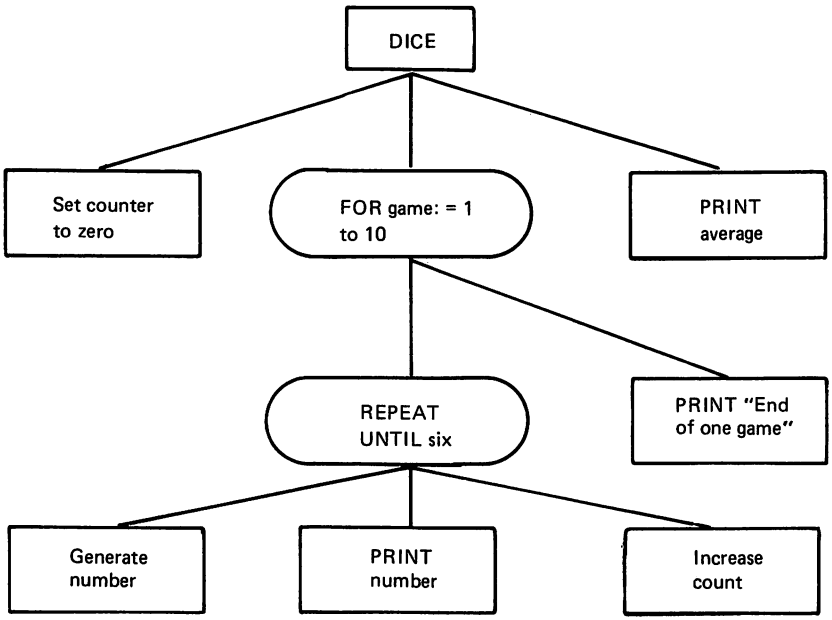


Fig. 3.8 – Final Structure Diagram.

Stage 5

The structure diagram could be re-written using the actual coding of COMAL but it is quite easy to proceed directly to the program. This can be done by means of a **natural walk** around the structure diagram. This kind of diagram is called a **tree**. The uppermost box is called the **root** (the tree is upside down). Boxes are called **nodes**. Lines between them are called **branches**. Boxes which have no lines attached beneath them are called **terminal nodes** or **leaves**.

The rules for a natural walk to get the program are as follows:

(1) Start at the root and "walk" so that your left hand always touches a branch or a box.

(2) When you come to a rectangular box write down the corresponding program statement.

(3) When you come to a non-rectangular box on the way down the tree write down the statement which opens the structure.

(4) When you come to a non-rectangular box on the way up the tree write down the statement which closes the structure.

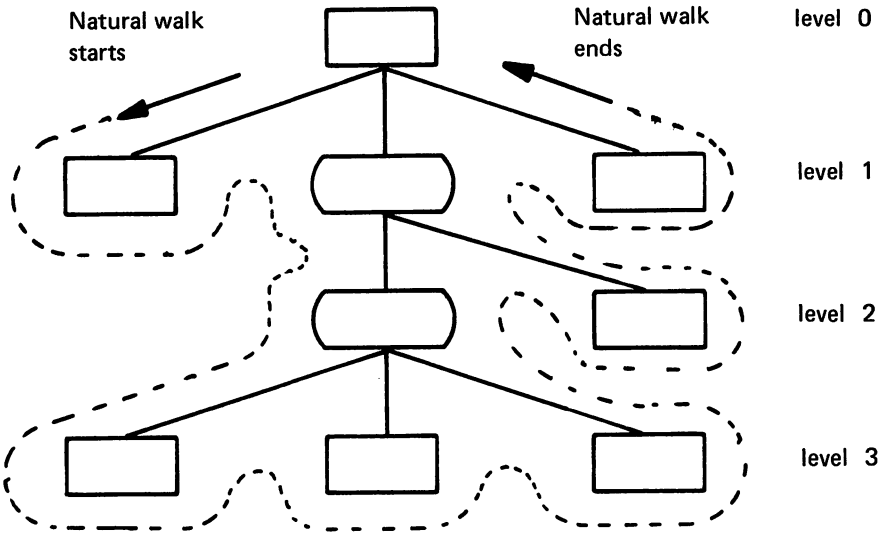


Fig. 3.9 – Walking round a structure diagram.

It can be seen from Fig. 3.9 that the relationship between the structure diagram and the program is very strong. One can easily be obtained from the other. Levels in the diagram correspond to the levels of indentation in the program of Example 3.4.

Some readers may feel that the method described above is overelaborate for a simple problem. This is true, but most people will find such ideas useful sometimes. In the writer's experience their value has been particularly evident in early stages of learning about problem analysis and program design and also when problems become complex or reveal unexpected snags. There will be more discussion about this crucial area in later chapters.

Example 3.5

Draw a triangle of stars on the screen as shown so that the top of the triangle is at postion 40 across, 5 down.

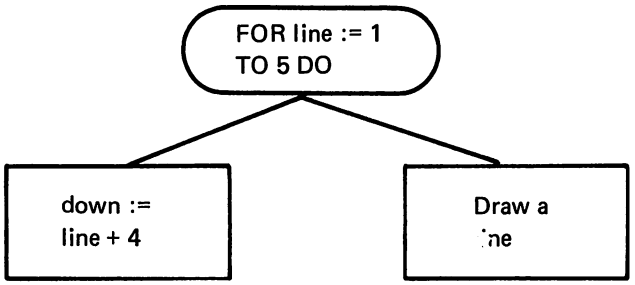
```
*
**
***
****
*****
```

METHOD

(1) We need to draw five lines which we will number 1 to 5 and we will consider how the (across down) co-ordinates change in each line.
(2) Within each line we must fix the first and last across values. The down value will be constant.

PROBLEM ANALYSIS/PROGRAM DESIGN

We can see that the down values which are constant in each line start at 5 for line 1 and increase to 9 for line 5.



In drawing a line we know that the across value always starts with 40. The end of line value is 40 on line 1 rising to 44 on line 5. The across values therefore range from 40 to *line* + 39 on each line. See Fig. 3.10.
The double rectangle indicates a procedure call which will be discussed later.

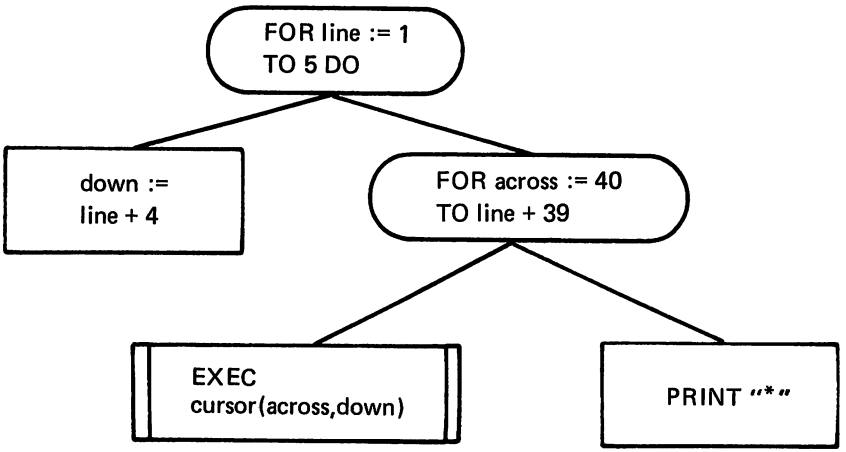


Fig. 3.10 – Drawing a rectangle of stars.

```
PROGRAM
        FOR line := 1 TO 5 DO
          down := line + 4
          FOR across := 40 TO line + 39 DO
            EXEC cursor(across, down)
            PRINT "*"
          NEXT across
        NEXT line
```

Example 3.6

Square roots can be calculated from the fact that if we make any guess at the square root of, say, 10 if the guess is larger than it should be then 10 ÷ *guess* must be smaller, and vice versa. For example if we guess 4 then 10 ÷ 4 = 2.5 which is smaller than the actual root. If we guess 2 then 10 ÷ 2 = 5 which is larger than the actual root.

It can also be shown that the average value of *guess* and 10 ÷ *guess* will be closer to the correct result than *guess* by itself. All we have to do is repeat this process until we get a sufficiently accurate answer.

METHOD

(1) The repetitions will be terminated when the difference between two successive results *old* and *newval* is less than, say, 0.0001.

(2) If we perform the subtraction, *newval* – *old*, the answer may be positive or negative. The function ABS will always return a positive value so the test is

```
ABS(newval – old) < 0.0001
```

(3) The sequence of calculations to get successively better approximations is

```
newval := ½(old + 10/old)
```

(4) An important point to recognise is that at some stage the *newval* value must become the *old* value so that we can use it in step (3). However, the assignment

```
old := newval
```

will cause both to be the same. The difference *newval* – *old* would be zero and the computation would finish before it should. The trick is to assign the *newval* value to *old* before doing step (3).

PROBLEM ANALYSIS/PROGRAM DESIGN

The heart of the program can be written immediately.

```
REPEAT
  old := newval
  newval := 0.5*(old + 10/old)
UNTIL ABS(newval – old) < 0.0001
```

It is only necessary to give *newval* an initial value and print the result.

```
PROGRAM
        newval := 4
        REPEAT
          old := newval
          newval := 0.5*(old + 10/old)
        UNTIL ABS(newval – old) < 0.0001
        PRINT newval
```

COMMENT

If we generalise the program to compute the square root of any number we can continue to provide the first guess or we can let it be half of the number. The method is so good and so fast that even a bad guess like this will do for a start. For example, the square root of 400 is 20 which makes a first guess of 200 very poor but the next approximation is

$$0.5 \times (20 + 400/200) = 11$$

which is very much closer to the correct result. The next result is even better:

$$0.5 \times (11 + 400/11) = 23.7 \text{ approximately.}$$

DEVELOPMENT

The following program reads five numbers from a DATA statement and prints each number and its square root.

```
FOR k := 1 to 5
  READ num
  newval := num/2
  REPEAT
    old := newval
    newval := 0.5*(old + num/old)
  UNTIL ABS(newval – old) < 0.0001
  PRINT num; newval
NEXT k
DATA 10, 400, 1326, 9999, 64
```

PROBLEMS

3.6 Print the upper case letters of the alphabet in sequence from A to Z. (3)

3.7 What is the result of doubling a number twelve times, starting with 1? (3)

3.8 The number of bacteria in a culture quadruples every hour. If there are ten bacteria now what will the number be in twelve hours? (3)

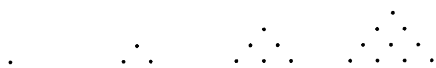
3.9 Simulate throws of a pair of dice, as in 'Monopoly'. Print the total of the two dice each time until a double appears, then stop. (3)

3.10 Generate upper case letters in a random manner until a "Z" appears. (3)

3.11 Include a counter in program 5 so that the number of letters generated in a run can be printed. (4)

3.12 How many times must a number be doubled, starting with 1, in order to exceed one million? (4)

3.13 The 'triangular' numbers are:



1 3 6 10 . . . etc.

Print all the triangular numbers until one of them exceeds 1000. (3)

3.14 Mathematicians can prove that the series:

$$1 + \tfrac{1}{2} + \tfrac{1}{3} + \tfrac{1}{4} + \tfrac{1}{5} \ldots \text{etc.}$$

is divergent which means that the sum of a sufficient number of terms will exceed any stated amount. How many terms would be required for their sum to exceed 6? (3)

3.15 Generate trios of letters selected from A,B,C randomly until the word "CAB" is accidently spelled. (4)

3.16 Include a counter in program 3.15 to compute and print the number of words generated until CAB is spelled. (4)

3.17 Two firms, Oldbats Ltd., and Upstart Ltd., produce cricket bats. In 1980 Oldbats produced 800 but its output is decreasing by 40 each year. Upstart produced 300 bats in 1980 and increases production by 50 each year. In what year will Upstart produce more bats than Oldbats? (4)

3.18 In the absence of predators an insect population would grow in such a way that each generation would be the sum of the sizes of the two previous generations. Predators cause a loss of 2000 in each generation. We can define three successive generations as *old, previous, new* and use the assignment

new := previous + old − 2000

If generations 1 and 2 are 4000 and 5000 insects respectively, compute all the generations up to the tenth. (4)

3.19 Write a program which will accept as input the value of any string, *word$* of up to five digits and convert it to the corresponding numeric value. (4)

Note

In a system with the function VAL the solution is trivial being essentially

```
num := VAL(word$)
```

However some systems have only ORD which simply returns the value of the numeric ASCII code of the first letter of a string. Thus ORD(word$) would return the code of the first letter of word$.

3.20 Print out all the times of buses scheduled to arrive every fifteen minutes from 7.00 a.m. up to 11.45 a.m. (7)

3.21 Draw on the screen a triangle of stars as shown with the top start at position 40 across, 5 down. (7)

3.22 Simulate die throws until a six appears then stop. Repeat this game twenty times and compute the average number of throws it takes to get a six. (8)

3.23 Compute the value of £1000 invested at
(a) 11% paid annually
and (b) 10% paid quarterly
giving the total accumulated at the end of seven years. (8)

3.24 Extend programs 5 and 6 to compute the average number of letters generated to get a "Z" in a total of 25 runs. (9)

3.25 In thirty runs of program 3.16 what is the average number of "words" generated per run? (9)

CHAPTER 4

Decisions

> There's a rule saying I have to ground anyone who's crazy. . . .
> Sure there's a catch. Catch-22. Anyone who wants to get out of combat duty isn't really crazy.
>
> Doc Daneeka in *Catch 22*

4.1 BINARY DECISIONS

In order to provide a simple context for discussion a special problem has been invented. It continues and elaborates the idea of a foreman giving instructions to a workman, and it will be used in this and further chapters. More realistic problems will also be introduced but this one has the advantage, despite its artificiality, that its solution can illustrate all the main structures likely to be used in introductory work.

Example 4.1
The following instructions are given to a workman by a foreman:

> We are going to put a fence around this field and your job is to dig all the post holes and put in concrete supports. Dig a half metre hole at each of the four corners. Between the corners dig post holes a third of a metre deep where the ground is soft and a quarter of a metre deep where it is hard. Put concrete supports in the holes between the corners according to the level of the ground. The supports can be one fifth, two fifths or three fifths of a metre. There are no trees at the corners but if there is a tree near a post hole position get Fred with the tractor to pull it out.
>
> Simulate the workman's actions using random numbers to make choices about the state and level of the ground and the presence of trees. Make assumptions about the length of each side of the field.

In this chapter we are concerned with decisions (also called selections or choices in computer literature) and there are examples of various types of decisions in the problem.

The most general form of a binary decision is:

```
IF (the ground is soft) THEN
  PRINT "Dig a half-metre hole."
ELSE
  PRINT "Dig a third-metre hole."
ENDIF
```

In this structure we have an opening keyword (IF) and a closing keyword (ENDIF) but ELSE is also important: it closes the first part of the structure and opens the second part. We may refer to it as a structure separating keyword.

A binary decision need not have an ELSE clause. "To be or not to be" is still a binary decision. For example we may write:

```
IF (There is a tree) THEN
  PRINT "Get Fred with the tractor."
  PRINT "Dig out roots."
ENDIF
```

Finally, in cases where we need only a simple 'one-liner' because there is no need for ELSE and only one statement after THEN we can write:

```
IF (There is a tree) THEN PRINT "Get Fred"
```

In this special case the closing keyword is not necessary.

The above program designs can be converted into correct syntax by using a random number to determine exactly what happens. The reader will have noticed the use of conditions or conditional expressions in REPEAT loops. A conditional expression can take one or only two values: TRUE or FALSE. For example we could write

```
soft := RND(0,1)
```

and the conditional expression in:

```
IF soft = 1 THEN
```

would take the value TRUE or FALSE. In order to improve the readability of a program a special arrangment is made in COMAL:

(1) Two predefined constants TRUE, FALSE are provided and given the values 1 and 0 respectively.
(2) A numerical expression placed where a conditional expression ought to be is considered to be
FALSE if it has the value zero and
TRUE if it has any other value.

This enables us to write, simply:

```
soft = RND(0,1)
IF soft THEN
```

or, if it is preferred:

```
soft = RND(0,1)
IF soft = TRUE THEN
```

Correct program segments can now be given.

```
soft = RND(0,1)
IF soft THEN
   PRINT "Dig a third-metre hole."
ELSE
   PRINT "Dig a quarter-metre hole."
ENDIF
```

```
tree = RND(0,1)
IF tree THEN
   PRINT "Get Fred with the tractor."
   PRINT "Dig out root."
ENDIF
```

```
tree = RND(0,1)
IF tree THEN PRINT "Get Fred."
```

It is important to realise that in the first two segments any legitimate program elements can replace the PRINT statements. For example loops can be placed within decisions just as decisions might be placed within loops.

Example 4.2

A bank charges its customers at the rate of £0.20 per transaction and debits the accounts each quarter. If the average balance for the quarter exceeds £100 the charge is halved and if it exceeds £200 there is no charge for the quarter. Compute the bank charges for five customers and print the total of their bank charges. The data in pairs gives number of transactions and average balance for each of the five customers.

91, 160, 72, 490, 98, 17, 64, 101, 86, 150

METHOD

The data will be read in pairs and the bank charge will be calculated.

PROBLEM ANALYSIS/PROGRAM DESIGN

There are several choices but a reasonable way of computing the bank charge is:

```
(1) Compute charge
(2) IF balance > 200 THEN
       charge := 0
    ELSE
       IF balance > 100 THEN charge := charge * 0.5
    ENDIF
```

PROGRAM

```
sum := 0
FOR cust := 1 TO 5 DO
  READ trans, bal
  charge := trans * 0.2
  IF bal > 200 THEN
    charge := 0
  ELSE
    IF bal > 100 THEN charge := charge/2
  ENDIF
  sum := sum + charge
NEXT cust
PRINT sum
DATA 91,160,72,490,98,17,64,101,86,150
```

COMMENT

It seems curious to fix a charge and then possibly alter it. There are other ways of handling this example but further techniques are required. One of these techniques is the concept of a **compound conditional expression.** If we list the three possible cases we can see how this happens:

```
IF bal > 200 THEN charge := 0
IF bal > 100 AND bal <= 200 THEN charge := trans * 0.1
IF bal <= 100 THEN charge := trans * 0.2
```

The second possibility is governed by two simple conditional expressions joined by AND to make a compound conditional expression. The word AND is called a logical or Boolean operator. Such concepts are a valuable aid to programming because, paradoxically, programming is simpler and more straightforward if compound conditional or Boolean expressions are allowed than if only simple ones can be used. We shall examine these ideas in more detail.

4.2 CONDITIONAL EXPRESSIONS

We have seen that a conditional expression in an IF or UNTIL statement takes the value TRUE or FALSE. Sometimes we require combinations of conditions and COMAL allows the construction of compound conditional expressions using AND, OR, NOT and brackets. Precise mathematical rules or truth tables are sometimes given as definitions for these words but here they will be explained by means of examples. Intuition is usually safe but it must be remembered that the computer version of OR is the inclusive one.

OR

```
IF day$ = "Saturday" OR day$ = "Sunday" THEN
  PRINT "Weekend"
```

```
IF sex$ = "Female" OR age < 14 THEN
   PRINT "Allowed in playground"
```

Note that a twelve year old girl would be allowed in the playground. Both the conditions "Female" and "under 14" are TRUE which makes the whole conditional expression TRUE. Equally, a thirty year old woman would be allowed in though she makes only one of the conditions TRUE.

AND

```
IF sex$ = "Female" AND age < 14 THEN
   PRINT "Allowed in playground"
```

In this case the mother would not be allowed in the playground. It is strictly for female children.

NOT

This makes the value of a conditional expression the opposite of what it would be standing alone.

```
IF NOT (sex$ = "Female" AND age < 14) THEN
   PRINT "Keep out of playground"
```

This is a warning to all males and older girls or women to keep out of the playground. Note also the use of brackets to make the meaning exactly what was intended. Without the brackets the NOT would apply only to the simple expression

```
sex$ = "Female"
```

and the compound expression

```
NOT sex$ = "Female" AND age < 14
```

is true for males under fourteen and they must keep out of the playground. But males over fourteen would not be excluded. We will give some more formal definitions of these concepts in terms of *logic blocks*, devised by Z. P. Dienes to teach children about logic and other matters. They are also excellent for older students.

A logic block may be red, blue, or yellow- thick or thin; large or small- its shape may be a circle, square, triangle or oblong. We shall use the symbols in Fig. 4.1.

Thus the symbol 

means a small, red, thick square.
The following statements might be made about this block.

Red AND square (True)
Blue AND square (False)
Small AND thick (True)

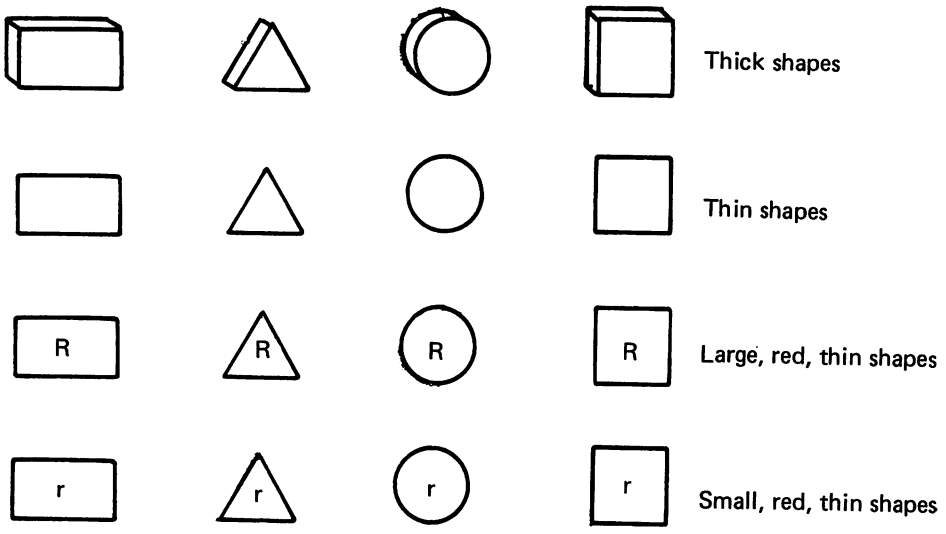


Fig. 4.1 – Symbols for logic blocks.

It can be seen that a compound statement with AND is true only when both the simple statements are true but

Blue OR square (TRUE)

shows that the operator OR has a different effect.

The following tables show all the possible fundamental combinations.

OR

Block	*Simple Expressions*		*Compound Expressions*
	Red	*Square*	*Red OR Square*
r	True	True	True
r	True	False	True
b	False	True	True
b	False	False	False

In programming OR leads to a true compound statement if both simple statements are true. This is sometimes referred to as the *inclusive* OR. This is not always the same as the everyday usage. "I will take a bus or go by taxi" is an

example of using OR in the exclusive sense – one or the other but not both. "If you want to do computer studies you must pass 'O' level English or mathematics." is an inclusive usage – one or the other or both will be acceptable evidence of computing potential. (As a matter of interest NCC studies have shown O level English language to be a better predictor of success in introductory computing courses than mathematics.)

AND

Block	*Simple Expressions*		*Compound Expressions*
	Red	*Square*	*Red AND Square*
r	True	True	True
r	True	False	False
b	False	True	False
b	False	False	False

There is little room for confusion here. An AND expression is true only if both its component simple expressions are true. Otherwise it is false. Note that AND and OR involve two simple statements and they are called *binary operators.* NOT applies to a single entity and is called a *unary* operator. That single entity may be a compound expression but it must have a single value.

NOT

Block	*Red*	*NOT Red*
r	True	False
b	False	True

PROBLEMS

4.1 Considerable educational value and fun can be gained from a set of logic blocks which might be bought, made or borrowed from a local junior school but the following exercise should be a good test of whether these ideas are understood. Place a sheet of paper over the answers and reveal an answer only when True or False has been decided for each line. Look at each logic block and decide whether the compound expressions about it are true of false.

B = Blue R = Red Y = Yellow T = True F = False

	Round AND Red	T
	Round AND Blue	F
	Round AND Red	F
	Square AND Red	F
	Round OR Blue	T
	Round OR Blue	T
	Round OR Yellow	T
	Square OR Red	F
	Red	T
	Red	F
	NOT Round	F
	NOT Blue	F
	NOT Blue	T
	NOT Round	T
	NOT (Round AND Red)	F
	NOT (Round AND Blue)	T
	NOT (Round OR Yellow)	F
	NOT (Square OR Red)	F
	NOT Round OR Yellow	T
	NOT Square OR Red	T
	NOT Round OR NOT Blue	T
	NOT Round AND NOT Blue	T

4.2 Ascribe a value true (T), false (F) or neither (N) to the following statements.

Blood is red AND grass is green	T
Blood is red AND circles are square	F
Blood is red OR grass is green	T
Blood is red OR circles are square	T
name$:= "Term"	N
four := 4	N
This statement is false	N

4.3 WORKED EXAMPLES ON BINARY DECISIONS

Example 4.3

Set up a string variable *letter*$ and place thirty random upper case letters in it. Invite the user to search or change. If he wishes to search ask which letter he wants and then search for each occurrence of the letter and print the position numbers if any. If he wishes to change a character ask for character position and new letter. Do not cater for incorrect input at any stage.

METHOD

The method is straightforward. We will use indexing to test each character in the string, or to replace an existing character.

PROGRAM

```
DIM letter$ OF 30, ch$ OF 1, seek$ OF 1
FOR k := 1 TO 30 DO letter$ (k:1) := CHR(RND(65,90))
INPUT "Type S or C and RETURN" : ch$
IF ch$ = "S" THEN
   INPUT "What is the letter you seek?" : seek$
   FOR pos := 1 TO 30 DO
      IF seek$ = letter$ (pos:1) THEN PRINT pos
   NEXT pos
ENDIF
IF ch$ = "C" THEN
   INPUT "Which position?" : pos
   INPUT "Which character?" : ch$
   letter$ (pos:1) := ch$
ENDIF
```

Example 4.4

Write down the acceptable inputs for each of the four INPUT statements in the above program and rewrite the program in such a way that a legitimate operation

is completed or the program may be terminated by the user. Allow only upper case letters to be inserted.

METHOD
(1) S for Search, C for Change or E for End will be acceptable.
(2) A letter for seeking must be in the range "A" to "Z".
(3) Position numbers requested must be in the range 1 to 30.
(4) A letter for insertion must be in the range "A" to "Z".

PROBLEM ANALYSIS/PROGRAM DESIGN
Each INPUT statement will be placed in a REPEAT loop from which exit is determined by a legitimate response. Otherwise the program follows the previous one fairly closely.

PROGRAM

```
DIM letter$ OF 30, ch$ OF 1, seek$ OF 1
FOR k := 1 TO 30 DO letter$ (k:1) := CHR(RND(65,90))
REPEAT
   INPUT "Type S or C or E and RETURN" : ch$
UNTIL ch$ = "S" OR ch$ = "C" OR ch$ = "E"
IF ch$ = "E" THEN STOP
IF ch$ = "S" THEN
   REPEAT
      INPUT "What is the letter you seek?" : seek$
   UNTIL seek$ >= "A" AND seek$ <= "Z"
   FOR pos := 1 TO 30 DO
      IF seek$ = letter$ (pos:1) THEN PRINT pos
   NEXT pos
ELSE
   REPEAT
      INPUT "Which position?" : pos
   UNTIL pos >= 1 AND pos <= 30
   REPEAT
       INPUT "Which character?" : ch$
    UNTIL ch$ >= "A" AND ch$ <= "Z"
    letter$ (pos : 1) := ch$
ENDIF
```

Some COMAL 80 systems offer the IN facility which searches a string for any substring automatically. For example, the search FOR loop could be replaced but it would find only the first occurrence, if any:

```
pos := seek$ IN letter$
IF pos > 0 THEN PRINT pos
```

The expression:

```
seek$ IN letter$
```

takes the position number of the first character of the character of the substring if it is found. Otherwise the value of the expression is zero. This means that it can also be used as a conditional expression:

```
IF seek$ IN letter$ THEN
```

However, the techniques of searching are of fundamental importance and we will use this simple example without using IN to explore the idea of searching a little further.

Example 4.5
Partially fill the string letter$ with a random selection of upper case letters in order. Terminate the sequence with "*". Invite the user to search for a letter by starting it or typing ">" to exit from the program.

PROGRAM

```
DIM letter$ OF 30, ch$ OF 1
pos := 0
FOR code := 65 TO 90 DO
   IF RND(1,3) > 1 THEN
      pos := pos + 1
      letter$(pos : 1) := CHR(code)
   ENDIF
NEXT code
letter$ (pos+1 : 1) := "*"
REPEAT
   INPUT "Enter letter or > " : ch$
UNTIL (ch$> = "A" AND ch$ <="Z") OR ch$ = ">"
IF ch$ = ">" THEN STOP
   pos := 0
REPEAT
   pos := pos + 1
UNTIL letter$ (pos:1) = ch$ OR letter$ (pos:1) = "*"
IF letter$ (pos:1) = "*" THEN
   PRINT "Not found"
ELSE
   PRINT "found at position"; pos
ENDIF
```

The important point here is that the compound condition in the search loop is essential to make sure that it does terminate. We might have, for example.

In certain situations it may be undesirably inefficient to perform a long search in which two conditions are tested each time. A trick known as "posting a sentinel" is used to avoid this. Suppose that the search letter is "L". This letter is placed at the end of the string in place of "*" and the search condition can be made simple:

```
UNTIL letter$ (pos : 1) = ch$
```

in the knowledge that if "L" is not found in the body of the string it will be certain to stop the search at the end. After exit from the search we can write something like:

```
IF pos = LEN (letter$) THEN
   PRINT "Not found"
ELSE
   PRINT "Found at position"; pos
ENDIF
```

Another way of handling a search condition in the case where items are ordered is to make the condition:

```
UNTIL letter$ (pos : 1) >= ch$
```

and ensure that the terminating symbol in the string has an internal code greater than that of "Z". The symbol "*" would not work but "↑" would.

In this case on exit from the search loop we would need something like:

```
IF letter$ (pos : 1) = "↑" THEN
   PRINT "Not found"
ELSE
   PRINT "Found at position" ; pos
ENDIF
```

PROBLEMS

4.3 Find the largest of three numbers given in a DATA statement. (7)

4.4 Find the alphabetically first name of five names in a DATA statement. (8)

4.5 Generate thirty random numbers in the range 1 to 100 and find the range (smallest and largest) of the generated numbers. (10)

4.6 In production runs of microprocessor chips the probability of a defective chip is 60%. Simulate a run of 400 applying this probability to each chip and find the number of good chips in the run. (7)

4.7 Six workers are entitled to bonuses of £20. if production on an individual's machine exceeds forty units in a week. The basic wages and production figures for the six workers are:
120, 42, 132, 46, 125, 31, 130, 36, 120, 44, 140, 52.
Compute the six gross wages. (7)

4.8 Rail sections are specified as 5 metres long with variations 3 millimetres either way being acceptable. An electronic measuring device provides the following data which a computer program reads and either allows a section to proceed or causes it to be pushed off the conveyor belt. Read the ten numbers and print "OK" for an acceptable length, otherwise "Push off.".
5.0017, 5.0042, 4.9978, 4.9991, 5.0005
5.0021, 4.9983, 4.9997, 4.9970, 5.0030 (8)

4.9 An automatic warning "No Bathing" is provided at a beach when the wind speed reaches an unsafe level. This is judged to be when the average of the last three readings exceeds 25 km/hour. When the wind recorder is switched off the warning is displayed and remains displayed until at least three readings have been taken. At this point a sign saying "Bathing Allowed" may be displayed. Give the appropriate sign for each of the following readings assuming the wind recorder was switched off before the first one was taken. A negative reading means the recorder is switched off.
13, 15, 26, 24, 28, 30, 27, 22, 21, 19, 17, 14, −1. (10)

4.10 Pythagoras' theorem enables the sloping side (hypotenuse) of a triangle to be computed, if the length of the opposite and adjacent sides are known, from the formula

```
hyp := SQR(opp ↑ 2 + adj ↑ 2)
```

The following pairs of numbers are length of *opp* and *adj* for six right angled triangles. Compute the length of the six hypotenuses and find the largest.
1, 6, 2, 5, 3, 4, 4, 3, 5, 2, 6. (7)

4.11 The diagram shows simplified possible routes from Reading to Sheffield by joining the London to Sheffield motorway at points 10, 20, 30 . . . miles along. Across country on the first leg a motorist's average speed is 45 miles per hour but on the motorway he averages 60 miles per hour. Compute the journey times for all routes joining the motorway at 10, 20, 30 . . . 150 miles along and find the quickest route. See Fig. 4.2.
(9)

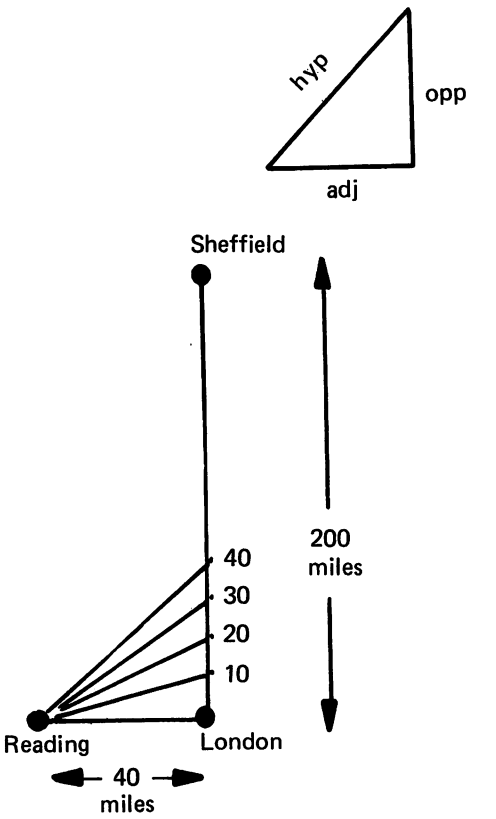


Fig. 4.2 – Routes from Reading to Sheffield.

4.4 MULTIPLE DECISIONS – CASES

The foreman (Example 4.1) required that his workman should use three different types of concrete support depending on the level of the ground. This involves the simplest type of decision beyond the binary one. It is best treated by means of a CASE statement which caters for any number of choices. The program should explain itself.

```
level := RND(1,3)
CASE level OF
WHEN 1
   PRINT "Use a one-fifth metre support"
WHEN 2
   PRINT "Use a two-fifth metre support"
WHEN 3
   PRINT "Use a three-fifth metre support"
ENDCASE
```

Again it should be noted that control can only enter the structure at the opening keyword, CASE, and it can only depart at the closing keyword, ENDCASE.

This program segment will be inserted in the final version of the program which performs the complete simulation.

The need for a CASE statement arises in slightly more complex problems concerning a simulation of "Drunken Duncan" moving randomly about the screen knocking over "milk bottles".

Example 4.6

An array of 21 X 21 dots is to be placed on the screen. These are to be regarded as milk bottles. Drunken Duncan starts in the centre and moves randomly north, south, east or west knocking over milk bottles as he goes. This continues until he reaches the edge of the array.

Write a program to simulate the actions of Drunken Duncan.

METHOD

(1) The screen consists of 25 rows of 80 columns. The array will be carefully placed but there will be a space between each dot along rows to make the array more like a square.
(2) The first dot will be in row 3 and column 21. The last dot will be in row 23 and column 61.
(3) Duncan will start at position:

row(down) 13
column(across) 41

(4) As Duncan moves the dot in his square it will be replaced by a space so that the area of his movements will be recorded.
(5) Duncan will be denoted by "*" which will overprint the dot, but the asterisk will immediately be overprinted by a space to show that the "milk bottle" (dot) has disappeared from that position.

PROBLEM ANALYSIS/PROGRAM DESIGN

(1) Since many screens are line oriented it is probably faster to print 21 lines of 21 "dot space" pairs rather than print 21 dots individually. The line will be placed in dot$ OF 42.

```
FOR line := 3 TO 23 DO PRINT TAB(20); dot$
```

(2) Initialise Duncan at (41,13)
(3) PRINT "*"
(4) The coordinates ac, dn will be used and a random number will determine the direction.

```
dir := RND(1,4)
CASE dir OF
WHEN 1
   dn := dn–1                (North)
WHEN 2
   ac := ac+2                (East)
WHEN 3
   dn := dn+1                (South)
WHEN 4
   ac := ac–2                (West)
ENDCASE
```

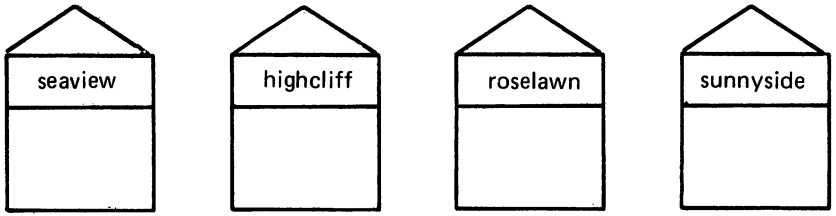
(5) Before entering the CASE structure a space will be printed in the correct screen position, and the routine will be repeated UNTIL Duncan is off the 21 X 21 grid.

```
PROGRAM
        DIM dot$ OF 42
        RANDOMIZE
        dot$ := "......................"
        EXEC cursor(1,3)
        FOR line := 3 TO 23 DO PRINT TAB(20); dot$
        ac := 41; dn := 13
        REPEAT
          EXEC cursor(ac,dn)
          PRINT "*"
          dir := RND(1,4)
          EXEC cursor(ac,dn)
          PRINT " "
          CASE dir OF
          WHEN 1
            dn := dn–1
          WHEN 2
            ac := ac + 2
          WHEN 3
            dn := dn + 1
          WHEN 4
            ac := ac–2
          ENDCASE
        UNTIL ac < 21 OR ac > 61 OR dn < 3 OR dn > 23
```

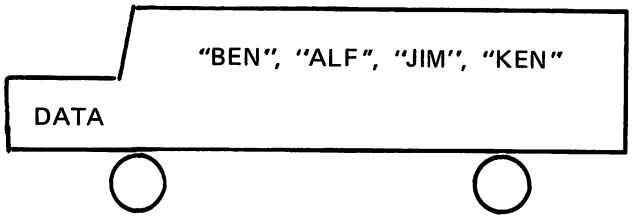
In this program we have used the general concept of an array of dots on the screen. We may wish to print a record of Duncan's path in which case we need to use the concept of an array as a coherent set of variables or **data structure.**

4.5 VARIABLE VARIABLES – ARRAYS

Suppose we have four boarding houses in Brighton and a travel agent wishes to place four holiday makers, one in each

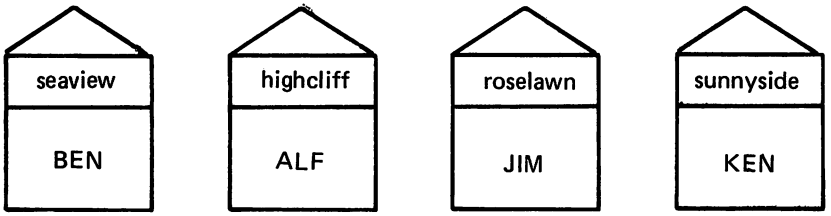


The holiday makers arrive in a DATA statement.



The agent programs the holidaymakers into the houses.

```
READ seaview
READ highcliff
READ roselawn
READ sunnyside
```



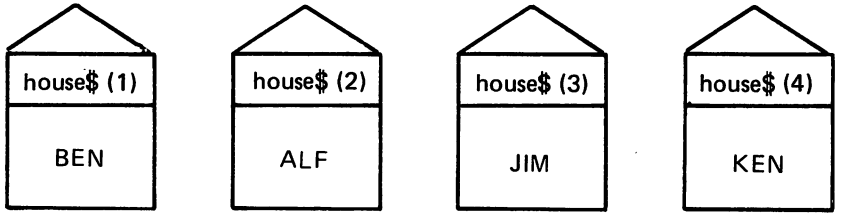
Thus we have assigned values (holidaymakers) to the variables (houses), but if there were forty or four hundred holidaymakers and houses a better method would be needed. We wish to do the same process to a sequence of houses – we

want the variable in the process to vary. An array enables this to be done. We declare four houses (string arrays) of three characters each.

```
DIM house$(4) OF 3
```

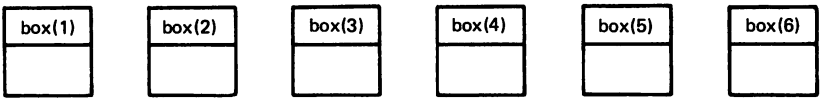
Assuming the same DATA as before a simple FOR loop will place a holiday-maker in each house

```
FOR number :=1 TO 4 DO
   READ house$(number)
   PRINT house$(number);" ";number
NEXT number
```



nb The notation house$(2) when referring to a string array address means the whole string.

Consider the numerical problem of simulating one hundred throws of a die and counting the frequency of each score. We set up an array of six (numeric) boxes,



When a score is generated we increase the count in the corresponding box. We could proceed as follows.

```
FOR throw = 1 TO 100 DO
   die = RND(1,6)
   IF die = 1 THEN box(1) := box(1)+1
   IF die = 2 THEN box(2) := box(2)+1
   IF die = 3 THEN box(3) := box(3)+1
   IF die = 4 THEN box(4) := box(4)+1
   IF die = 5 THEN box(5) := box(5)+1
   IF die = 6 THEN box(6) := box(6)+1
NEXT throw
```

It will be observed that the box number is always the same as the die value and we can use this fact to write the program more concisely.

```
FOR throw := 1 TO 100 DO
  die := RND(1,6)
  box(die) :=box(die)+1
NEXT throw
```

To complete the program we need a DIM statement at the beginning.

```
DIM box(6)
```

and a PRINT loop at the end:

```
FOR value := 1 TO 6 DO
  PRINT box(value)
NEXT value
```

The FOR loops in this program are worthy of careful study. Remember that the static text represents a dynamic process in time when the program runs. As the loop count proceeds *box(die)* or *box(value)* change their meaning. At the first execution of a PRINT, *box(value)* refers to *box*(1) but at the last iteration the reference is to *box*(6). As the loop progresses the variable itself varies. This most powerful concept enables huge quantities of data to be handled with relative ease.

Example 4.7

A binary search or binary chop can be performed on any ordered sequence of data in a suitable structure. Instead of searching linearly from the start, the middle item is examined and the sequence is 'chopped' into two halves one of which is known to contain the required item if it exists. The process is repeated until the item is found.

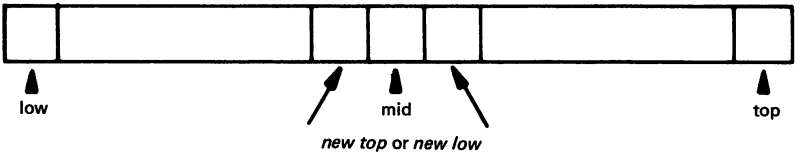
Set up an array of about one hundred ordered numbers in the range 1 – 200 and search them for a specified number.

METHOD

(1) A number will be placed in the current array element if RND(0,1) = 1

BINARY SEARCH

(2) Variables low, top and mid will define the upper, lower and mid points of the array segment containing the required number.
(3) mid := (low + top) DIV 2 will compute the mid point.
(4) The mid point will be tested and if the required number is not there the following changes are made.



```
IF num(mid) > req THEN
the required number is in the left half and
top := mid − 1
IF num(mid) < req THEN
the required number is in the right half and
low := mid + 1
```

(5) If, when a new low or top has been defined, low <= top the search can proceed but if low > top it means that the number does not exist in the array.

PROBLEM ANALYSIS/PROGRAM DESIGN

(1) After setting up and initialising low and top the process described above will take place in a loop of the form:

```
found := FALSE
REPEAT
   mid := (low+top) DIV 2
   IF num(mid) = req THEN
      found := TRUE
   ELSE
      (alter boundaries)
   ENDIF
UNTIL found OR low > top.
```

(2) On exit from this loop the result will be printed.

PROGRAM

```
DIM num(150)
RANDOMIZE
pos := 0
FOR value := 1 TO 200 DO
   IF RND(0,1) = 1 THEN
      pos := pos + 1
      num(pos) := value
   ENDIF
NEXT value
low := 1;top := pos
INPUT "Search number?" : req
REPEAT
   mid := (low + top) DIV 2
   IF num(mid) = req THEN
      found := TRUE
   ELSE
      IF num(mid) > req THEN
         top := mid−1
```

```
      ELSE
        low := mid+1
      ENDIF
    ENDIF
UNTIL found OR low > top
IF found THEN
  PRINT "Number found at position"; mid
ELSE
  PRINT req; "not found."
ENDIF
```

TESTING

Assume that the required number, 17, does not exist at position 8 and the numbers in the array start with 2 at position 1 and end with 200 at position 102.

The values of low, top and mid in the binary search as shown below.

low	top	mid	
1	100	50	
1	49	25	
1	24	12	
1	11	6	
7	11	9	
7	10	8	found in 6 attempts

COMMENT

A simple linear search would find an existing number in about fifty attempts on average for an array of one hundred elements. The search time is roughly proportional to ½n where n is the size of the array.

The binary search is a "halving" process. It diminishes the area of search as fast as doubling increases the size of a number. Consider the doubling process:

1 2 4 8 16 32 64 128

The associated powers of 2 are:

2^1 2^2 2^3 2^4 2^5 2^6 2^7

It is not a coincidence that the **searchlength** of six attempts is about the same as the index of the power of two which matches the array size (approximately). Another name for index is logarithm (base 2) or log (base 2). Whether one calls it the log (base 2) of n or the index of the power of two which matches

n, it is clear that this number is a remarkable improvement on ½n. This makes the binary search much faster than a linear search, particularly with large numbers. For example log (base 2) of 1000 is only 10. Of course the binary search can only be used on ordered data, but the overhead of extra computation is not high as an examination of the program will show.

The concept of binary chopping and the associated expression log n is useful in computing. Non-mathematical readers who may have believed that logarithms are complex things need not worry. They are only indices. It is a pity that a separate word, logarithm, which seems to evoke mystery, was ever invented. The logarithms at school are based on ten rather than two but the idea is the same. In a computing context log n will usually imply "based on 2" unless it is stated otherwise.

Example 4.8

Armed with some understanding of arrays we can now return to the problem of printing out the results of a run of the Drunken Duncan program.

METHOD

(1) We will set up a string array

```
store$(21) OF 42
```

in which we shall initially store the dot-space pairs of characters.

(2) The "*" representing Duncan is no longer needed but the spaces to replace dots are.

(3) A print loop at the end will display all the 21 lines of string data.

PROBLEM ANALYSIS/PROGRAM DESIGN

(1) A slight complication is the fact that we are now indexing into an array element.

```
store$(7,k:1)
```

will now mean the kth character in row number 7 or line number 7 of the array.

(2) The position references now apply to an array: 21 lines of 42 characters rather than a screen of 24 X 80.

PROGRAM

```
DIM dot$ OF 42, store$(21) OF 42
RANDOMIZE
dot$ := ". . . . . ."
FOR line := 1 TO 21 DO store$(line) := dot$
ac := 21; dn := 11
REPEAT
```

```
    store$(dn, ac:1) := " "
    dir := RND(1,4)
    CASE dir OF
    WHEN 1
      dn := dn-1
    WHEN 2
      ac := ac+2
    WHEN 3
      dn := dn+1
    WHEN 4
      ac := ac-2
    ENDCASE
  UNTIL ac < 1 OR ac > 42 OR dn < 1 OR dn > 21
  FOR line := 1 TO 21
    PRINT TAB(20); store$(line)
  NEXT line
```

PROBLEMS

4.12 Place two randomly generated letters into an array and ensure that they are in alphabetical order by testing and exchanging if necessary. (6)

4.13 Overseas letters from UK were charged (1981) at the following rates,

	10g	Each extra 10g
Zone A	20p	11p
Zone B	22p	14p
Zone C	25p	15p

Write a program to accept zone and weight for a letter and compute the postal charge.

4.14 In the ancient game of "Morra" two players place one or two fingers on the table and money changes hands according to the scheme adapted for a computer.

	Computer	
Player	1 finger	2 fingers
1 finger	Player pays 1	Computer pays 2
2 fingers	Computer pays 2	Player pays 3

Write a program to play the game, generating the computer's play randomly, and give the result. (7)

4.15 Generate thirty numbers in the range 1 to 100 and store them in an array. Find the largest and smallest numbers by scanning the array. (10)

4.16 A queue is to be represented by an array of twelve elements. When an item arrives its arrival time is placed in the next empty place. The variable, *front*, always indicates the front of the queue and the variable, *next*, always indicates the next empty place. The following pairs of data represent arrivals or departures. Assuming that the queue is empty at the start READ each data pair and take appropriate action. When an item departs print its total time in the queue.

1, "A", 3, "A", 7, "A", 10, "A", 15, "A"
16, "D", 18, "D", 20, "A", 21, "D", 23, "D" (21)

4.17 A stack in computing is like a stack of plates: items may be placed on or removed from the top of the stack and from nowhere else. Items are said to be *pushed* on to the stack or *popped* off it. Write a program which enables a user to push and pop numbers. Terminate the program when a user tries to pop an item off an empty stack, or if the stack fills the array. (15)

4.18 In the childrens' game of "Paper, Scissors, Stone", two players simultaneously make a shape, with one hand, to represent one of the three items. The winner is decided by the following impeccable logic:

Scissors cut paper.
Paper wraps stone.
Stone blunts scissors.

In some parts of England the words used are "Scis, Pap, Brick". Write a program which invites the user to type one of:

S for scissors
P for paper
B for brick or stone

Generate the computer's choice in a random manner and print a message saying who has won and why. (12)

4.19 Guests in a hotel order breakfast by ticking any of twelve items on a card which is then hung outside the room. The following data gives the prices of the twelve items in Swiss francs, followed by three sets of items, each set terminated by −1 representing three orders. Compute the charge for each guest. (15)

1, 1.20, 1.80, 2, 1.50, .80, .60, 1.30, 1, 2, 1.30, 1.40
2, 6, 8, −1, 1, 2, 5, 7, −1, 3, 6, 8, −1

4.20 Refer to problem 4.12 for a method of exchanging the values of two variables and use it to sort ten numbers into ascending order in the following way.

(1) Place ten numbers in an array.

(2) Compare items 1 and 2 and exchange if necessary.
(3) Compare items 2 and 3 and exchange if necessary.
(4) Continue comparing and exchanging up to the pair 9 and 10.
(5) Examine the effect of this process on the ten numbers and call it one "run".
(6) Deduce that nine such runs would cause the numbers to be sorted.

Write a program to do this. Print out the sorted list. (16)

4.21 Concrete is mixed in three ways for foundations, paths or bedding mortar.

	Sand	*Cement*	*Aggregate*
Mix A	2½	1	4
Mix B	2	1	3
Mix C	3	1	0

Write a program which accepts as input a letter A, B or C and a number representing the weight required in tons.

Compute the required quantities of sand, cement and aggregate and print them. (10)

CHAPTER 5

Modularity and Procedures

> The good writers use identifiable, controlled patterns of prose, producing a book whose hierarchial structure is elegantly displayed by the table of contents.
>
> Maurice Wilkes, reported by Peter J. Denning
> *ACM Computing Surveys*, December 1974.

5.1 SIMPLE PROCEDURES

Size or complexity are problems in computing as they are in many other human endeavours. The building of an aeroplane, the control of military operations, the government of a country or anything to do with space vehicles are just a few examples. A Prime Minister or President copes with an otherwise impossible task by breaking it down into sub-jobs: health, finance, law and order, education, industry and so on. Each minister or secretary responsible for an area will break it down further into such things as policy, staffing, buildings, organisation. Thus problems or jobs are made more manageable as they are broken down until the parts are small enough for useful action by individuals or small teams.

Some hierarchical systems may be more elitist than others but the breaking down creates the need for explicit relationships between the pieces, and the hierarchal 'tree' structure is the one that seems to work well in a variety of situations. The Knights of the Round Table may have coped with relatively simple tasks like rescuing ladies or fighting other knights but they did not found an empire.

In computing we can take one part of a job, concentrate on dealing with it, give it a name and forget the details. Such a sub-job or module may form a **procedure**. We need only remember the name and in general terms what the procedure does, thus freeing the mind to concentrate on other details. The simplest form of a procedure can be illustrated by the post-holes problem. The job of dealing with a tree in the way of a post-hole can be simulated as shown.

```
PROC treechop
  PRINT "Knock down tree."
  PRINT "Dig out roots."
ENDPROC
```

The opening and closing keywords PROC and ENDPROC isolate this bit of code, creating a sort of island in the program. There may be many such islands, and the only way of gaining access to any of them is by using the word EXEC (for execute) followed by the particular procedure name. Thus:

```
EXEC treechop
```

would activate the relevant procedure and after its execution, control would continue with the statement after EXEC treechop.

There are two obvious advantage of procedures. On is to do with breaking a complex or large program into 'modules', each reasonably easy to comprehend. The other is that the same job may be required in different sections of a program. Since there are no restrictions on the number of times a procedure may be activated it need only be written once and called or executed from wherever necessary in the program. Moreover it should be possible to use the same procedure in more than one program if it is appropriate, thus avoiding further duplication of effort.

Natural questions are what is meant by 'large' and what is meant by 'complex'? Kenneth Bowles (*Problem Solving Using Pascal,* Springer Verlag, 1977) gives an answer.

> . . . a program should be broken up into small groups of statements (sometimes called 'modules' or 'sub-routines'), each of which can be thought of as performing a single action. Inside such a group, several statements must be performed in order to cause the required action to take place. However, once the group has been written, you can think of it as one unit and forget the fact that it really consists of several independent statements. When you write a program containing more lines than will fit easily on one page, say 25 to 50 lines, it gets too complicated to think of as a simple unit. Then it is time to think of breaking the program up into separate modules.

There are two differences between the context of Bowles statement and ours. His students were doing Pascal and they were expected to achieve the standards usual for university undergraduates. Our language is COMAL and there is very little overhead or difficulty in the use of COMAL procedures. Certainly they are easier to handle than those of Pascal. Secondly we are aiming for simplicity and ease of comprehension. Our answer might be that procedures should be used very frequently indeed. Typically a program might consist mostly of procedures with a small main program doing little more than a linking operation.

If anyone has difficulty comprehending precisely what happens when a FOR loop is nested inside another it may help to see the identical job done with a procedure. The program which writes a rectangle of stars (section 3.2) could be re-written:

```
FOR down := 6 TO 10 DO
   EXEC starline
NEXT down
PROC starline
   FOR across := 21 TO 27 DO
      EXEC cursor(across, down)
      PRINT "*"
   NEXT across
ENDPROC
```

While the use of a procedure in this example is unusual and certainly unnecessary, it does no harm to see things from different points of view.

5.2 PROCEDURE PARAMETERS

5.2.1 Actual Parameters and Formal Parameters

There is one other thing to be learned from this simple example. The value of the control variable, *down,* is determined in the main program and used in the procedure. Information has been passed to the procedure without any special arrangements being made. The variable, *down,* is simply used where it is needed. However, more formal arrangments for transferring information are sometimes made. Consider again the foreman's instructions. Suppose that, Fred, the man with the bulldozer and other tree felling equipment, has three methods of dealing with trees which he classifies:

Large tree: Cut it down and dig out the root.
Medium tree: Cut it five feet above the base and pull out the root.
Small tree: Pull it out.

The procedure, *treechop*, needs to know which type of tree to deal with. A word, large, medium or small, needs to be passed to the procedure. This can be done as shown, assuming a large tree.

```
EXEC treechop("large")
```

The word "large" is called an **actual parameter** because it is real information as opposed to the **formal parameter**, treetype$, in the procedure definition.

```
PROC treechop(treetype$)
   CASE treetype$ OF
   WHEN "large"
      PRINT "Cut down tree. Dig out root."
   WHEN "medium"
      PRINT "Cut above base. Pull out root."
   WHEN "small"
```

```
      PRINT "Pull out tree."
    OTHERWISE
      PRINT "Tree type not known."
    ENDCASE
  ENDPROC
```

The formal parameter, treetype$, is called a **value parameter** because, in effect, at execution time a value, "large", is simply substituted for any occurrence of treetype$ in the body of the procedure. To put it another way, the value "large" is assigned to the variable treetype$ at the start of execution of the procedure. Parameters help to clarify just what information is passing between the main program and a procedure but there is more to be gained from their use. To fix ideas and further the discussion let us consider an example which helps psychologists with memory tests.

Example 5.1 Numbers for Psychology Testing
A psychologist requires eight sets of random digits in the range 0 – 9 for short-term memory tests. In each set there should be ten lines of numbers starting with three to a line and increasing to twelve to a line, followed by another ten lines which start at twelve and decrease to three.

PROBLEM ANALYSIS/PROGRAM DESIGN
(1) Suppose we have a procedure *lindig* which prints *num* digits on a line.
(2) This procedure should be called by a FOR loop which varies *num* from 3 – 12 and then by a second FOR loop which does the reverse.
(3) The whole should be within a FOR loop which does the job eight times.

Note
A FOR loop can have a STEP element which determines the way the control variable is incremented when the increment is different from one.

PROGRAM

```
RANDOMIZE
FOR set := 1 TO 8 DO
  FOR num := 3 TO 12 DO
    EXEC lindig
  NEXT num
  FOR num := 12 TO 3 STEP -1 DO
    EXEC lindig
  NEXT num
  PRINT
NEXT set
PROC lindig
```

```
    FOR digit := 1 TO num DO
      PRINT RND(0,9);
    NEXT digit
    PRINT
ENDPROC
```

SAMPLE OF OUTPUT

One of the eight sets of twenty lines is shown in Fig. 5.1.

```
2 0 4
3 9 4 8
4 3 8 2 1
5 8 2 9 1 4
8 0 6 1 5 2 9
5 8 0 0 3 1 4 8
7 3 3 8 0 6 0 1 5
6 1 4 7 7 7 3 9 6 4
9 3 5 4 3 2 9 4 9 4 1
0 9 5 7 3 0 2 7 6 0 9 5
9 3 4 1 6 5 4 2 4 0 2 7
5 4 8 9 8 5 5 2 8 2 2
4 3 6 8 9 0 3 5 0 2
5 5 7 3 3 3 3 5 9
5 0 1 6 3 2 1 2
0 6 2 3 3 9 9
7 0 0 4 6 6
2 6 0 3 5
3 0 6 2
8 0 3
```

Fig. 5.1 – Sample of output.

5.2.2 Local Variables (parameters)

Note that the procedure, *lindig* needs to know how many digits to produce each time it is called to print a line. This information is passed to the procedure by means of the variable, *num.* An alternative treatment would be to recognise that the procedure needs one piece of information and to provide a formal parameter, *numdig.*

```
PROC lindig(numdig)
    FOR digit := 1 TO numdig DO
      PRINT RND(0,9);
    NEXT digit
    PRINT
ENDROC
```

The main program remains the same except that the two procedure calls become:

```
EXEC lindig(num)
```

An advantage of this version is that the variable, *num*, in the main program cannot be influenced by anything that happens in the procedure. The idea is that procedures should be as self-contained as possible. Ideally they should be like **black boxes** which communicate with the outside world only through their formal parameters. This means that a procedure can be more easily moved from one program to another without fear of causing trouble because the same variable name might be used for different purposes in the main program and in the procedure. A formal parameter is automatically protected by being made local to the procedure. COMAL 80 does this so that even if, in the above procedure the parameter were *num* instead of *numdig* it would not matter. The formal parameters are tagged internally by some special mark which makes them accessible only within the procedure.

To see that this is so we can rewrite the program as follows and it will run. For simplicity only the upper half of the original "triangle" of numbers is printed.

```
RANDOMIZE
FOR set := 1 TO 8 DO
  FOR num := 3 TO 12 DO
    EXEC lindig(num)
  NEXT num
  PRINT
NEXT set
PROC lindig(num)
  FOR digit := 1 TO num DO
    PRINT RND(0,9);
  NEXT digit
  PRINT
ENDPROC
```

The astute reader may observe that the actual parameter, *num*, only passes information to the formal parameter *num* and neither is used for any other purpose. It is difficult to see what could go wrong. We shall see later that results of computations within a procedure can be passed back to the main program and there could be trouble. At this stage it is only necessary to know that the names of formal parameters can be chosen without worrying about possible use of the same names in the main program.

5.2.3 Local Working Variables

The above procedure has a working variable *digit*, which is not a parameter. This

could cause trouble if the main program happened to use the same name for something else.

For example, suppose we rewrite the program using the variable, *set*, in both main program and procedure.

```
RANDOMIZE
FOR set := 1 TO 8 DO
   FOR num := 3 TO 12 DO
      EXEC lindig(num)
   NEXT num
   PRINT
NEXT set
PROC lindig(num)
   FOR set := 1 TO num DO
      PRINT RND(0,9);
   NEXT set
   PRINT
ENDPROC
```

We know that the double usage of *num* will cause no trouble, but what about the double usage of *set*? After the first ten executions of the procedure, one "triangle" of numbers has been printed and *set* has the value 13 (one above the final loop counter value). Control returns to the line "NEXT set" in the main program. But the final value of this loop counter is 8 and the program halts. Thus a working variable in a procedure has interfered with the proper functioning of the main program.

The complete closure of the procedure to the outside world is achieved by writing CLOSED. All working variables are now made local to the procedure and it becomes "safe".

The completely closed procedure is written:

```
PROC lindig(num) CLOSED
   FOR set := 1 TO num DO
      PRINT RND(0,9);
   NEXT set
   PRINT
ENDPROC
```

Note

The variable *num* is local because it is a formal parameter and cannot be confused with *num* in the main program. The variable *set* is local because the procedure is CLOSED.

All this may seem rather elaborate but the lack of properly organised procedures in BASIC has caused trouble even in fairly simple work or in the organisation of software libraries.

On the other hand, these facilities can be ignored in introductory work. If parameters are not used and the procedure is not declared CLOSED then all variables are global and can be used quite effectively. In this case one has to be careful not to use the same variable names in different parts of the program unless they refer to exactly the same thing. This is not normally a problem in introductory work though it can become one quite suddenly and unexpectedly, sometimes leading a programmer into unfortunate improvisations which lead to more trouble.

SUMMARY

(1) A procedure may have formal parameters for communicating information to or from the main program.
(2) The main program makes use of actual parameters in procedure calls.
(3) The simplest case occurs when the main program passes a value to the procedure. Such parameters are called value parameters (actual or formal).
(4) Formal parameters in COMAL 80 are automatically local to the procedure and the same variable names can be used in the main program without confusion.
(5) Working variables in COMAL 80 procedures are not automatically local and can cause trouble.
(6) The use of CLOSED will turn a procedure into a *black box* making it quite safe for use by any main program (or other procedure).

PROBLEMS

5.1 Write a procedure, *double*, which doubles the value of a variable *num*. Call this procedure from the main program until *num* exceeds one million. Do not use parameters and treat all variables as global. Include a count in the main program which records how many doublings are required. (7)
(See Problem 3.12)

5.2 Write a procedure, *double*, which receives by a value parameter a number from the main program and computes the number of doublings necessary to exceed one million. Use a global variable to pass this information to the main program. Do not close the procedure. (7)

5.3 Write a procedure which generates letters randomly until a "Z" appears. Use a variable, *count*, to count the number of letters in each execution and print the number. Call this procedure twelve times from a main program using the variable, *count*, in the FOR loop. Show that the program will fail unless the procedure is CLOSED. (9)

5.4 Write a program which accepts as input a string representing a number. Convert the number to numeric type.
(a) Assume digits only as input (4)
(b) Assume digits and possibly one decimal point as input. (11)

5.3 MODULARITY

Modularity is one of the weapons we need to defend clarity and control against the attacks of length or complexity. It would be boring for the reader to be asked to wade through a long program which may have little intrinsic interest. Complexity can arise, sometimes quite unexpectedly, in short programs and modularity is something worth getting used to as soon as one can. It would be unwise to wait until the need becomes urgent; the necessary skills should be developed from an early stage so that they are available when needed. We shall examine some programs which are just long enough and just complex enough to indicate the need for, and introduce the techniques of, modular programming.

Example 5.2
Write a program to draw on the screen the picture in Fig. 5.2 positioning it roughly in the middle of the screen.

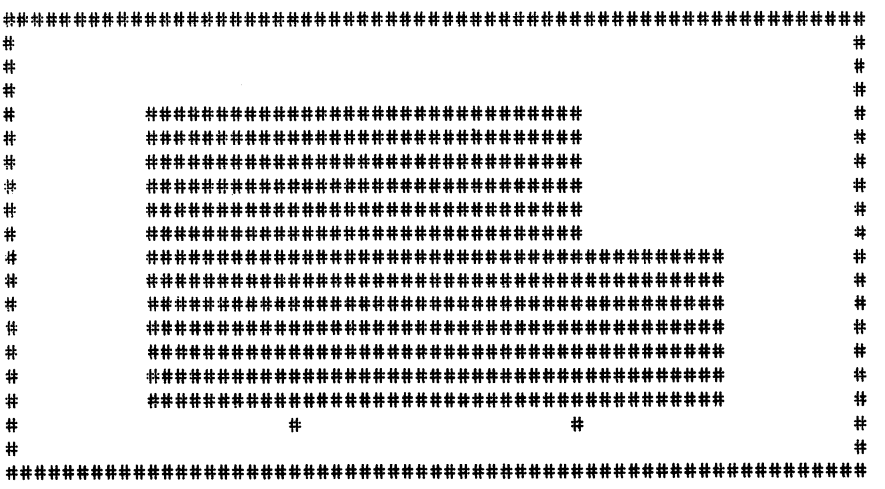


Fig. 5.2 – Picture using simple screen graphics.

METHOD
(1) The picture consists of twenty lines of characters and we could simply construct the appropriate set of twenty print statements. We could reduce the labour by noticing three sets of repeated lines and using FOR loops.

There is no point in doing it that way and the problem might have been phrased to exclude such an approach. A good programmer will often seek a method which might be useful on other occasions or might lead easily to some such program.

In this case the obviously useful approach is to write procedures which will place rectangles, lines and single characters in any specified position. We can observe that a line consists of single characters and a rectangle consists of lines and so build up a hierarchy of procedures according to the scheme in Fig. 5.3. Note that a line may be a row or a column. The arrows indicate which modules contain procedure calls (EXEC statements) to which other modules.

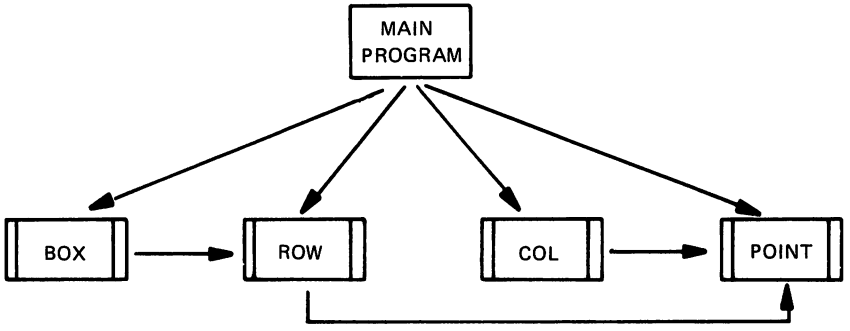


Fig. 5.3 – Modules of picture drawing program.

PROBLEM ANALYSIS/PROGRAM DESIGN

(1) It is necessary to specify the information which must be passed to each module.
In the case of procedures this can be done easily and precisely by stating the procedure name and parameters.

(2) All procedures will deal with a point (ac,dn) which may indicate the position of an element (top, left end or top left corner) and the other meanings will be obvious.

(3) The procedures will be:

```
point(ac,dn)
row(ac,dn,width)
col(ac,dn,hite)
box(ac,dn,width,hite)
```

(4) The main program will consist of up to four FOR loops. Each one will deal with all the elements of a particular type.

(5) For each main FOR loop there will be a DATA statement (or sequence of DATA statements) with the format:

Number of elements, set of data, set of data . . .

Each set of data is two, three or four numbers which define a point, row, col or box.

(6) For simplicity the asterisk will be used in all elements.

(7) It is easy to analyse the picture in Fig. 5.1 and construct the following DATA.

Rectangles

```
DATA 2, 20, 7, 31, 13, 51, 13, 10, 7
```

Rows

```
DATA 2, 10, 3, 61, 10, 22, 61
```

Columns

```
DATA 2, 10, 3, 19, 70, 3, 19
```

Points

```
DATA 2, 30, 20, 50, 20
```

PROGRAM

The first part of the program matches the data above and is specific to the drawing of a car.

Rectangles

```
READ num
FOR item := 1 TO num DO
   READ ac, dn, width, hite
   EXEC box(ac, dn, width, hite)
NEXT item
```

Rows

```
READ num
FOR item := 1 TO num DO
   READ ac, dn, width
   EXEC row(ac, dn, width)
NEXT item
```

Columns

```
READ num
FOR item := 1 TO num DO
   READ ac, dn, hite
   EXEC col(ac, dn, hite)
NEXT item
```

Points

```
READ num
FOR item := 1 TO num DO
   READ ac, dn
   EXEC point(ac, dn)
NEXT item
```

The second part of the program is a set of procedures called from within the FOR loops.

```
PROC box(ac, dn, width, hite)
   FOR line := dn TO dn + hite – 1 DO
      EXEC row(ac, line, width)
   NEXT line
ENDPROC
PROC row(ac, dn, width)
   FOR a := ac TO ac+width–1 DO
      EXEC point(a, dn)
   NEXT a
ENDPROC
PROC col(ac, dn, hite)
   FOR d := dn TO dn + hite – 1 DO
      EXEC point(ac, d)
   NEXT d
ENDPROC
PROC point(ac, dn)
   EXEC cursor(ac, dn)
   PRINT "*"
ENDPROC
```

Note

The procedure call EXEC *cursor* will call a procedure which positions the cursor. This will vary with different systems.

PROGRAM
The program consists of the following elements.

DIM statement and initialisation for procedure *cursor.*
Four FOR loops as above.
Procedures *box, row, col, point, cursor*
Four DATA statements as above.

The program given below has the procedure *cursor* and initialisation which is specific to PET COMAL 80.

```
DIM down$ of 24, right$ OF 80
FOR k := 1 TO 24 DO down$(k) := CHR(17)
FOR k := 1 TO 80 DO right$(k) := CHR(29)
PRINT CHR(147)
(FOR loops)
(procedures box, row, col, point)
PROC cursor(ac, dn)
   PRINT CHR(19)
   PRINT down$(1 : dn-1), right$(1 : ac-1),
ENDPROC
(DATA statements)
```

The problem broke down quite naturally into four procedures and four FOR loops. But notice that the pattern of data was established before the final writing of the program. The structure of the data is a simple linear pattern but it is systematic, conforming to a carefully established format. This program illustrates the typical sequence of:

INPUT DATA – PROCESSING – OUTPUT DATA

We shall see in later work that it can be a helpful approach to the solution of problems. Important questions are:

What data do I have to start with?
Exactly what output do I need?
What is the easiest way of transforming the available data,
or part of it, into the required output?

Sometimes the questions may be asked in a different order:

What output do I need?
What is the best combination of input data and processing to achieve this result?

Such questions may seem obvious but a non-trivial amount of computer time and programming effort is wasted because they are sometimes not asked or not answered properly.

5.4 DEVELOPMENT OF PROGRAMS

The best way to achieve useful objectives in computing, as in other areas of work, is to try define, as precisely as possible, what is required or desirable as early as one can, and certainly before any programs are written. A change in specified objectives can alter program designs in ways which might be trivial or so fundamental that substantial re-designing is necessary. It should therefore be an aim of a computer programmer or systems analyst that programs should not need much development.

However, in the real world of inadequately specified projects or changing circumstances beyond the control of the programmer, sometimes it is necessary to amend a program. If it is well-structured this should not be too difficult. Suppose for example it is decided that some pictures drawn on the screen should be copied to a printer. Assuming that it is a normal printer which outputs a line at a time it is immediately obvious that one cannot print with the same freedom as one uses the cursor on a screen.

METHOD

(1) An array must be set up to match the screen (24 lines of 80) and codes must be placed into the array.
(2) The question arises whether to use a string array (24 strings of 80 characters) or to use a two dimensional numeric array – 24 rows of 80 numeric elements. Such a decision must be made for a particular system taking into account the speed with which the system may handle strings as opposed to individual characters and also what restrictions the choice of strings might place on permissible characters.
(3) For simplicity we will use a numeric array (two dimensional).
(4) An examination of the program will show that only the procedure, point, actually sends characters to the screen. This means that we can easily transfer a code for every character into the array.
(5) When the picture is built up on the screen it will also be in the array as a set of character codes for asterisk.
(6) The array can then be taken a line at a time and printed.

PROBLEM ANALYSIS/PROGRAM DESIGN

(1) DIM screen(24,80) will declare an array. This will immediately be filled with space codes, 32.
(2) In procedure, *point*, after a character has been printed the values of *ac, dn,* are exactly what the array, *screen,* requires:

```
screen(dn,ac) := 42 (ASCII code for*)
```

Note

Note that an element is thought of as (row,column). This is the reverse of the coordinate system (across, down) for the cursor but it should cause no great difficulty.

(3) In order to preserve good structure and perhaps provide a choice as to whether printing is required, the screen drawing and printing routines will be turned into procedures and called from a new very short main program which is essentially:

```
EXEC spaces
EXEC drawpic
EXEC printit
```

(4) A statement of the form

OUTPUT PRINTER

or SELECT OUTPUT PRINTER

will be required in the procedure *printit*.

The construction of the program for a particular implementation is left as an exercise for the reader.

PROBLEMS

5.5 Design five pictures such as *cross, square, window, face, car* on a scale which will enable three of them to fit across the screen as in Fig. 5.4

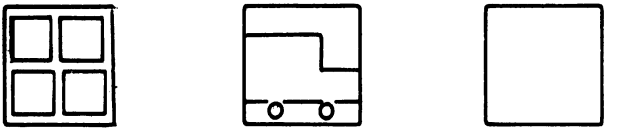


Fig. 5.4 – Screen pictures.

5.6 Write procedures cross, square, etc. each with a single parameter which determines whether the display occurs in position 1, 2 or 3.

5.7 Design and write a program which simulates a "One-armed bandit" or "Fruit machine". The user should simply press a key and this should cause the display of three random pictures. Do not necessarily worry about stake or prize money.

Note

This problem is as much about data analysis and program design as aboutcoding. Coding is important but as problems become more realistic, that is, longer and more complex, the other aspects of programming become more significant. The problem is not graded in the usual way.

CHAPTER 6

Structure Diagrams

I think that I shall never see a poem lovely as a tree.

Trees Rasbach/Kilmer

6.1 GRAPHS OF ALGORITHMS

In the early days of computing, people discovered or re-discovered how helpful a picture can be in conveying information. Like the doctor with his anatomical diagram or the engineer with his blueprint, the programmer found flowcharts useful. The literature abounds with them though their use seems to be in decline.

From the point of view of a computer manager or programming team leader it is better for a programmer to work out his program – typically hundreds or thousands of lines – in considerable detail, and have it checked, before the machine processes begin: program entry, testing, editing. In this way it is hoped that only a modest number of "runs" would be required to get a program working and ready for systematic testing. Before the program gets into the machine it is not easy to manipulate. A flowchart is easier to deal with at this stage and for a second person to use in checking the program.

On the other hand a programmer might prefer to have his program entered at an early stage because, once in the system, it is much easier to manipulate – extend or alter – than it is in the form of lines on coding sheets or flowcharts. But this method uses more computer time and can cause bottlenecks at computer keyboards. Sometimes the organisation wins, sometimes the programmer wins and draws the flowchart *after* the program is tested and working. Even then further amendments to the program might not be incorporated into the flowchart. Thus flowcharting is a mixed blessing however it is handled.

But the major reason for the decline of the flowchart is the decline of the GOTO statement – the two things go together. As the emphasis in problem analysis and programming moves away from flow of control in the computer towards structured methods the flowchart simply becomes largely irrelevant as the GOTO statement becomes much more rarely used. The realisation that most problems can be solved in terms of about half a dozen fundamental algorithmic structures related in only three different ways has illuminated the 1970s like a rising sun – truly the light of experience. But as languages become more

humanised and problem-oriented is it necessary to stop believing that a picture is worth a thousand words?

The answer is that the simplicity and near-completeness of the new structures provides a much more comprehensible framework for the examination of ideas, but there is still a place for visual aids. Before examining one such system in detail a brief glance at four types of graphs of algorithms is interesting. The example is only illustrative and certainly not a sound basis for judging relative merits.

The example in Fig. 6.1 has been used as an introduction to flowchart concepts and records what most people do most nights.

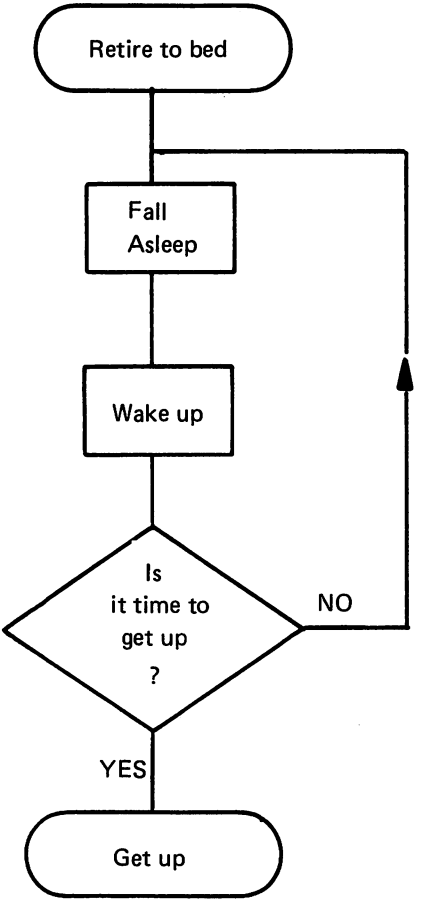


Fig. 6.1 – Flowchart.

The flowchart, as the name suggests, is a representation of program elements connected by lines representing flow of control. The repetitive process can be

inferred from the way control flows. The structure diagram attempts to convey repetition as an entity and lines show the subordinate relationship, not flow of control. Fig. 6.2 shows the same example. A structure diagram usually has a title which is the name of the program, perhaps the same name as is used for the storage device such as a disc.

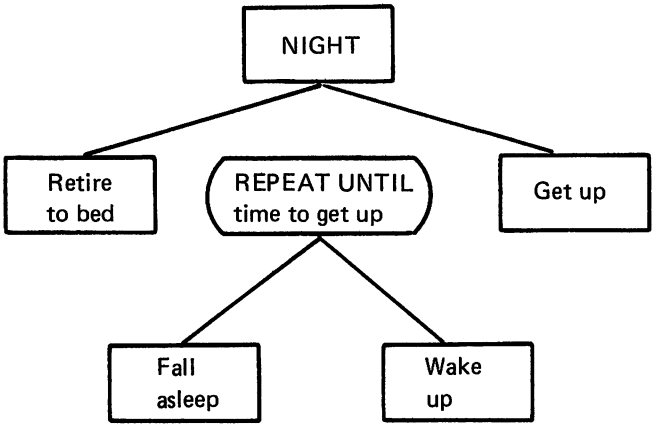


Fig. 6.2 – Structure Diagram.

A third method of representing the same processes is the iterative graph which again exhibits relationships.

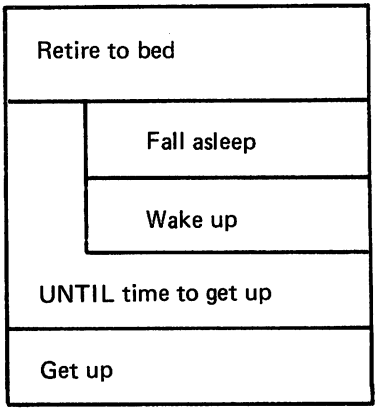


Fig. 6.3 – Iterative Graph.

Finally the design structure diagram in Fig. 6.4 allows for the testing of the exit condition anywhere in the loop.

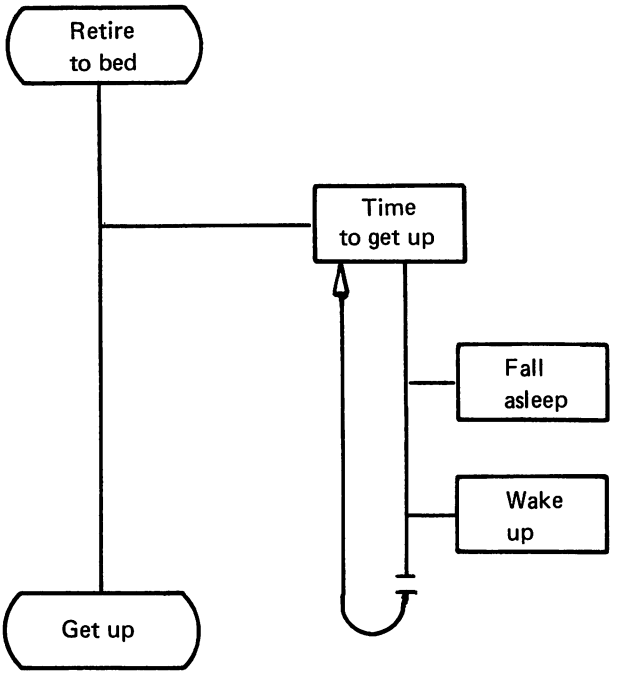


Fig. 6.4 – Design Structure Diagram.

It is fairly easy to reject the flowchart as irrelevant to modern methods and the writer finds iterative graphs awkward to manipulate at the construction stage. Of the remaining two both are easy to manipulate. Both reflect the solution of a problem and therefore the function of the program. The design structure diagram matches the program and its indentation very clearly. In a structure diagram the levels of the tree match the levels of indenting in the program and one is obtainable from the other by following simple rules. On the other hand the structure diagram seems to reflect more closely the hierarchial structure of the analysis of a problem. This relationship is even closer if top-down methods of analysis have been used in achieving it.

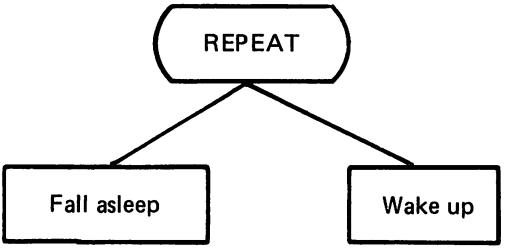
The decision is subjective. The elegant simplicity of trees, real or abstract, appeals greatly and the writer follows Ken Bowles, who developed the UCSD Pascal system, and Borge Christensen in preferring them. But it must be said that this book could be based equally correctly on the use of design structure diagrams.

6.2 STRUCTURE DIAGRAMS

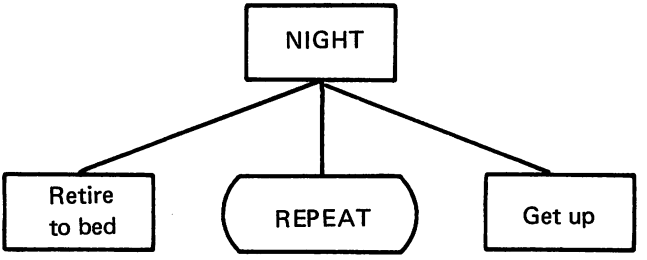
6.2.1 Sequence

The sequential position of statements is represented by following the boxes from left to right along the same level.

However these two instructions are subordinate to a repeat structure and the subordinate relationship is shown by lines connecting the adjacent levels of the tree



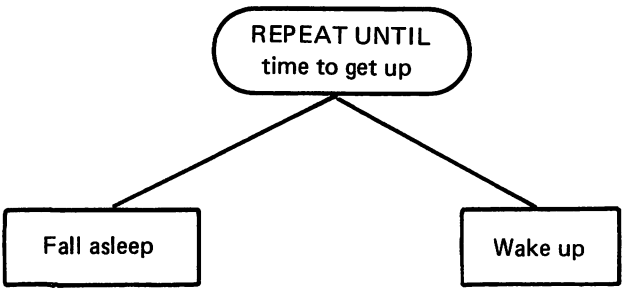
By making the title of the program the root of the tree we establish that all statements (except the title) are subject to something and there is a uniformity of treatment between programs and procedures. If we regard the loop as a single entity its place in the overall sequence is clearly displayed.



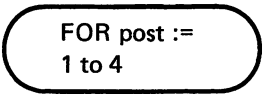
In order to understand the working of the repeat loop in detail we must slip down to the next level and apply exactly the same rules when we get there.

6.2.2 Repetition

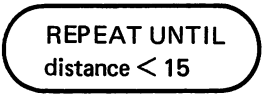
As we have seen, repetition is indicated by a box with curved ends which may possibly be intended to symbolise the cyclic nature of the process.



The types of repetition already discussed should be indicated with a minimum of working. Space is a problem with structure diagrams as it is with flowcharts and we need to be as concise as possible with sacrificing clarity. In simple or introductory examples we might use the complete forms:

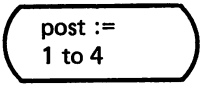


FOR loop

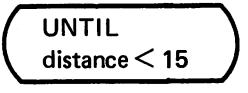


REPEAT loop

However, there is a saving of space without loss of clarity in the shorter forms:



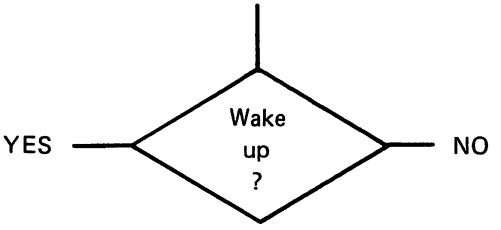
FOR loop



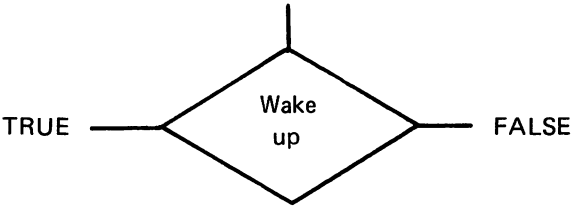
REPEAT loop

6.2.3 Binary Decisions

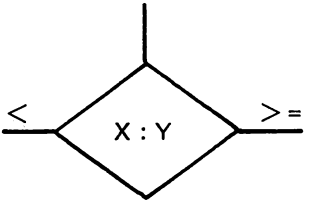
The words choice and selection are used in the literature in the same sense as decision. In particular the phrase, Decision box, is common for a diamond shaped flowchart symbol. The contents of the box may be a question:



or a statement:

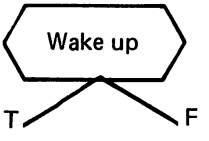


or, sometimes in mathematical work, a choice of relations is implied:



which means take the left hand branch if X is less than Y.

In a structure diagram the decision box is cut down to save space and we place only *statements* inside one which represents a binary decision. T or F will stand for TRUE or FALSE.



It is quite usual for more than one statement to occur within either or both sections of an IF/THEN/ELSE structure. This can be handled by interpolating a small box which serves only to show that the statements are grouped together as in Fig. 6.5.

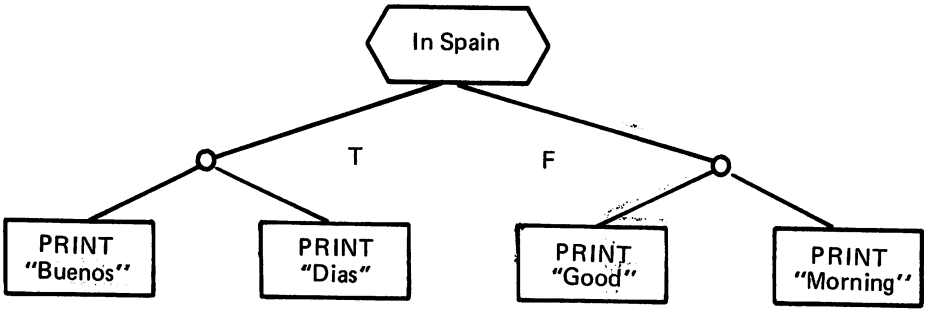


Fig. 6.5 – Grouping of statements.

One or other of the T and F branches may be without any boxes. Usually, if one branch is empty, it would be the F branch which corresponds to the ELSE section but a programmer who wishes to phrase his condition in such a way that the T branch is empty is allowed to do so.

6.2.4 Multiple Decisions

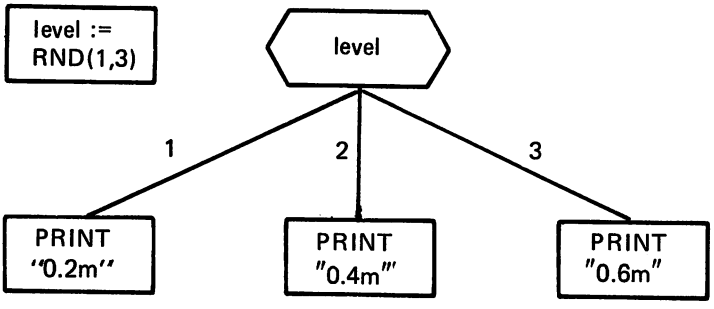
It is easy to extend the use of a binary decision box to accommodate a multiple decision but there are three differences:

(1) The statement in the decision box is replaced by an expression which must evaluate into a number or string.

(2) The letters T and F must be replaced by the possible values of the expression which must be associated with particular lines.

(3) There can be any number of lines.

A segment of the post holes problem will exemplify this (see section 4.4).

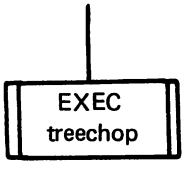


An extra line could be inserted to deal with the case where *level* evaluates to some number other than 1, 2 or 3 perhaps because of an error in the preceding assignment statement. This extra line should be labelled OTHERWISE or OTH to correspond with COMAL syntax:

```
—
—
—
—
WHEN 3
  PRINT "0.6m"
OTHERWISE
  PRINT "error"
ENDCASE
```

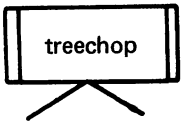
6.2.5 Procedures

The double-ended rectangle as a terminal node indicates a procedure call. Some people use a rectangle inside another but the double-ended rectangle is much easier to draw.



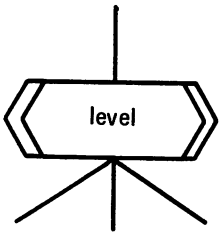
Procedure Call

The corresponding procedure definition will have a similar box to the root:

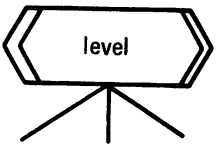


Procedure Definition

A special type of procedure called a function operates in such a way that the name of the procedure, which is also a variable, takes a particular value as a result of the function being executed. If this value is used as the basis for a decision it seems natural to indicate it with a double lozenge shape. For the sake of completeness we give the symbol here but the concept will be treated later (section 8.4).



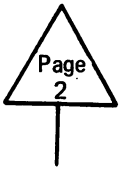
Function Call



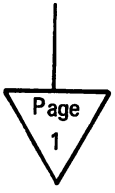
Function Definition

6.2.6 Connectors

If there is not enough space on a page to continue a structure diagram this can be done on another page. The continuity being re-established by connectors.



Start of Page 2



End of Page 1

6.2.7 Creating Space

Sometimes one cannot fit enough boxes on the same level. The lines show exactly what the relationships are and it does no harm to place some of the boxes below others provided their correct status is clear (Fig. 6.6).

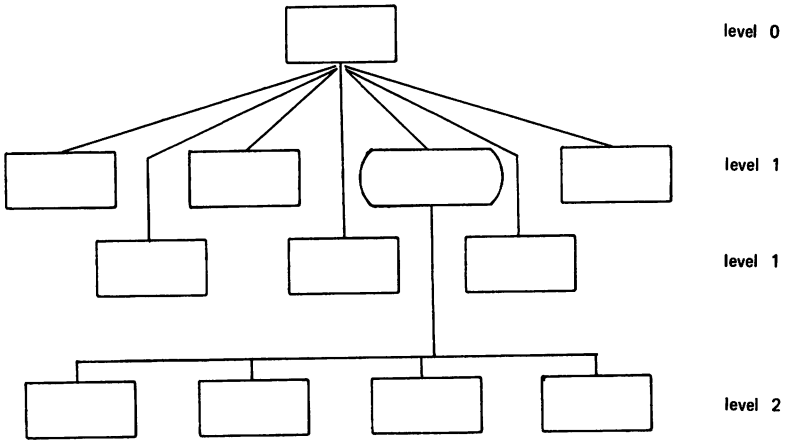


Fig. 6.6 – Creating space.

6.3 PROGRAMS FROM STRUCTURE DIAGRAMS

A set of four rules was given in section 3.5 for writing a program from a simple structure diagram. We are able to refine these rules further to give more precise instructions for a larger variety of diagrams.

There are essentially three types of boxes or nodes. The rectangular boxes represent action or the title of a program or procedure. A procedure call or a procedure definition requires a double-ended rectangular box. A rectangular box is one of three things:

(1) A title.
(2) A terminal node or leaf representing action.
(3) An intermediate stage in the analysis of a problem.

The other two types of boxes are curved ones for repetition or lozenge shaped for decision. If the decision is based on evaluating a function the lozenge shape has double ends for the function call and for its definition title.

6.3.1 Natural Walk Rules

(1) Start at the root and "walk" so that your left hand always touches a branch or a box.
(2) When you come to a rectangular box write down the corresponding program statement.
(3) When you come to a curved box (a repetition node) on the first encounter, going down the tree, open the structure. On the final encounter, going up the tree, close the structure.
(4) When you come to a lozenge shaped box representing a binary decision:
    (a) On the first encounter write the IF . . . THEN part.
    (b) On the second encounter write ELSE.
    (c) On the final encounter, write ENDIF.

The exceptions to this are the *one liner* which requires neither ELSE nor ENDIF and where the F line requires no action in which case ELSE is not required.

(5) When you come to a lozenge shaped box representing a multiple decision:
   (a) On the first encounter write the CASE . . . OF part and the first WHEN part.
   (b) On intermediate encounters write the other WHEN parts and the OTHERWISE part if necessary.
   (c) On the final encounter write ENDCASE.

Although these rules seem elaborate at first glance, what is required essentially is a natural walk around the tree and at each node the particular programming requirement is usually obvious. It is suggested that the reader applies the rules to the examples in the next section.

6.4 EXAMPLES OF STRUCTURE DIAGRAMS

The example in Fig. 6.7 is the structure diagram for the program given in Fig. 2.2.

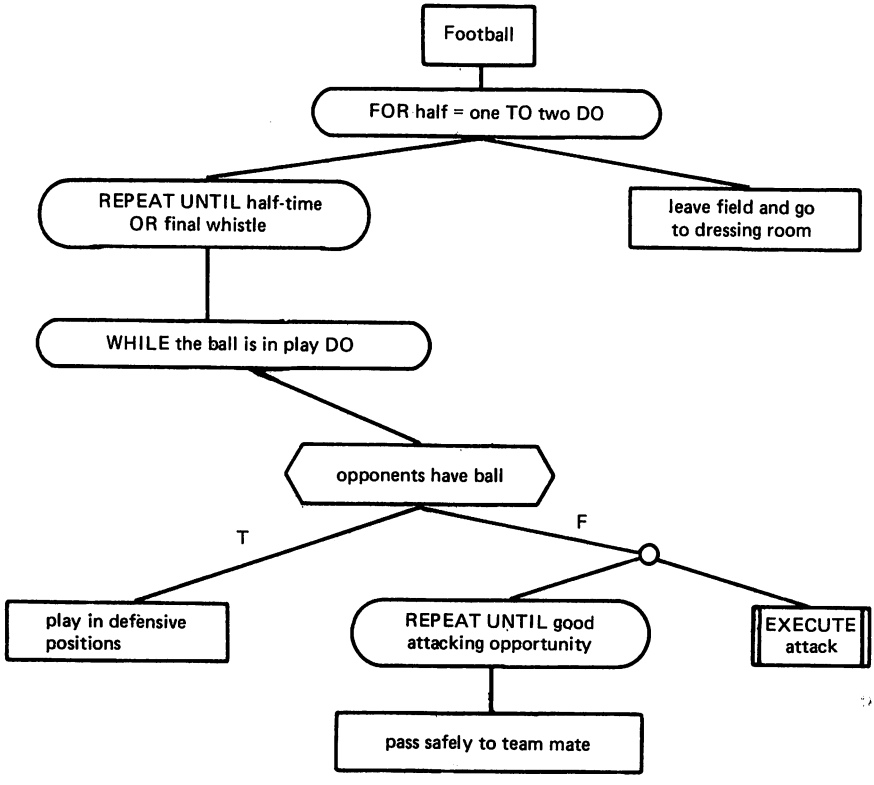


Fig. 6.7 – Football manager's instructions.

The procedure *attack* is given in Fig. 6.8.

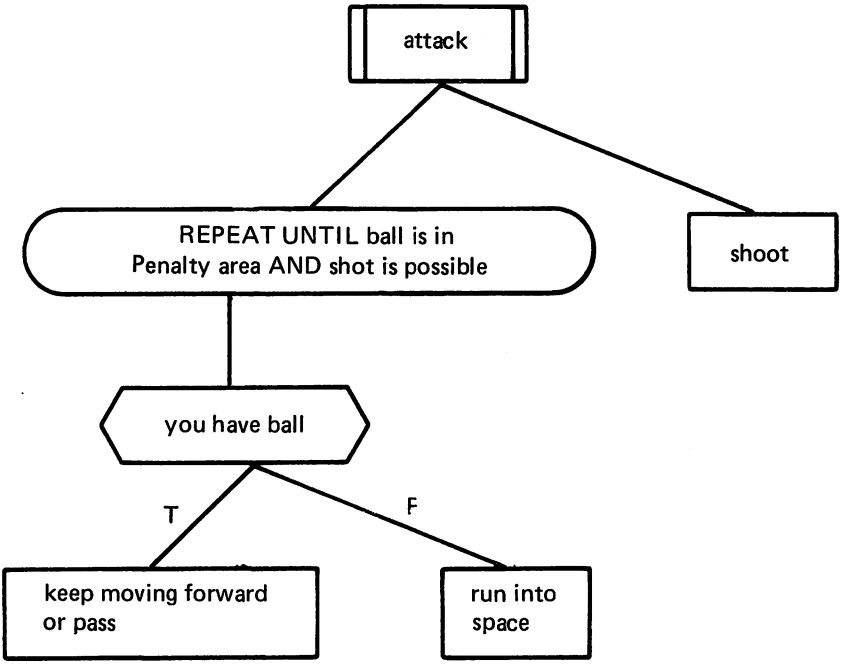


Fig. 6.8 – Procedure attack.

The post holes problem is outlined in Fig. 6.9 and the related procedure is given in Fig. 6.10.

PROBLEMS

Refer to Problems 4.4 through to 4.21.

6.1 By re-examining and analysing each problem, draw a structure diagram from which a program might be written.

6.2 Check the correctness of your analyses by writing and testing programs derived from the diagrams or by checking against the given solutions to the problems.

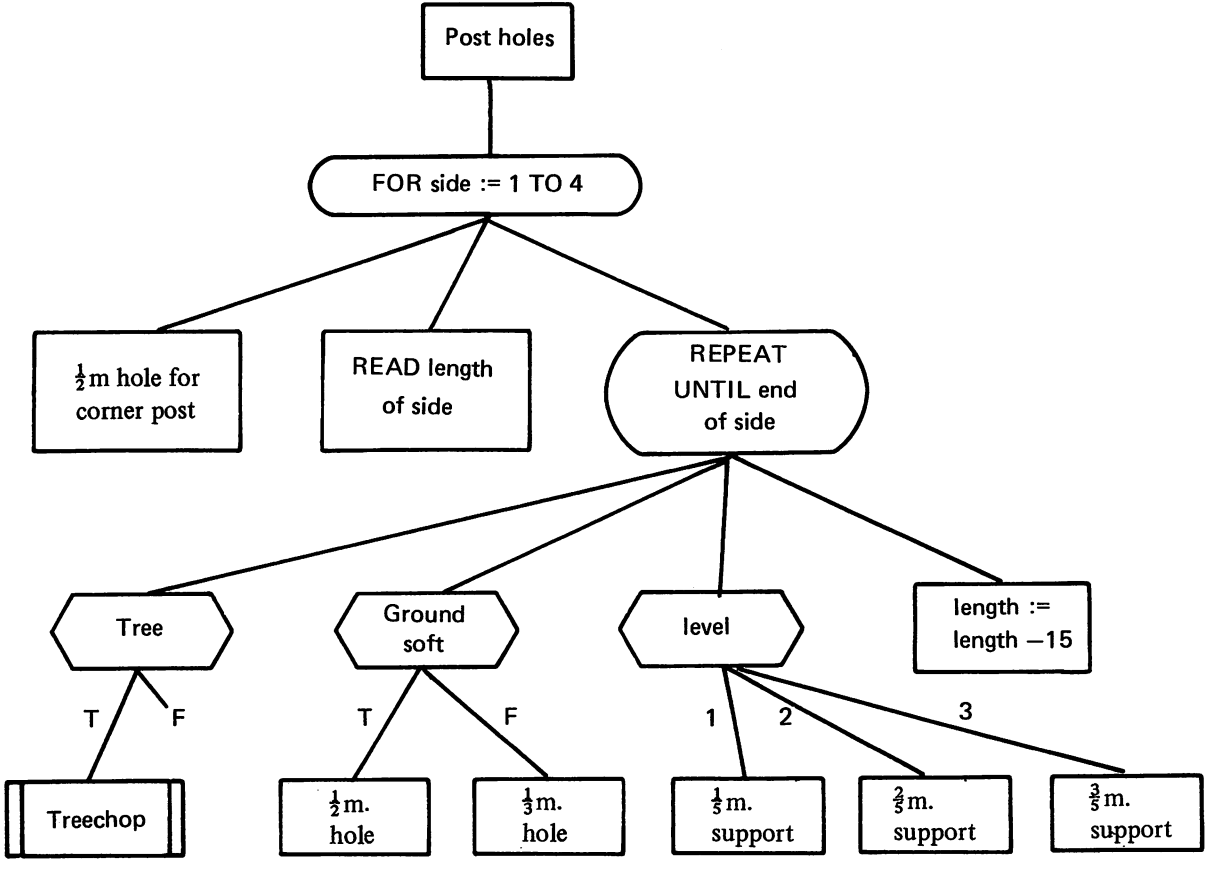


Fig. 6.9 – Post holes problem.

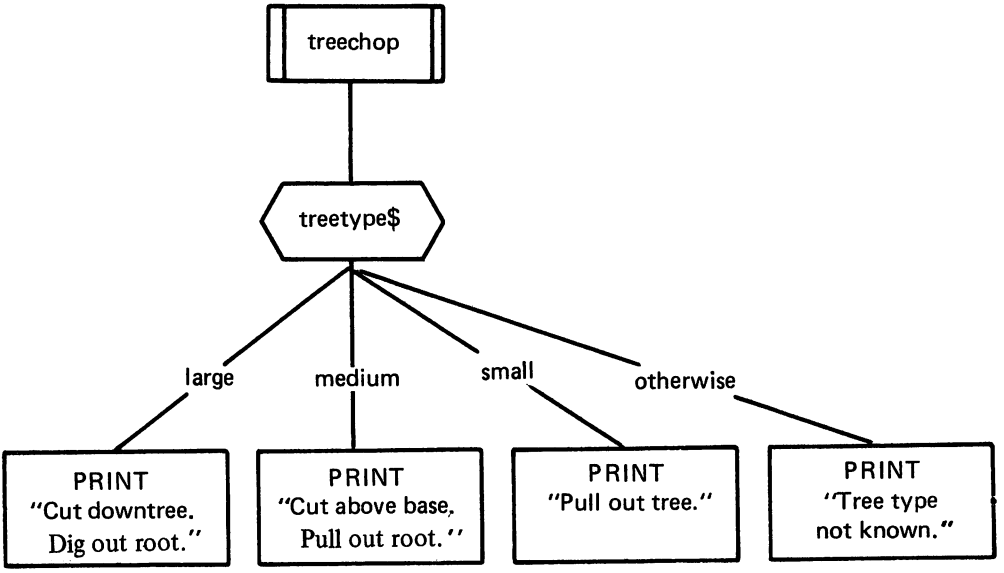


Fig. 6.10 – Procedure treechop.

CHAPTER 7

Top Down Analysis

> The hierarchial listing displays graphically the broader term/narrower term relationship . . . To broaden a search move up the hierarchy; to narrow it move down.
>
> *Thesaurus of Computing Terms*, National Computing Centre (UK)

7.1 THE TOP DOWN METHOD

When men first thought of making a motor cycle or motorised bicycle the first inspirational flash of thought probably had only two elements – bicycle and motor. A second level of analysis might have broken down the word bicycle into frame, wheels and chain, the pedals being omitted for obvious reasons. The word motor might have been broken down into engine, controls, gearbox as in Fig. 7.1. Other structured groupings of parts are possible and certainly other words might have been used but a simple linear list of all the necessary parts could not easily be achieved and would not be much use without special groupings and relationships being made explicit.

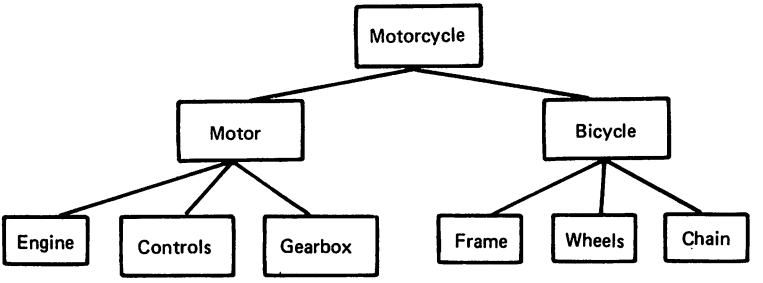


Fig. 7.1 – First stage of motorcycle design.

At this stage it might be realised that there is more sensible structuring which combines chain with gearbox, adds a clutch and calls the new element, transmission, as in Fig. 7.2.

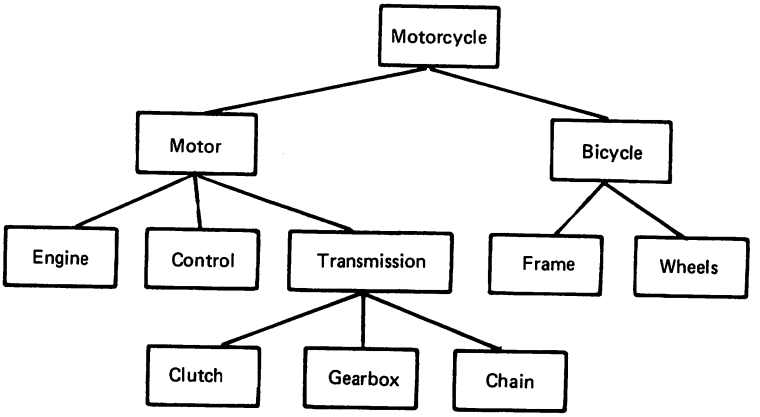


Fig. 7.2 – Second stage of motorcycle design.

It is important to note what has happened here. An examination of a certain level of detail has altered the original analysis. Such 'bottom-up' or 'sideways' effects often occur with very useful consequences in what is intended to be essentially a top down process.

In *Programming and Problem-solving in Algol 68* (Macmillan, 1978) Andrew Colin quotes from Robert Pirsig's *Zen and the Art of Motorcycle Maintenance.*

> And you see that every time I made a further division up came more boxes based on these divisions until I had a huge pyramid of boxes. Finally you see that while I was splitting the cycle into finer and finer pieces, I was also building a structure.

The motor cycle dismantler was actually re-discovering the structure which was implicit in the design and assembly of the motorcycle. One cannot say with certainty how the original design was achieved but the result is undoubtedly a hierarchial structure.

A programmer is like the designer of a motorcycle. He has a global concept of what is required and he may have some knowledge of lower levels of detail. When he de-bugs the program he is like a mechanic who must diagnose and correct a fault. In both cases an appreciation of the structure is necessary and in both cases a hierarchial structure is likely to be easier to follow. Such a structure can be achieved by methods which are essentially bottom-up. For example one could start with an idea for a new type of carburettor and design that first, add an existing engine and wheels, then design the frame around these items.

Some writers have called the methods, in which one starts with the global concept, 'stepwise refinement' and others have used the phrase, 'Top down analysis with stepwise refinement'. An accurate and informative phrase to describe what happens might be rather longer but, given that a title can only give a simplified notion of what is referred to, the latter has considerable merit.

If a program needed to be written and it was known that some already existing elements – procedures or programs or program segments – could be usefully incorporated then a 'bottom-up' process of synthesis might be employed to design the program. Starting with these elements, and perhaps writing some more, a process of linking and relating could eventually produce the required program. This type of approach to problem-solving or job organisation for a computer is essentially a synthesis of elements to produce a desired result.

On the other hand one could start by defining the job in simple global terms and break it down into subjobs. These subjobs could be encoded if they were simple, alternatively they might be broken down further. At each stage there is a 'refinement' of relative complexity into relative simplicity. Eventually something emerges which is simple enough to encode safely and efficiently. This 'Top-down' approach is essentially analysis rather than synthesis.

A top-down analysis does not preclude the use of existing elements but it does encourage the natural emergence of program elements which are suitable and properly related to each other and to the original definition of the job.

Structured programming ensures that only certain agreed program elements will be used. Thus it is easy to recognise when the refinement of a job has proceeded far enough to be recognised as such an element. A hierarchial structure emerges naturally from a top-down analysis and a structured program is also a hierarchy. Thus the two go well together.

In practice a job may be started in the middle or it may be a mixture of top-down and bottom-up processes. But the performance of a programmer who strives to analyse problems in a methodical top-down manner is likely to improve. His problem solving skills should develop and his presentations of analyses should get better.

7.2 A WORKED EXAMPLE

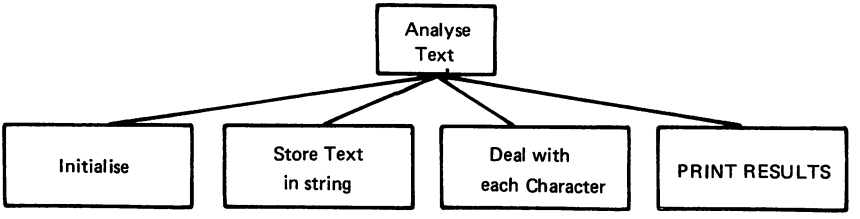
7.2.1 Problem

A piece of text is to be analysed in order to obtain a count of the number of vowels, consonants, digits and other characters. For simplicity assume all letters are upper case and the line will fit into a simple string. Spaces should be ignored.

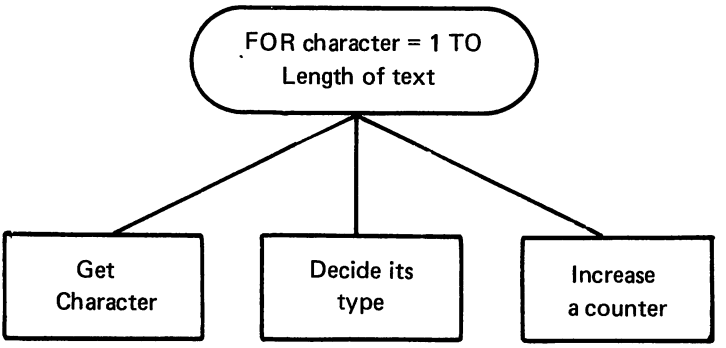
7.2.2 Analysis

Thus far we have split the job of analysing text into four subjobs.

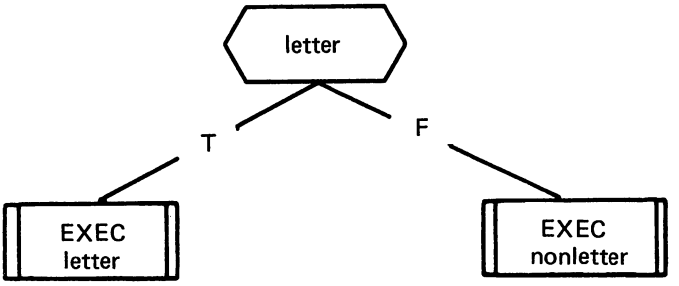
We now proceed to a further analysis of any jobs not yet simple enough to encode. The only one which requires further analysis is "Deal with each



character". We now refine this further and because we know that it can be handled within a FOR loop the analysis proceeds as follows:



The only job which can be further refined is "decide its type". From our knowledge of the syntax of COMAL and some knowledge of character codes, we know that we can easily distinguish letters from non-letters. For the sake of illustration and perhaps adding marginally to program clarity we will use two procedures to carry out further analysis.



7.2.3 PROCEDURE letter

We can easily identify a vowel so, given that the character is a letter, if it is not a vowel it must be a consonent. See Fig. 7.3.

7.2.4 PROCEDURE nonletter

It is easy to test for a space and for a digit so our analysis of a nonletter tests for these in turn. However, we must ignore spaces and it is generally better to word such a decision so that the action results from the TRUE branch. Again this bit of the analysis is dependent on a knowledge of a language syntax. See Fig. 7.4.

We now have a complete analysis of the problem which has started from the overall requirement and broken down the job in subjobs, continuously refining

where necessary until no further analysis is necessary. The notation of structure diagrams has been used but this is a matter of personal preference. At this stage one could proceed straight to the coding. However, though the procedures are satisfactory the rest of the analysis could be tidied up a little and one item has changed in the process of analysis. It has become clear that the simplest thing to

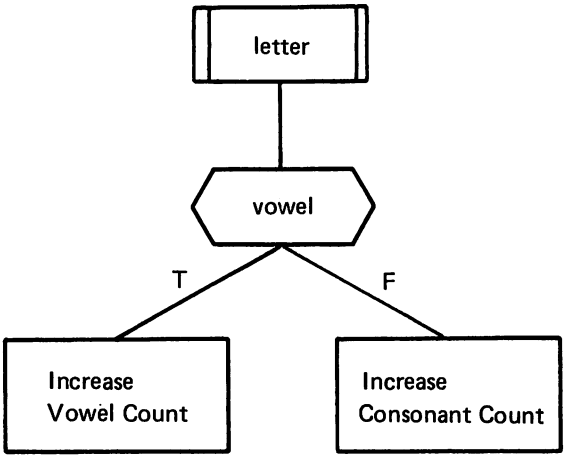


Fig. 7.3 – Procedure *letter*.

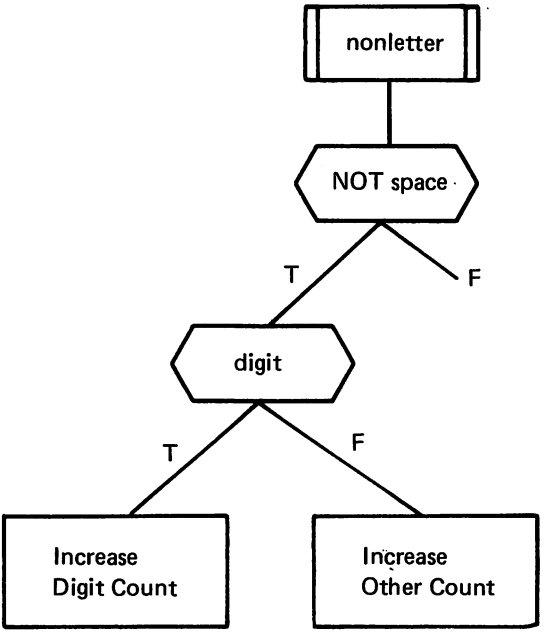


Fig. 7.4 – Procedure *nonletter*.

do is increase a counter as soon as a character has been properly identified. This occurs at four separate parts of the program and therefore the box which says "Increase a counter" in the second diagram becomes redundant. Such feedback of information often occurs because it is not always possible to get a complete analysis right at every stage.

Although we started with the intention of doing a top-down analysis even in this simple program there was a back-tracking process. This bottom-up movement is often necessary because few programmers have such complete analytical ability that they can produce a correct analysis in one top-down sweep. Experience indicates that a judicious mixture of a bird's eye general view needs to be informed by what is more like a worm's closer view of specific details.

We are now in a position to blend the separate parts, except the procedures, to make one diagram as in Fig. 7.5. The reader will note that the wording is not COMAL but it does get close to it in parts. This, again, is largely a matter of personal preference. It is quite impossible to analyse a problem in this way without knowing the necessary concepts, and it is unlikely that one could acquire these concepts without learning some COMAL or COMAL-like computer language syntax. Structure diagrams can be written in formal COMAL syntax but the ideas are more important than the notation at this stage.

This diagram together with the two procedure diagrams constitutes the complete analysis. We can make a list of identifiers and then either write a detailed structure diagram using program syntax or we could proceed straight to

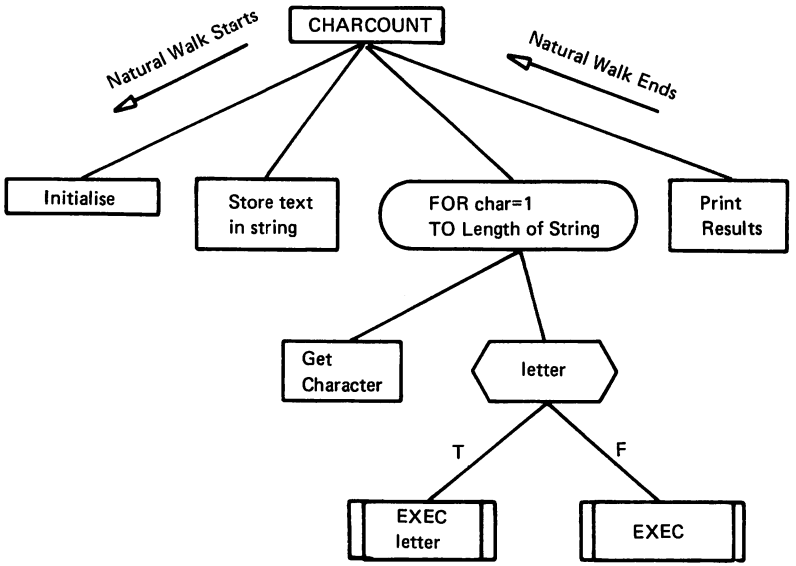


Fig. 7.5 – Final Structure Diagram.

a program. In a more complex situation with a less easy operating environment this should certainly be done but it seems unrealistic to draw more diagrams for this fairly simple problem.

We now do a "natural walk" around the main program tree to get our program.

```
PROGRAM CHARCOUNT
        DIM text$ OF 80, char$ OF 1
        row := 0; cons := 0; dig := 0; oth := 0
        READ text$
        FOR ch := 1 TO LEN(text$)
           char$ := text$(ch:1)
           IF char$ => "A" AND char$ <= "Z" THEN
              EXEC letter
           ELSE
              EXEC nonletter
           ENDIF
        NEXT ch
        PRINT vow; "vowels"; cons; "consonants"
        PRINT dig; "digits"; oth; "others"
        PROC letter
           IF char$ IN "AEIOU" THEN
              vow := vow + 1
           ELSE
              cons := cons + 1
           ENDIF
        ENDPROC
        PROC nonletter
           IF char$ <> " " THEN
              IF char$ IN "0123456789" THEN
                 dig := dig + 1
              ELSE
                 oth := oth + 1
              ENDIF
           ENDIF
        ENDPROC
        DATA "CRY 'HAVOC,' AND LET SLIP THE DOGS OF WAR.-
                                        -JULIUS CAESAR 3.2"
```

7.3 PROGRAM CORRECTNESS

Programmers have always tried to write programs that work properly but the methods described so far can justify a much stronger confidence that they will

work. Programs must still be tested but the emphasis should be on correctness achieved by sound methods of analysis in terms of well-proven structures rather than by getting rid of bugs. Prevention is better than cure.

Definition A program is correct if it is:
(1) Error free;
and (2) Does exactly what was intended.

Test data can increase confidence but:

> Program testing can be used to show the presence of bugs but never to show (prove) their absence.
>
> (Dijkstra, 1968)

Programs can rarely be tested with every possible set of data.

> However, people do write correct programs. In fact many who follow a structured programming methodology find that over half their programs work first time and continue to work through extensive testing. Such programs are typically only 50 to 400 lines of code. But they are often part of much larger programs.
>
> (McGowan and Kelly, 1975)

> There is no foolproof way to ever know that you have found the last error in a program. So the best way to acquire the confidence that a program has no errors is never to find the first one, no matter how much it is tested and used. It is an old myth that programming must be an error-prone, cut-and-try process of frustration and anxiety. But there is a new reality that you can learn to consistently write programs which are error free in their debugging and subsequent use. This new reality is founded in the ideas of structured programming and program correctness, which not only provide a systematic approach to programming but also motivate a high degree of concentration and precision in the coding subprocess.
>
> (Harlan Mills, 1973)

The most eloquent statement known to the writer of the general attitude of a good programmer is by Laurie Moore of Birkbeck College, London:

> It is well-known that the way to deal with biological bugs is to have a good bath, and to continue the treatment with regular and ample doses of soap and hot water. Furthermore, it is held that if you keep clean in the first place, you will not be attacked by bugs anyway. Similar considerations have been applied to programs. Bugs love to inhabit programs which are large,

> full of narrow twisting passages along which program control flows in a bewildering maze, even many passages through which control never flows at all, thus allowing dirt to accumulate and bugs to breed, and they love the dark. Simple sparkling programs, well lit by the disciplined use of uncomplicated structures and illuminating comment seem to keep them away, at least except for the stray intruder who is easily spotted and quickly dealt with.
>
> Hence the importance of step-by-step development of a program in small well-defined modules which are simple, clean even to the point of elegance, and easily tested. Hence the importance of fitting these modules together step-by-step, adding one at a time, and keeping the bugs out. Keeping the bugs out of programs is done by disciplined attention to routine checking of detail at the preparation stage, not by the exercise of fiendish skill in outwitting the bugs.
>
> *Foundations of Programming with Pascal*
> (Ellis Horwood, 1980)

Like an engineer or a carpenter or a doctor or any professional or skilled person, a programmer must be prepared for whatever is necessary to guarantee satisfactory results. He should use the best methods he can and be as careful and as painstaking as the job requires. Finally if his computer system is unfriendly or if he is required to use an unhelpful computer language he should make his views known and use whatever influence he may have to get improvements.

7.4 TESTING AND DEBUGGING

Despite the carefully chosen words of such expert and experienced people many programs do have bugs in them and people, particularly people who have just started to learn and develop their programming skills, have to carry out extensive testing of new programs which seem to work. How should one proceed?

7.4.1 Variables

A complete list of all variables should be made, if it does not already exist, with an indication of its purpose. Check that all variables have the correct initial value. Check every occurrence of a variable in the program against the list to make sure that it is spelled properly and of the right data type.

7.4.2 Control Structures

Ensure that every structure is properly opened and closed. The system is probably very helpful about this but errors are possible.

7.4.3 Loops

Examine each loop in the program noting carefully the conditions which must be satisfied at entry and the conditions which cause exit. Are these in accord with the intended function? Trace the values of all the variables once or twice round the loop and at exit from it to see that it is working properly. Be particularly suspicious of an unexpected zero case in a REPEAT loop.

7.4.4 Decisions

Are the conditions by which a decision is made in the program the right ones? Is there provision in a case statement for an unusual and incorrect value of the controlling expression?

7.4.5 Procedures and Functions

These can be tested separately quite easily by writing short routines which simply execute the modules with different sets of entry conditions.

7.4.6 Control Paths

Ideally every control path through the program should be tested. If every procedure and function is tested separately and there are no GOTO statements there are as many control paths as there are terminal nodes in the program's structure diagram. All the major ones and as many as possible of the others should be tested.

7.4.7 Data

It is probably impossible to test a program with every possible data set. Constraints can be placed on files of data by means of special data validation programs but if a program is interactive the only rule is that if something is possible, then it will happen. Input from keyboards should be trapped and checked by a special routine which should be designed to accept only valid types of input.

Some test data should have known consequences so that actual results can be compared with these.

Special values, such as very large, very small or negative numbers, might be revealing. Data which is close to the ends of valid ranges might disclose an error.

Generally try to test a program to destruction by any possible means. Some programs work beautifully on harder tasks which the programmer has considered very carefully but fail or perform badly on easy ones.

7.4.8 Field Testing

The ultimate test for a program is what users do with it. It may be necessary to

release a program to just a few users initially. They should be selected for their ability to put a program through its paces and for their ability to react constructively if something does go wrong. Howls of rage are slightly helpful but a careful record of the circumstances of the failure sent promptly is what is really required.

7.4.9 Debugging

When a bug is known or suspected the first thing to do is to locate the part of the program in which it shows evidence of its existence. It is easy to establish the route through which control flows to that point from the start of the program, but it is best to start with the local structure, and backtrack through the various structures on the control route. Carefully updated lists of variable values may be necessary and sometimes this can be hard work. However, comfort can be taken from the knowledge that if sensible methods of analysis and programming have been followed the bugs will be rarer and easier to track down when they do occur.

PROBLEMS

7.1 The following specification is for an interactive game. The user is required to provide input several times during one play. Perform a top down analysis and develop a structure diagram in such a way that all input is carefully validated. Try to ensure that a program written from the diagram will work first time apart from possible typing or minor syntax errors.

ROULETTE GAME SPEC.

(a) The player can specify between 1 and 100 pounds to play and continue until his money is gone or he decides to quit.

(b) Print screen heading "Roulette".

(c) Prompt "How much to play with". User enters sum between 1 and 100 pounds – validate.

(d) Generate random number in range 1 – 36. Store as *chance*. Set *oddch* to 0 if chance is even, otherwise to 1.

(e) Invite user to play *single number* or *Odds/Evens*. Validate choice.

(f) Display funds and invite bet. Validate.

(g) According to choice invite user to pick a number (1 – 36) or to back odds or evens. Validate guess.

(h) According to guess and number generated, either
  (1) Pay winnings of 35 times bet
  or (2) Pay winnings of 2 times bet
  or (3) Pay nothing (stake is automatically kept)

(i) Report what has happened to user.

(j) Adjust cash funds and report to user inviting him to play again or quit. (30+)

7.2 Write and test the program from the structure diagram. Make a list of all possible errors at each input stage and test the effect. (30+)

Note

The use of 30+ as an indication of difficulty implies that the problem is complex enough to be beyond the range for which the scheme is really satisfactory.

CHAPTER 8

More about Structures and Control

Forms more real than living man,
Nurslings of immortality.

Prometheus Unbound, Percy Bysshe Shelley.

8.1 REPETITION

8.1.1 FOR loops

The counter in a FOR is usually incremented by one each time the loop is traversed but sometimes this is not the necessary increment. Fractional or negative increments are allowed though the former can cause trouble in certain circumstances. One can write.

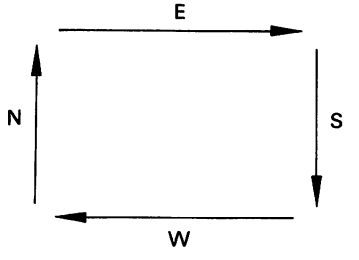
```
        FOR inc := 0 TO 5 STEP 0.1 DO
or      FOR inc := 7 TO 1 STEP -1 DO
```

Example 8.1

Write a program to draw the frame of the picture in Fig. 5.1 in one continuous clockwise movement.

METHOD

(1) The four corners of the square are: (10,3) (60,3) (60,21) (10,21).
(2) Instead of dealing with just two types of lines, we now have four types which we might specify as east, south, west, north, according to the direction of the drawing.



(3) Because we now have different directions the most suitable data is a set of three items:

start position
end position
direction

The direction should be specified as E, S, W or N for clarity and this will determine which way the other data is treated.

(4) One could write a clever routine which could deal with all cases in the same FOR loop but, for simplicity and clarity, we will use a CASE structure.

PROBLEM ANALYSIS/PROGRAM DESIGN

(1) A variable dir$ will accept the character E. S, W or N.
(2) The appropriate FOR statement will be selected and a line drawn.
(3) The four sets of data will be picked up in a FOR loop.

PROGRAM

```
DIM direction$ OF 1
FOR side := 1 TO 4 DO
  READ startac, startdown, endac, endown, direction$
  CASE direction$ OF
  WHEN "E"
    FOR ac := startac TO endac DO EXEC cursor(ac, startdown)
  WHEN "S"
    FOR dn := startdown TO endown DO EXEC cursor(startac,
                                                        dn)
  WHEN "W"
    FOR ac := startac TO endac STEP -1 DO
      EXEC cursor(ac, startdown)
    NEXT ac
  WHEN "N"
    FOR dn := startdown TO endown STEP -1 DO
      EXEC cursor(startac, dn)
    NEXT dn
  OTHERWISE
    PRINT "Data Error"
  ENDCASE
NEXT side
DATA 10,3,60,3,"E",60,3,60,21,"S",60,21,10,21,"W",10,21,10,
                                                        3,"N"
PROC cursor(ac, dn)
  (content of cursor procedure for system used)
  PRINT "*"
ENDPROC
```

Before leaving the FOR loop a word should be said about its internal mechanism. Upon encountering a FOR statement the system must:

(1) Set up the counter for the loop and assign its initial value.
(2) Make a note of the final value of the counter.
(3) Make a note of starting point of the statements within the FOR loop.

When the corresponding NEXT is encountered the system must:

(1) Increment or decrement the counter.
(2) Test the counter value against the noted final value.
(3) Send control back to the starting point or out of the loop.

The significance of this information for the programmer arises if he should wish to use the counter value after exit from the loop. Because it has just been incremented or decremented its value will be one step beyond the final value given in the FOR statement. A properly implemented FOR loop should execute the statements *zero* times if the initial and final values in the FOR statement imply this.

8.1.2 WHILE loops

The FOR loop deals effectively with exit on a count and the REPEAT loop enables exit on a condition but it does have a defect. Suppose we simulate a football or rugby player running with the ball.

```
REPEAT
   PRINT "Place one foot in front of the other."
UNTIL (tackled)
```

It is possible for a player to receive the dreaded "hospital pass" which arrives at the same instant as his opponent. Before he can take even one step his run is ended. The REPEAT loop cannot cope with this situation because the statements within the loop must be executed once before the condition is tested. An alternative type of loop with exit on a condition is available and it has the form exemplified.

```
WHILE (not tackled) DO
   PRINT "Place one foot in front of the other."
ENDWHILE
```

Notice that the condition is turned round and it may seem unnatural to say in effect:

```
WHILE something is not true DO
   Perform this action
```

but this is the right construct for the case where zero executions must be allowed as a possibility. In all other cases the programmer has a choice and the most

popular one seems to be the REPEAT loop, certainly in the early stages of programming. However, the balance sometimes changes in more advanced work partly because the need arises more frequently and partly because it is a more general construct, and safer than the REPEAT loop.

The history of the WHILE notation goes back at least as far as ALGOL 60 and it is a powerful concept. We have seen how it can do anything that a REPEAT loop can do and it is obvious that, by setting up a counter. we could make it behave like a FOR loop. Why then do we bother with REPEAT and FOR?

The answer is simply that these structures often express the sense of what is being done in terms that people can more readily appreciate. For example some might think it reasonable to say of Drunken Duncan:

```
WHILE (he is still on the grid) DO
   Keep moving
ENDWHILE
```

though others might prefer:

```
REPEAT
   Keep moving
UNTIL (he is off the grid)
```

Essentially one has to choose between:

```
WHILE (reason for looping) DO
   (action)
ENDWHILE
```

and

```
REPEAT
   action
UNTIL (reason for not looping).
```

If there is a choice the writer advocates the use of the one which seems most natural, but a word of caution is needed about the word WHILE. It has several meanings in natural language but it is used in computing to mean strictly during the time that. Of other natural usages the one which might be confusing is found occasionally in Shakespeare and often in the North of England:

'I am waiting while the others arrive'.

which gives while the same meaning as until.

A simple example of the need for WHILE loops arises if we develop the NIGHT algorithms (section 6.1) into a more realistic simulation.

Example 8.2

Assume that a person goes to sleep within half a minute either side of midnight. Call this time zero and count each successive minute as 1, 2, 3 . . . If the sleeper wakes within half a minute of time 3 we will say that he awoke at time 3.

Suppose that, up till time 420, in any minute there is a probability of 0.01 that he will wake up and if he does he will go to sleep again immediately. After time 420 (about 7.00 a.m.) the probability of his waking in any minute increases to 0.05 and when he wakes he gets up.

METHOD

(1) A FOR loop will be used to test all times from 1 to 420 and a note will be made of waking times.
(2) After time 420 a WHILE loop will be used to continue the testing with the new condition.

PROBLEM ANALYSIS/PROGRAM DESIGN

(1) The variable *tim* will be used to control the FOR loop and on exit it will have the value 421.
(2) The WHILE loop will take the form:

```
WHILE RND >0.05 DO
  tim := tim +1
ENDWHILE
PRINT "Awoke and arose at"; tim
```

This allows the possibility of waking and arising at time 421.

PROGRAM

```
FOR tim := 1 TO 420 DO
  IF RND <0.01 THEN PRINT "Awoke at time"; tim
NEXT tim
WHILE RND >0.05 DO
  tim := tim +1
ENDWHILE
PRINT "Awoke and arose at time"; tim
```

8.2 DECISIONS

We have seen that a binary decision can be based very sensibly on a condition which may be true or false. A clause is selected in a CASE structure slightly differently. One of a set of values is chosen and a selection from multiple choices is made. Sometimes we have a decision to make from multiple choices which do not convert naturally into discrete sets of values. An example will clarify this.

Example 8.3

Write a program to accept a percentage mark and convert it into a class of degree according to the given scheme. Marks are computed to one decimal place.

Mark	*Degree*
Under 40	Fail
40 – 49.9	Third class pass
50 – 59.9	Lower second class pass
60 – 69.9	Upper second class pass
70 and over	First class pass

We could use a slightly cumbersome sequence of IF/THEN/ELSE constructs as follows:

```
INPUT mark
IF mark < 40 THEN
   PRINT "Fail"
ELSE
   IF mark < 50 THEN
      PRINT "Third class pass"
   ELSE
      IF mark < 60 THEN
         PRINT "Lower second class pass"
      ELSE
         IF mark < 70 THEN
            PRINT "Upper second class pass"
         ELSE
            PRINT "First class pass"
         ENDIF
      ENDIF
   ENDIF
ENDIF
```

Two things make this construction clumsy and awkward to read:

(1) The use of ELSE IF three times.
(2) The string of ENDIFs at the end.

The word ELIF has been invented to make possible a neat multiple decision based on conditions rather than sets of discrete values. ELIF gets exactly the same status as ELSE in the sense that it closes one section of the structure, opens another and does not cause any extra indenting. This means that the program can now be written more sensibly. Logically ELIF means ELSE IF.

```
INPUT mark
IF mark < 40 THEN
   PRINT "Fail"
ELIF mark < 50 THEN
   PRINT "Third class pass"
```

```
ELIF mark < 60 THEN
   PRINT "Lower second class pass"
ELIF mark < 70 THEN
   PRINT "Upper second class pass"
ELSE
   PRINT "First class pass"
ENDIF
```

There has been some comment in computer literature about the confusion that can result from bad usage, or even good usage, of IF/THEN/ELSE clauses. Certainly the ordinary nesting of such constructs should be avoided or done very carefully. The neat layout, system-forced indenting and ELIF help to eliminate some of the trouble and it seems clear that these structures, used sensibly, are valuable in COMAL.

8.3 PROCEDURES

8.3.1 The Post Holes Problem

It would be useful at this stage to recall the post holes program as far as we have developed it. In section 5.2.1 we examined the use of parameters as a method of passing information to a procedure. However, we did not discuss a method by which the program could choose an actual parameter which could be "large", "medium" or "small". Perhaps the simplest way to do this is read the three values into a string array and choose one array element by means of a random number. The relevant instructions are:

```
DIM treesize$(3) OF 6
K=1 TO 3 DO READ treesize$(k)
DATA "small", "medium", "large"
```

The program can now be brought up to date and is listed in Fig. 8.1. It should be easy to follow but perhaps two notes will help:

(1) The use of a variable *soft* or *tree* where one would expect a conditional expression is allowed by special arrangement. If the variable takes the value zero it is treated as FALSE and all other values are treated as TRUE.

(2) The statement

```
IF tree THEN EXEC treechop(treesize$(RND(1,3)))
```

is quite powerful. The RND(1,3) evaluates to 1, 2 or 3 and this is used as the subscript for *treesize*$ which then takes the value "small", "medium" or "large". This word is passed as an actual parameter to procedure *treechop*. For example if the RND function produced 2 the effect would be

```
IF tree THEN EXEC treechop("medium")
```

and then the procedure would know what was required.

FOREMAN'S INSTRUCTIONS

I want a corner post hole at each corner of this field and along each side you must dig post holes fifteen metres apart. Where the ground is soft we need a half-metre hole and elsewhere, a third metre hole. If there is a tree in the way get Fred to deal with it. Tell him what size the tree is: small, medium or large and he will know what to do. You can get the length of each side of the field from a data sheet.

```
DIM treetype$ OF 6, treesize$(3) OF 6
FOR k := 1 TO 3 DO READ treesize$(k)
FOR side := 1 TO 4 DO
   PRINT "Dig a corner post hole."
   READ sidelength
   REPEAT
      tree := RND(0,1)
      IF tree THEN EXEC treechop(treesize$(RND(1,3)))
      soft := RND(0,1)
      IF soft THEN
         PRINT "Dig a half metre hole."
      ELSE
         PRINT "Dig a third metre hole."
      ENDIF
      sidelength := sidelength - 15
   UNTIL sidelength < 15
next side
PROC treechop(treetype$)
   CASE treetype$ OF
PROC treechop(treetype$)
   CASE treetype$ OF
   WHEN "small"
      PRINT "Pull out tree"
   WHEN "medium"
      PRINT "Cut above base. Pull out root."
   WHEN "large"
      PRINT "Cut down tree. Dig out root."
   OTHERWISE
      PRINT "Tree type not known."
   ENDCASE
ENDPROC treechop
DATA "small", "medium", "large"
DATA 100,120,140,110
```

Fig. 8.1 – The Postholes Program with a Value Parameter.

8.3.2 Reference Parameters

If we wish to pass information back to the main program we must provide a variable to receive the information. Such parameters are called variable or reference parameters because they must be variables, not values or expressions, and the procedure is able to refer to them to pass information.

The corresponding formal parameter in the procedure must be marked REF, otherwise the notation and syntax is not changed. As an example suppose that we wish the procedure *treechop* to compute a cost for the job of removing a tree. This cost can then be passed back to the main program. A scale of charges might be fixed.

	£
Small tree	25
Medium tree	50
Large tree	75
Extra charge	0 – 25

The last part can be computed randomly and the routine shown below can be placed in procedure treechop.

```
CASE treetype$ OF
WHEN "large"
   charge := 75
WHEN "medium"
   charge := 50
WHEN "small"
   charge := 25
ENDCASE
charge := charge + RND(0,25)
```

This structure already exists in the procedure and we merely add in the charge assignment statements.

The variable charge must be the formal reference parameter and the procedure heading becomes:

```
PROC treechop(treetype$, REF charge)
```

The variable cost will be the actual reference parameter and the procedure call becomes:

```
EXEC treechop(treesize$(RND(1,3)), cost)
```

Having received the information the main program can print it.

```
PRINT cost; "pounds – tree fee."
```

A summary of the properties and classification of the parameters used in procedure treechop might be useful.

FORMAL PARAMETERS

treetype$ A value parameter which received the value "small", "medium" or "large" from the main program.

charge A reference parameter which refers the charge for tree felling back to the main program.

ACTUAL PARAMETERS

treesize$( ) The array element which passes the value "small", "medium" or "large" to the procedure.

cost The reference parameter to which the procedure passes the charge for tree felling.

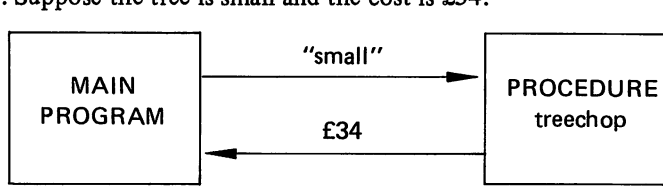
The procedure does not use any variables other than the formal parameters described above and these do not occur in the main program. If they had been used in the main program it would not matter because the system treats formal parameters as local to the procedure. This means that the word *charge* could have been used as an actual parameter in the main program without confusion to the system. On the other hand working variables in a procedure are not automatically local. The word CLOSED placed after the parameters' closing bracket would make them local. This would make the procedure a completely "black" box. It could be placed in any program without fear of conflict with variable names accidently chosen similarly by another programmer.

The final point to be noted about procedure parameters now is that, if, having CLOSED a procedure, it is necessary to make particular variables non-local this might be done by writing GLOBAL followed by the list of variables to be made non-local. Not all systems allow this, and manuals should be consulted.

8.4 FUNCTIONS

A function in computing is simply a procedure whose name is used as a variable to carry information back to the main program. It follows that a function can only be used when there is exactly one piece of information to return to the main program. COMAL keywords such as RND, SIN, COS, etc., are functions whose definitions are built in to the system.

The advantages of a function over a procedure are convenience and economy. The use of a reference parameter is avoided because the function name does the job. Consider again what happens when the procedure in the post holes problem is called. Suppose the tree is small and the cost is £34.



Clearly the procedure satisfies the condition for a function that exactly one piece of information goes back to the main program. It is only necessary to rewrite things slightly differently. The procedure loses its formal parameter, *charge*, and uses instead the name, *treechop*.

```
PROCEDURE treechop(treetype$)
  CASE treetype$ OF
  WHEN "large"
    PRINT "Cut down tree. Dig out root."
    charge := 75
  WHEN "medium"
    PRINT "Cut above base. Dig out root."
    charge := 50
  WHEN "small"
  PRINT "Pull out tree."
    charge := 25
  ENDCASE
  treechop := charge + rnd(0,25)
ENDPROC
```

The procedure call and printing in the main program is simplified from:

```
IF tree THEN EXEC treechop(treesize$(RND(1,3)), cost)
PRINT cost; "pounds – tree fee".
```

to

```
IF tree THEN PRINT treechop(treesize(RND(1,3))); "pounds –
                                                  tree fee".
```

This might be less clear than the original because the function name, *treechop*, is being used to convey a number (representing a cost) back to the main program. It is not a meaningful name in that sense, but it is now doing two jobs. It is still also producing the printout which indicates what type of tree has been removed and by what method. It should show the reader what a function is and how it is related to procedure.

It should be noted that the variable, *charge*, in the function *treechop* is no longer a parameter. It is just a working variable. It is not local to the procedure, though it could be made so by using the word CLOSED.

8.4.1 A Random Number Generator

The RND function supplied with most COMAL or BASIC systems is rarely good enough to withstand careful statistical testing. Without worrying too much about the theory concerning the generation of pseudo random numbers (they are not really random) there is a very simple method which can produce reasonable results.

METHOD

(1) Start with a variable x given an initial value 0.82337.
(2) Compute $y := 25.173 * x + 3.921$.
(3) Remove the whole number part of y using $y - \text{INT}(y)$.
(4) Take this value as the first random number and repeat the process using the same value as the new x. The three numbers used have been discovered by trial and error. Others would do the job.

PROBLEM ANALYSIS/PROGRAM DESIGN

(1) Weshall need a function *ran* which uses a global variable x and computes the new x as indicated.
(2) x will be initialised at the start of the main program.
(3) Twenty pseudo random numbers will be computed.

PROGRAM

```
x := 0.82337
FOR count := 1 TO 20 DO
  PRINT ran;
NEXT count
PROC ran
  y := 25.173 * x + 3.921
  x := y - INT(y)
  ran := x
ENDPROC
```

COMMENTS

(1) The variable x must be global in order to initialise its value outside the function definition.
(2) Because x is global its value is preserved from call to call of the function.
(3) The values 0.82337, 25.173 and 3.921 were obtained by extensive trial and error and testing. They produce reasonable results on one system but such things need extensive testing on the system in which they are used.
(4) The function is not written for efficiency and could be made to work faster, but only by using more global variables.

8.5 RECURSION

Any definition is said to be recursive if it uses within the definition the thing being defined. Even a cursory treatment of recursion is beyond the scope of this book but we can say what recursion means in computing and a serious example will be given in a later chapter (section 10.4.3).

A formal definition of a noun phrase can be given neatly using recursion.

```
⟨noun phrase⟩ = ⟨noun⟩ | ⟨adjective⟩ ⟨noun phrase⟩
⟨adjective⟩ = large | small | blue | white | red | round | square | thick | thin.
⟨noun⟩ = dog | shape | man
   = means "is defined as"
| means "or".
```

Clearly "shape" is a noun phrase. It follows that "red shape", "thin red shape" and "square thin red shape" are also noun phrases. Unfortunately such phrases as "round square shape" also fit the syntax requirements, but it is well known that a grammar or syntax can only define what is grammatically correct. It cannot stop people from writing meaningless or factually wrong sentences.

In computing, a procedure or function is said to be defined recursively if it contains a call to itself. If, during execution, the recursive call is actually executed the procedure is said to be called or executed recursively.

The following simple example has a recursively defined procedure and it can be executed recursively.

```
EXEC line(10)
PROC line(num)
  PRINT "*";
  IF num> 1 THEN EXEC line(num−1)
ENDPROC
```

The procedure is called for the first time, *num* takes the value 10 and a star is printed. At the second call from within the procedure the actual parameter becomes 9 and a new variable which the system might call *num2* is established for the second execution. This takes the value 9, a second star is printed. The process continues until the actual parameter becomes zero.

This is an odd way to print ten stars but most simple examples of recursion are not very useful. There are better treatments of the subject in almost any book about programming in a modern language such as Pascal. Examples tend to be either trivial or to do with games or puzzles or, if they are really useful, rather difficult.

At this stage we will be content with one trivial and much quoted example which demonstrates that something which is defined recursively can sometimes be programmed recursively in a natural way. There is in mathematics something called the Factorial Function such that:

```
Factorial (5) = 5X4X3X2X1 = 120
Factorial (4) = 4X3X2X1   =  24
Factorial (3) = 3X2X1     =   6
Factorial (2) = 2X1       =   2
Factorial (1) = 1
```

The reader can observe that:

Factorial (5) = 5 X Factorial (4)
Factorial (4) = 4 X Factorial (3)

or generally:

Factorial (number) = number X Factorial (number−1)

This together with the fact that

Factorial (1) = 1

can be taken as a definition of the factorial function. It is a recursive definition because the thing being defined is used in the definition. The definition is rescued from circularity by the second part which makes Factorial (1)=1.

Suppose, for example, that we want to compute Factorial (4). From the definition:

Factorial (4) = 4XFactorial (3)
= 4X3XFactorial (2)
= 4X3X2XFactorial (1)
= 4X3X2X1
= 24

This can be programmed directly from the definition.

```
PRINT fact (5)
PROC fact(num)
   IF num > 1 THEN
      fact := num * fact(num−1)
   ELSE
      fact := 1
   ENDIF
ENDPROC
```

fact is clearly a function and it is defined recursively. On first entry to the procedure or function, *num* has the value 5 so there is a recursive procedure or function call. In this and subsequent incarnations of the procedure, each time the variable *num* is used the system will alter it to make it unique and specific to that particular incarnation.

The way the system manages to cope with a whole set of executions of the function each with its own version of the parameter *num* is not our concern. But it is interesting to note that, at least occasionally, an elegant and natural recursive solution can be used as an alternative to a more conventional kind of loop. In this case the loop would be very simple.

```
Fact := 1
FOR num := 5 TO 1 STEP -1 DO
  fact := fact * num
NEXT num
PRINT fact
```

The importance of recursion lies not in these examples, but in more complex problems, particularly in the fields of systems software and artificial intelligence. In some such cases the conventional solution may be more difficult than the recursive one.

8.6 THE GOTO STATEMENT

In COMAL, as in BASIC and almost every computer language, it is possible to transfer control directly to any other part of a program. As an example we can simulate the digging of a row of post-holes in this way.

```
length := 110
again
PRINT "Dig a post hole."
length := length - 15
IF length >= 15 THEN GOTO again
```

The label, *again,* is used to identify the target of the GOTO statement. In BASIC a line number would be used instead of a label. Sometimes a bug is removed from a BASIC program by deleting a line, only to create another bug if the deleted line happened to be the target of a GOTO statement.

However, the major argument against the use of the GOTO statement is that it can and usually does lead to a quite bewildering variety of structures. The idea of just half a dozen essential types of program elements and just three relationships between them is destroyed. Anything becomes possible, and a glance through the popular BASIC-oriented computer journals will show that the famous phrases such as "Spaghetti-like control paths" or "Programs like a snakes and ladders game" or "Rats' nest" are well justified. This is not to say that experienced , disciplined professionals cannot write well-structured programs in BASIC or old FORTRAN. They can and do. The next chapter will show how careful one has to be to do it and how it is, in any case, necessary to understand the use of the correct structures first.

The reader who is learning to program and to analyse problems in terms of COMAL should need a GOTO statement very rarely, if at all. Sometimes, having used one, a little further study will show how the same result can be achieved using the standard structures only. The major use of a GOTO might be the case where a data error or some other condition arises which calls for the abandonment of the program. Control can be directed to a routine which prints diagnostic information before halting the program.

Readers who wish to use older languages, such as BASIC, with inadequate structural features can see in the next chapter how GOTO statements can be used in suitably constrained ways.

PROBLEMS

8.1 When children learn division they write such things as $23 \div 7 = 3$ remainder 2. The remainder or *modulus* is the residue after subtracting a smaller number from a larger as many times as possible without needing negative numbers. For example

23 MOD 7 is 2
and 48 MOD 21 is 6

Write a program to input two numbers and find the value of *first* MOD *second*. Ensure that the program always gives a correct result. (4)

8.2 Read characters from a DATA statement until * is encountered. Place the characters in a string ignoring the *. Print the characters in reverse order together with a count of the total number. Ensure that the program works correctly even if there are no characters other than *. (7)

8.3 An experiment requires that a die be thrown until a six appears. The number of throws before the six is called a run and it may be zero. Find the average value of a run in the simulation of twenty such experiments. (8)

8.4 Write a program which accepts as input a positive whole number and outputs the equivalent in Roman numerals. For example: 1666 = MDCLXVI. Numbers such as 4 and 9 may be written as IIII and VIIII rather than IV and IX. (9)

8.5 Euclid's algorithm can be used to find the greatest common divisor of two numbers.

Example

Find the greatest common divisor of 21 and 6

	21	6
Take lesser from greater	−6	
	15	
Take lesser from greater	−6	
	9	
Take lesser from greater	−6	
	3	
Take lesser from greater		−3
		3

Stop, because numbers are now equal. The greatest common divisor of 21 and 6 is 3.

Write a program to compute the GCD of any two positive numbers. Ensure that the program does not fail if the numbers are equal. (9)

8.6 In a gambling game the banker agrees to pay £1 for each consecutive "head" when a coin is tossed repeatedly. The game stops when a "tail" first appears. Try to establish a fair stake for a player to pay for one such game by simulating twenty games and finding the average payout. (9)

8.7 If the game in problem 8.6 is modified so that the banker doubles his payout each time a "head" is thrown after the first (payments are £1, £2, £4, £8 . . .) the problem of computing a fair stake becomes difficult to solve mathematically. Modify the above program to compute the fair stake ten times. The results might be inconsistent. (14)

8.8 Salesmen send in daily reports of items sold by placing a number against some of 100 possible items. These are converted into DATA statements terminated by −1, −1. For example,

DATA 2,6,3,15,7,94,−1,−1

would indicate that a salesman had sold 2 of item 6, 3 of item 15 and 7 of item 94. Sometimes a salesman sells nothing and a nil return is indicated by: DATA −1, −1.
The item numbers are arranged in ascending order of price and salesmen's commission is calculated by the formula

(item number) * (number sold) / 10

for all items.
Compute the commission for each of four salesmen, and the total for all, from the following data. (10)
DATA 2,6,3,15,7,94,−1,−1
DATA 7,13,5,21,13,83,6,94,−1,−1
DATA −1,−1
DATA 8,11,7,34,11,60,−1,−1

8.9 A hairdressing salon requires that, for each customer a card be completed to show extra services provided. These are numbered and each employee places ticks against any of ten numbers. From the cards a DATA statement is prepared. For example:

DATA 3, 7, 9, −1

would indicate that items 3, 7 and 9 had to be paid for. No extra services would produce:

DATA −1

Write a program to read five such DATA lines and print:
(a) The total cost to each customer calculated as £3. + extra items as priced below.
(b) A list showing how many times each service was used. (14)

1	2	3	4	5	6	7	8	9	10
2	4	2	1.5	2.5	3	2	1.5	6	8

8.10 In a fairground game six dices are thrown and the quality of the prize is determined by the score. In one hundred such games how many scores would you expect in each of the ranges 6-10, 11-15, 15-20, 21-25, 26-30, 31-35? Simulate one hundred throws and use the ELIF construction to grade the results. If six sixes appear print the words "SPECIAL PRIZE". (16)

8.11 Generate one hundred random alphabetic characters and use the ELIF construction to grade them in five groups A-F, G-K, L-Q, R-U and V-Z. Place the letters in five strings and Print each string together with counts of the number of letters in each group. (20)

8.12 Write a program to read a sample of text and plot a graph showing the frequency of occurrence of each letter. Write the upper case letters across the top of the screen and plot stars downwards to indicate the frequencies. For simplicity consider text written entirely in upper case letters. Ignore spaces, punctuation, digits or other characters. (23)

```
DATA "HERE WITH A LOAF OF BREAD BENEATH THE BOUGH,"
DATA "A BOOK OF VERSE, A GLASS OF WINE AND THOU"
DATA "BESIDE ME, SINGING IN THE WILDERNESS,"
DATA "AND WILDERNESS IS PARADISE ENOW."
```

8.13 Write a program to count the number of words in the following text. A word is recognised by the fact that it is a sequence of letters followed by a non-letter. The text is terminated by *. The program should not fail in any circumstances provided the text is properly terminated. (17)

```
DATA "SHE IS FOREMOST OF THOSE THAT I WOULD HEAR
PRAISED.*"
```

8.14 Modify the program from problem 8.13 so that it computes and prints the average word length of the words in the text, the average sentence length, and the automated readability index from the following formula.

ari := 0.5* (average sentence length) + 4.71* (average word length) – 15.43 (modified for use in UK) (22)

```
DATA "THE ICE TALONS SET HARDER IN THE LAND."
DATA "NO TWITTER OF FINCH OR LINNET WAS HEARD"
DATA "ON THE BURROWS, FOR THOSE WHICH REMAINED"
DATA "WERE DEAD. VAINLY THE LINNETS HAD SOUGHT"
DATA "THE SEEDS LOCKED IN THE PLANTS OF THE"
DATA "GLASSWORT. EVEN CROWS DIED OF STARVATION.*"
```

8.15 Rewrite program 8.14 with two procedures, *gettext* and *compute*. The main program should call *gettext* which places data in *text*$ and then call the second procedure, *compute*, which computes the automated readability index, *ari*. The second procedure should take *text*$ as a value parameter and pass back *ari* through a reference (or variable) parameter. The main program can then print the value of *ari*. (28)

8.16 Rewrite program 8.15 so that procedure, *compute*, becomes a function, *ari*, which conveys the result back to the main program. (28)

CHAPTER 9

Structured Programming with Basic

. . . That blessed mood
In which the burden of the mystery,
In which the heavy and the weary weight
Of all this unintelligible world
Is lightened . . .

Michael, William Wordsworth.

9.1 BASIC

Apart from miniscule versions which have names such as Tiny BASIC or Integer BASIC the name BASIC, qualified only by the name of a computer or company, has been applied to a range of implementations from a primitive 4k Sinclair ZX80 BASIC to a 96k HP BASIC which has all that COMAL 80 has, except system-forced indenting, plus a good deal more, including many facilities for scientific work such matrix manipulation. Other BASICs have been strengthened with a bias towards commercial data processing and some have outstanding graphics or communications facilities.

Users of BASICs which have control structures which differ from COMAL on only minor points of syntax or notation will not find this chapter useful. Others, to the extent that their BASIC is without some of the structures, should find it helpful. The control structures assumed in this chapter are available in a large number of BASIC implementations and they will be described briefly.

Note
[ ] means that parts within are optional
a,b,c . . . z are numeric variables
A,B,C are numeric expressions

9.1.1 FOR loop

SYNTAX

```
FOR x = A TO B [STEP C]
—
—
—
—
NEXT x
```

EXAMPLE

```
FOR p = 1 TO 4
PRINT "Dig a post hole"
NEXT p
```

9.1.2 CONDITIONAL statement

SYNTAX

IF ⟨condition⟩ THEN ⟨statement⟩

EXAMPLE

IF s = 1 THEN PRINT "Dig a half metre hole".

9.1.3 GOTO statement

SYNTAX

GOTO ⟨line number⟩

EXAMPLE

GOTO 70

9.1.4 Multiple Branching

SYNTAX

ON A GOTO ⟨line number⟩, ⟨line Number⟩, . . .

EXAMPLE

ON INT(RND(0)*3)+1 GOTO 30,80,120

Notes

1. INT(RND(0)*3)+1 is the equivalent of RND(1,3)
2. Depending on the value of the expression, control will go to line 30, line 80 or line 120.

SYNTAX

ON A GOSUB ⟨line number⟩, ⟨line number⟩, . . .

EXAMPLE

ON INT(RND(0)*3)+1 GOSUB 100,200,300

Note

The effect is the same as ON . . . GOTO except that a subroutine jump is performed. See the next item.

9.1.5 Subroutine

SYNTAX

```
GOSUB⟨linenumber⟩
```

⟨linenumber⟩ –
–
–
–
RETURN

EXAMPLE

```
GOSUB 200

200 PRINT "Subroutine"
210 RETURN
```

Note

A BASIC subroutine is a primitive form of procedure. It is not properly named and its start is not defined. Otherwise it performs like a simple COMAL procedure without formal parameters.

9.1.6 Random Number Function

Most BASICs allow only the generation of a number in the range 0 to 1 by some function like RND(1). It is also a little unusual to find a numeric value acceptable as a condition so that one has to write.

```
IF s = 1 THEN PRINT "Ground is soft"
```

rather than

```
IF s THEN PRINT "Ground is soft"
```

This is no great loss because neither statement is particularly helpful to the reader. But it does mean that binary decisions are best treated on the lines:

```
s = RND(1)
IF s < 0.5 THEN ...
```

9.2 STRUCTURAL PRINCIPLES AND RULES

The central theme of structured programming is that structure should reflect function. This theme can be considered in three stages: Problem analysis/ program design, coding and presentation.

A structure diagram, achieved essentially by a process of top down analysis, or an equivalent (iterative graph or design structure diagram) might be the result of good program design. It is equally acceptable to think in COMAL and, in effect, write a COMAL program as the program design for conversion into

BASIC. Each structural element of the program should have one entry point and one exit point. Elements should be related only in the ways already discussed: sequential, subordinate, procedure calls.

The coding should reflect the program design, that is, it should indicate what the program will actually do. The trouble here is that a running program is a dynamic thing whereas a program on paper is static. That is why it should show clearly which parts are loops and which kind, and which parts are binary or multiple decisions. It should also show clearly the function of any procedures. Dijkstra puts this very succinctly.

> We should do our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

Obviously programs must be as easy as possible to read. If they are not readable they cannot reflect anything comprehensible by people.

The good presentation of a program in BASIC is often frustrated by the system. Keywords in upper case, spacing, indenting are all helpful, but most BASIC implementations remove some or all of these things even if the programmer puts them in. The only aid to good presentation in many systems is the REM statement. Nevertheless it is helpful to write programs with these helpful features before they are keyed in to the machine. It is probably better to publish them in this way too, though the risk of error is higher when machine-produced listings are not used.

In deciding on the particular BASIC constructs the following rules have been observed:

(1) The constructs will simulate very closely the structures of COMAL.
(2) REM statements will be used where necessary to show explicitly the opening, closing and intermediate keywords of each construct.
(3) The target of GOTO or GOSUB statements will always be a REM statement so that additions or deletions can take place within the structure without causing a GOTO or GOSUB to lose its target.
(4) Indenting will be used and the distinction between keywords and other words of programs will be maintained though it is recognised that most systems would not recognise or would remove such features. For this purpose words of COMAL used in REM statements will be treated in the same way as keywords.

9.3 BASIC CONSTRUCTS

9.3.1 FOR loop

PROGRAM DESIGN

```
FOR post := 1 TO 4 DO
```

```
    PRINT "Dig a corner post"
  NEXT post
```

BASIC PROGRAM

```
10 FOR p=1 TO 4
20     PRINT "Dig a corner post hole"
30 NEXT p
```

9.3.2 REPEAT loop

PROGRAM DESIGN

```
fieldlength := 100
REPEAT
  PRINT "Dig a hole"
  fieldlength := fieldlength - 15
UNTIL fieldlength < 15
```

BASIC PROGRAM

```
10 f = 100
20 REM REPEAT
30     PRINT "Dig a hole"
40     f = f - 15
50 IF f >= 15 THEN GOTO 20
```

Notes

(1) The condition has been reversed. Some programmers would write:

```
NOT(f < 15)
```

in order to preserve a stronger relationship between analysis and coding.

(2) The end of the REPEAT loop is not as obvious as it might be, but it seems verbose to add an extra line REM UNTIL *fieldlength* < 15.

9.3.3 Binary Decision

PROGRAM DESIGN

```
soft := RND(1)
IF soft < 0.5 THEN
  PRINT "Dig a half metre hole".
ELSE
  PRINT "Dig a third metre hole".
ENDIF
```

BASIC PROGRAM

```
10 s = RND(1)
20 IF s > 0.5 THEN GOTO 50
30    PRINT "Dig a half metre hole"
40 GOTO 70
50 REM ELSE
60    PRINT "Dig a third metre hole"
70 REM ENDIF
```

Notes

(1) Some BASICs allow local forms of IF . . . THEN . . . ELSE and in simple cases one could write:

```
IF s < 0.5 THEN PRINT "Dig a half metre hole" ELSE PRINT "Dig
a third metre hole"
```

Such "one-liners" are better for the simple cases but students will need to simulate the global IF . . . THEN . . . ELSE structure at some stage.

(2) The rectangles are written at the end of a line because, at the time of writing, the line number target is not known. When it becomes known the rectangles are filled.

9.3.4 Procedure

PROGRAM DESIGN

```
tree := RND(1)
IF tree< 0.5 THEN EXEC treechop
-
-
-
-
-
PROC treechop
   PRINT "Cut down tree"
   PRINT "Dig out root"
ENDPROC
```

BASIC PROGRAM

```
10 t = RND(1)
20 IF t < 0.5 THEN GOSUB 110
30 REM treechop
   -
   -
   -
   -
   -
```

```
100 STOP
110 REM PROC treechop
120     PRINT "Cut down tree".
130     PRINT "Dig out root".
140 RETURN
```

Notes

(1) The STOP at line 100 is necessary to prevent procedures being entered accidently. It would not be needed if another procedure preceded this one because RETURN would have the same protective effect.

(2) All information must be passed to or from procedures by means of global variables. Particular assignments should be placed immediately before the GOSUB statement if possible.

9.3.5 WHILE loop

PROGRAM DESIGN

```
tackled := RND(1)
WHILE tackled < 0.5 DO
   PRINT "Take one step"
   tackled := RND(1)
ENDWHILE
```

BASIC PROGRAM

```
10 t = RND(1)
20 REM WHILE
30 IF t > 0.5 THEN GOTO 70
40     PRINT "Take one step"
50     t = RND(1)
60 GOTO 20
70 REM ENDWHILE
```

9.3.6 Multiple Decision

PROGRAM DESIGN

```
level := INT(RND(1)*3)
CASE level OF
WHEN 0
   PRINT "Use a one-fifth metre support"
WHEN 1
   PRINT "Use a two-fifth metre support"
WHEN 2
   PRINT "Use a three-fifth metre support"
ENDCASE
```

BASIC PROGRAM

```
 10 1 = INT(RND(1)*3)
 20 ON 1 GOSUB 40, 70, 100
 30 GOTO 130
 40 REM WHEN 0
 50    PRINT
 60 RETURN
 70 REM WHEN 1
 80    PRINT
 90 RETURN
100 REM WHEN 2
110    PRINT
120 RETURN
130 REM ENDCASE
```

9.3.7 A Complete Program

The rules given in the preceding sections are not claimed to be easy to use. In the short term they are troublesome, they take time to learn and they do not seem entirely necessary. A complete program in BASIC is given below to give an impression of just how awkward BASIC programs can be to read even when they are about as well-written as one can hope for. The everyday reality is that BASIC programs are not usually written with such care and the results are inevitably worse, varying from slightly more difficult, to almost impossible, to read. The indenting, lower case letters and spacing are used to help readability. While COMAL systems usually insert such features, most BASIC systems take them out even if the user puts them in. The program is based on that of section 8.3.1.

Notes

(1) Certain parts of the program might be omitted without causing failure but it is intended to represent, in a simple way, the use of features and constructs which will continue to work in more realistic situations. For example the tree size parameters are generated even though they cannot be used to select a CASE routine.

(2) The rectangles are inserted at the time of writing a line with a forward jump. The line number is filled in when it is known.

PROBLEMS

9.1 Search computer journals for examples of BASIC programs which are poorly structured. What procedure would you have to adopt to try to understand such programs? If you had to translate into COMAL would you prefer to use the programs or the original specification of the problem?

```
 10 DIM t1$(6), t2$(3,6)
 20 FOR k=1 TO 3:READ t2$(k):NEXT k
 30 FOR s=1 TO 4
 40    PRINT "Dig a corner post hole"
 50    READ s1
 60    REM REPEAT
 70       t3=RND(1)
 80       IF t3<0.5 THEN GOSUB 210
 90       REM treechop
100       s2=RND(1)
110       IF s2>0.5 THEN GOTO 140
120          PRINT "Dig a half metre hole"
130       GOTO  160
140       REM ELSE
150          PRINT "Dig a third metre hole"
160       REM ENDIF
170       s1=s1-15
180    IF s1>=15 THEN GOTO 60
190 NEXT s
200 STOP
210 REM PROC treechop
220    z=INT(RND(1)*3)+1
230    ON z GOSUB 250, 280, 310, 340
240    GOTO  370
250    REM WHEN 1
260       PRINT "Pull out tree."
270    RETURN
280    REM WHEN 2
290       PRINT "Cut above base. Pull out root."
300    RETURN
310    REM WHEN 3
320       PRINT "Cut down tree. Dig out root."
330    RETURN
340    REM OTHERWISE
350       PRINT "Treetype not known."
360    RETURN
370    REM ENDCASE
380 RETURN
390 DATA "small", "medium", "large"
400 DATA 100, 120, 140, 110
```

Fig. 9.0 – Post-holes Program in BASIC.

9.2 Search journals or books for examples of reasonably well-structured programs in BASIC. Try to discern the rules, if any, which the author has followed to achieve reasonable structure.

9.3 In their book, *BASIC Programming* (Wiley, 1967), John Kemney and Thomas Kurtz discuss program structure and style in BASIC. They have an automatic indenting system which depends on two rules.
(1) All IF–THEN statements must jump forward.
(2) Overlapping constructs are not allowed.
The second rule is automatically observed if COMAL is the Program Design Language but how would the first rule alter the structures of this chapter?

9.4 In the same book the following comments appear,

> Organisation has to do with the relationship between parts of a program. Structure concerns the coding of constructs that are used. Another equally important factor in writing clear and readable programs is style. Style is concerned with the use and wording of remarks, the use of blank lines and blank space, the choice of variable names, the use of parameters and the general "flow" of the code.

Do you consider that these comments would be appropriate in a book about structured programming with COMAL? How would you re-write this passage for a COMAL book?

Note
Solutions to this and subsequent sets of problems are not given. Many readers will, by this stage, have found particular directions in which they wish to continue working and experimenting. Problems in this and subsequent chapters are offered as possible areas of investigation.

CHAPTER 10

Sort Processes

There are not many techniques that do not occur somewhere in connection with sorting algorithms.

N. Wirth, *Algorithms + Data Structures = Programs.*

10.1 INTRODUCTION

We can do little more than introduce the subject of sorting. So much has been written and so many techniques have been developed in this area that it is necessary to limit our examination quite severely. Our interest derives only partly from the fact that sorting is a useful process in computing – or in any activity concerned with information processing. The search for various types of sorting methods and attempts to make them more efficient seem to take us to the heart of programming and problem analysis skill development. There are needs for a careful analysis of a problem leading to a particular method, an efficient program design, careful testing and evaluation of results.

Further it seems that not only is there a diversity of approaches to the problem of sorting, but also, in each particular approach there is a progression of algorithms each better than the one before. Generally speaking greater efficiency goes with more sophisticated algorithms but there seems always to be a place for the humbler, simpler ones. They are, in any case, sometimes useful because of their simplicity. In cases where the number of items to be sorted is small they can be faster because the power of their more sophisticated cousins does not outweigh the overhead of a longer algorithm. But they are also valuable in helping a programmer to approach in easy stages algorithms whose effectiveness and ingenuity is unfortunately matched by their difficulty.

Broadly speaking there are two types of sorting: internal sorting of arrays and external sorting of files. We shall concentrate on the former in this chapter. Within the methods of array sorting there are essentially three types: sorting by selection, insertion and exchange of elements. We will discuss a simple case of each and pursue one type to the point of what seems to be near maximum development.

A company may have many customers and find a need to sort into alphabetical order of names. A typical record might consist of:

name, address, representative, amount of orders, amount paid.

The company may wish to sort according to the name of the customer, the name of the company representative, or some other field in the record. The item by which the sorting is guided is called the key. We shall deal with cases where there is only one simple item in each record and it must be the key, but the term key will be used in anticipation of sort operations in which the key is not the whole of the record.

If a set of records were sorted in customer name order and it was required to sort them into county order without spoiling the name order within each county, then a sort method could be chosen accordingly. Such a method would need to be **stable**. In a stable sort, keys with equal value do not shift relative to each other.

The efficiency of sort processes is sometimes discussed according to two measures:

(1) The number of comparisons of two keys
(2) The number of moves of items.

Clearly both these measures increase if the number of items in the sort increases, but for a fixed number of items these measures are good indicators of the relative efficiences of various methods.

10.2 SELECTION SORTS

Perhaps the most obvious method of sorting is to search for the first item, then the next and so on until the array is sorted. Consider the eleven names placed in an eleven element array:

JIM	BEN	ZOE	PAT	VAL	KEN	RON	HAL	LEN	ALF	TOM
1	2	3	4	5	6	7	8	9	10	11

Assuming that we require a sort into ascending order, we can soon identify ALF as the first item. The best thing to do is exchange ALF and JIM. This has two advantages. We need no extra storage outside the array and, since ALF is now correctly located the area of operation is reduced to items 2 to 11. It is true that we have done an exchange in what is described as a *selection* sort but exchanges occur in most types of array sorting. The guiding principle is still *selection.*

METHOD

(1) Select the alphabetically least key (we regard A as less than B because the internal codes have this property).
(2) Exchange this item with that in position 1.
(3) Repeat, starting the search and select process with the second, third . . . tenth items.

PROBLEM ANALYSIS/PROGRAM DESIGN

(1) The names will be READ into array *name*$(11) OF 3 from a DATA statement.
(2) An element will be denoted *start*. This will be 1 on the first run, 2 on the second, up to 10 on the last run.
(3) On each run the *start* item will be noted as *least* and if, as the comparisons occur, a lesser one is noted, *least* will take this value.
(4) At the end of a run the *start* item and the *least* item will be exchanged.

PROGRAM

```
DIM name$(11) OF 3, temp$ OF 3
FOR item := 1 TO 11 DO READ name$(item)
FOR start := 1 TO 10 DO
   least := start
   FOR test := start TO 11 DO
      IF name$(test) < name$(least) THEN least := test
   NEXT test
   EXEC swap
NEXT start
PROC swap
   temp$ := name$(start)
   name$(start) := name$(least)
   name$(least) := temp$
ENDPROC
```

COMMENTS

(1) The procedure *swap* will be used again later. It does not seem worth writing a CLOSED procedure for this simple operation, but the form is worth noting. In order to exchange items a and b we usually need a temporary store to avoid losing data. The standard form is:

```
TEMP := a
a := b
b := TEMP
```

(2) The number of comparisons starting with n items is

```
n – 1 on the first run
n – 2 on the second run
  –
  –
  –
1  on the last run
```

There are (n−1) numbers of average value ½(1 + (n−1)) = ½n.
The number of comparisons is therefore

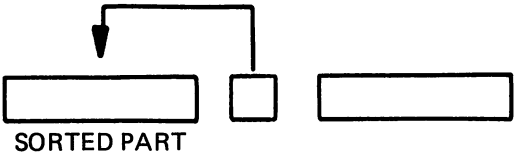
$$(n-1) \times \tfrac{1}{2}n$$
$$= \tfrac{1}{2}(n^2-n)$$

The number of moves varies according to the degree of sorting in the initial position. Its average value is difficult to compute.

(3) The process is stable.
(4) The program has been written for eleven items for initial clarity but it would be very easy to generalise to *num* items.
(5) It is not easy to improve this method substantially but it can be done with a rather complex algorithm known as heapsort.

10.3 INSERTION SORTS

At any time during a simple insertion sort the array is divided into a sorted and unsorted parts. The first item in the unsorted part is then inserted in its correct place in the sorted part which immediately grows by one element



METHOD

(1) There is no point in starting with item 1 because a single item cannot be sorted. In contrast with the selection sort, in which we considered start items from 1 to 10, we now consider items 2 to 11 and place them in their correct positions. We will refer to the item being inserted as *name$(insert)*.
(2) The inserting of the insert item can be done efficiently or inefficiently and it is worth careful consideration. To fix our ideas suppose we have reached the stage shown and we wish to insert KEN in the right place.

BEN	JIM	PAT	VAL	ZOE	KEN	RON	HAL	LEN	ALF	TOM
1	2	3	4	5	6	7	8	9	10	11

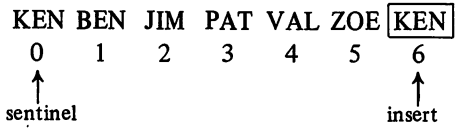
The best method involves moving items to the right, starting with ZOE, until an item is encountered such that the condition for moving fails. The condition for moving might be:

name$(test) > name$(insert)

Unfortunately this would not work when ALF became the insert item because an attempt would be made to compare ALF with something to the left of BEN which does not exist. In fact there are *two* conditions for continuing to move along testing items:

(1) The place for the insert item is not found.
(2) The end of the array is not reached.

This is a well known situation in computing and a trick known as 'posting a sentinel' is used to make sure that the search terminates properly and, at the same time, to reduce the two conditions to one. The item KEN is placed in position *name*$(0) to the left of BEN



Now there is no possibility of any trouble if the search continues to the left end of the array. In order to get the extra item in an array we must declare

```
DIM name$(0:11) OF 3
```

The condition for continuing to search can now be:

```
WHILE name$(test) > sentinel$
(continue searching and moving items to right)
```

PROGRAM

```
DIM name$(0:11) of 3, sentinel$ of 3
FOR k := 1 TO 11 DO READ name$(k)
FOR insert := 2 TO 11 DO
   sentinel$ := name$(insert)
   name$(0) := sentinel$
   test := insert - 1
   WHILE name$(test) > sentinal$ DO
      name$(test+1) := name$(test)
      test := test - 1
   ENDWHILE
   name$(test+1) := sentinel$
NEXT insert
FOR 1 := 1 TO 11 DO PRINT name$(k); " ";
DATA "JIM", "BEN", "ZOE", "PAT", "VAL", "KEN", "RON",
   "HAL", "LEN", "ALF", "TOM"
```

COMMENTS

(1) The variable *sentinel*$ could be avoided by using *name*$(o) instead but the use of *sentinel*$ clarifies the algorithm and would be more efficient on a longer sort because a simple variable can be accessed more quickly than an array variable.
(2) The search for the correct place for each item takes place in a sorted sub-array. A binary search could be used to speed up the process.
(3) The method is stable.
(4) In the worst case the number of comparisons is about the same as in the selection sort but if the data is ordered or partially ordered there is a great improvement. In the best case it comes down to n−1 comparisons. Associated with each run will be at least three moves but this could be reduced to two. There could be as few as 2(n−1) moves. However, the number of moves in this sort increases considerably for data which is not in a favourable condition at the start. Generally it does not perform as well as the selection sort.

10.4 EXCHANGE SORTS

10.4.1 Bubblesort

Although we have used the technique of exchanging two elements of an array in other sort processes, this has been incidental to the essential idea. In this section we are exchanging elements as the major operation in the algorithms. The simplest of these is called the Bubblesort. We start with an array and compare the first two items.

JIM	BEN	ZOE	PAT	VAL	KEN	RON	HAL	LEN	ALF	TOM
1	2	3	4	5	6	7	8	9	10	11

These are exchanged and items 2 and 3 are now JIM and ZOE which are also exchanged. At the end if the first run of 10 comparisons of pairs of adjacent items we have

BEN	JIM	PAT	VAL	KEN	RON	HAL	LEN	ALF	TOM	ZOE
1	2	3	4	5	6	7	8	9	10	11

We see that item 11 must now be in its correct position and we need concern ourselves only with items:

1 to 10 on the second run
1 to 9 on the third run
1 to 8 on the fourth run
—
—
1 to 2 on the tenth run

This leads immediately to the program below, assuming FOR loops for reading and printing data.

```
FOR run := 1 TO 10 DO
   FOR pair := 1 TO 11 – run DO
      IF name$(pair) > name$(pair+1) THEN EXEC swap
   NEXT pair
NEXT run
PROC swap
   temp$ := name$(pair)
   name$(pair) := name$(pair +1)
   name$(pair +1) := temp$
ENDPROC
```

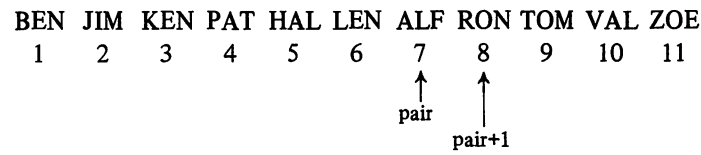
We can see immediately that it may be unnecessary to execute 10 runs if the data is already sorted before the tenth run starts. We could use a flag setting it to zero at the start of each run and to one within the procedure *swap*. If it is zero at the end of a run it means that the data is sorted and we can stop. We will use *en* instead of *end* which is a keyword.

```
en := 11
REPEAT
   FLAG := 0
   en := en – 1
   FOR pair := 1 TO en DO
      IF name$(pair) > name$(pair+1) THEN EXEC swap.
   NEXT pair
UNTIL flag = 0
PROC swap
   flag := 1
   –
   –
   –
ENDPROC
```

We can do even better by noting the point at which the last exchange took place in a run and terminating the next run at that point. This means that we use *flag*, not merely to record 0 or 1 but to record the number of the last unsorted element. For example, on the second run we get to the position:

BEN	JIM	PAT	KEN	RON	HAL	LEN	ALF	TOM	VAL	ZOE
1	2	3	4	5	6	7	8	9	10	11

On the third run, immediately after the last exchange which is RON with ALF we have:



Note that *pair* = 7 which can mark the end of the unsorted data.

Thus we simply write *en* := *pair* in procedure swap instead of *flag* := 1. A few other adjustments produce an improved algorithm.

```
en := 10
REPEAT
   FOR pair := 1 TO en DO
      IF name$(pair) > name$(pair+1) THEN EXEC swap
   NEXT pair
UNTIL en <= 1
PROC swap
   en := pair - 1
   -
   -
   -
ENDPROC
```

The above algorithm still qualifies for the name Bubblesort because, if the data is recorded vertically instead of horizontally the items 'float' upwards into their correct positions like bubbles rising in water. However, the next improvement spoils the metaphor and the process is given a new name.

10.4.2 Shakersort

We note that, in spite of all the improvements to Bubblesort, it is still an asymetric process. ZOE gets to her correct position quickly but ALF does not. The final improvement is to alternate the direction of successive runs so that any extreme cases will not hold up completion of the sort. The name Shakersort reflects this alternating movement of data. The idea is simple but, as is often the case in computing, it requires a careful re-analysis.

METHOD

(1) The variables *left* and *right* will define the start and end of the unsorted part, but it must be remembered that *right* is one short of the end of the true end of the unsorted part. This is because we always think of pairs of elements defined by the left one.

(2) There will be two FOR loops taking the direction of the run alternately right and left.

PROBLEM ANALYSIS/PROGRAM DESIGN

(1) The initial values of *left* and *right* will be 1 and 10.
(2) A FOR loop will read in the names.
(3) A procedure *testswap* will be called from two FOR loops. This will test a pair and swap if necessary.
(4) The main program will be:

```
REPEAT
   FOR pair := left TO right DO EXEC testswap
   right := right -1
   FOR pair := right TO left STEP -1 DO EXEC testswap
   left := left +1
UNTIL left > right
```

PROGRAM

```
DIM name$(11) OF 3, temp$ OF 3
FOR k := 1 TO 11 DO read name$(k)
left := 1; right := 10
REPEAT
   FOR pair := left TO right DO EXEC testswap
   right := right -1
   FOR pair := right TO left STEP -1 DO EXEC testswap
   left := left +1
UNTIL left > right
PROC testswap
   IF name$(pair) > name$(pair +1) THEN
      temp$ := name$(pair)
      name$(pair) := name$(pair+1)
      name$(pair+1) := temp$
   ENDIF
ENDPROC testswap
DATA "JIM", "BEN", "ZOE", "PAT", "VAL", "KEN", "RON",
"HAL", "LEN", "ALF", "TOM"
```

COMMENTS

(1) Further improvements could be made to Shakersort on the lines of the improvements to Bubblesort.
(2) The analysis of the efficiency of Shakersort is very involved but it is not very efficient whatever one does to improve it.
(3) The trouble with Bubblesort and Shakersort is the smallness of each improvement as an exchange is made. This suggests that a method with longer 'jumps' between positions should be sought.

10.4.3 Quicksort

The sorts discussed so far are what might be called the elementary ones. A diligent reader should expect to gain a comprehensive understanding of their workings, if necessary by following through a complete sort using the eleven names. Initial letters would suffice because they are all different.

Quicksort is in a different category. Although it can be seen as an exchange sort, and is in that sense a development of Bubblesort and Shakersort, it is so far in advance of these in concept and performance that it may be a little out of place in a first book on programming. It is included because even a partial comprehension would be rewarding and readers who do not seek full understanding can still use the method to advantage. It is doubtful whether any clearer presentation of the algorithm could be achieved in any language other than COMAL.

As its name implies Quicksort is quick, but it was not so named by its inventor, Tony Hoare, in a spirit of boastfulness. It is simply a fair description of the process. The elementary exchange sorts are least satisfactory and it might be expected that a substantial improvement is possible. It is also reasonable to seek such an improvement by trying to arrange longer jumps when exchanges take place.

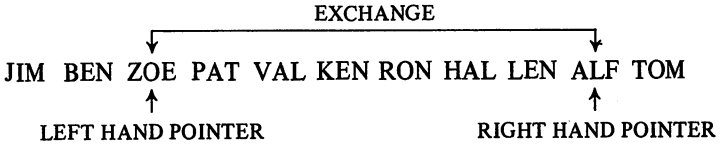
Returning to the standard list of names we try the following algorithm:

(1) Pick any item.
(2) Use this item as a *comparand* so as to *partition* the array:

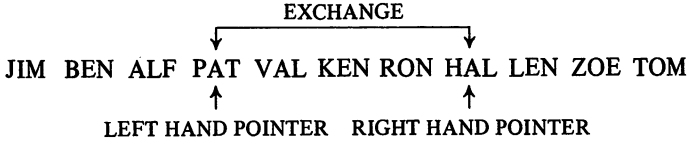
lesser items | comparand | greater items.

(3) Scan from the left until we reach an item greater than comparand.
(4) Scan from the right until we reach an item less than comparand.
(5) Exchange the two items.
(6) Continue scanning and exchanging until the partition is complete.

Choosing KEN as the comparand, the first exchange is between ZOE and ALF.



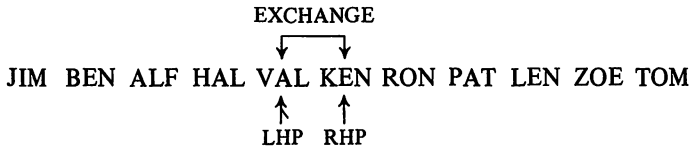
the scans continue:



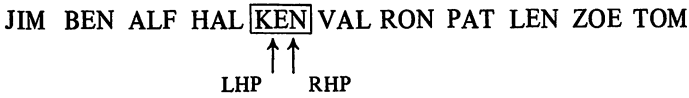
At this point we can see that the left pointer will move on to VAL next but the right pointer would move on to ALF. There is nothing to be gained by letting a pointer move past the comparand so the conditions for movement can be formulated more precisely:

```
WHILE name$(LHP) < comparand$ DO LHP := LHP+1
WHILE name$(RHP) > comparand$ DO RHP := RHP-1
```

This will stop the scan in the following position:



Finally both pointers stick at KEN the comparand which has now moved from its original position.

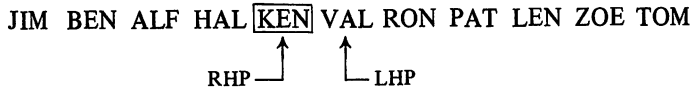


What has been achieved so far is:

(1) KEN is in the correct position.
(2) All items to left of KEN are lesser items.
(3) All items to right of KEN are greater items.

We need only repeat the process on the left hand and right hand sub-arrays and on further sub-divisions until the whole array is sorted. But we need to pay closer attention to the condition for ending a run and for defining the sub-arrays on which the process is to be repeated.

It is tempting to say that we will repeat the process of scan-exchange until the pointers meet, that is, LHP = RHP. However, testing and analysis will show that this condition fails in certain circumstances. It turns out to be necessary to move the pointers on, not only when scanning items, but also when an exchange has been performed. This means that, after VAL and KEN have been exchanged we have:



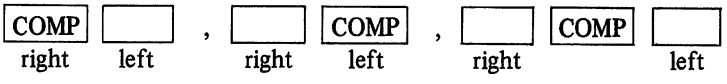
The condition LHP > RHP proves to be an adequate one for terminating each run in a satisfactory manner.

However, in order to guarantee that the pointers will actually cross we must be prepared to swap elements which may be equal. This will happen because

sometimes the pointers will stop at the same element (the comparand) and the exchange will take place of an element with itself. While this seems a waste of time it does force the pointers to move on and terminate the run. Other methods are possible but need careful testing. The algorithm below achieves the first partition.

```
left := 1 ; right := 11
comp$ := name$((left+right)DIV 2)
REPEAT
   WHILE name$(left) < comp$ DO left:=left+1
   WHILE name$(right) > comp$ DO right:=right-1
   IF left <= right THEN EXEC swap
UNTIL left > right
PROC swap
   temp$ := name$(left)
   name$(left) := name$(right)
   name$(right) := temp$
   left := left+1; right := right-1
ENDPROC swap
```

The questions now arise of how to record the first and last number of the sub-arrays and to repeat the partition process. The problem of the sub-array bounds is that there can be three possible types of end position when the pointers have crossed.



Wirth does not attempt to identify the comparand's position but simply accepts *right* as the end of the left sub-array and *left* as the end of the right sub-array. This means that sometimes the sub-arrays are one larger than they need be but the sort is still correct and fast.

We can keep the partitioning processes going by rearranging the algorithm as a procedure and calling it recursively. We shall need two new variables *rend* and *lend* to denote right and left ends of arrays. Thus, at the end of a run the sub-arrays are defined by:

(lend, right)
and (left, rend)

If the sub-arrays consist of at least two elements they need to be sorted and we use the two statements:

```
IF lend < right THEN EXEC quick(lend,right)
IF left < rend THEN EXEC quick(left,rend)
```

The fact that we place these statements *within* procedure *quick* means that the system takes care of all the recording of array bounds representing sub-arrays requiring to be sorted. This is one case where recursion seems natural and easier than other methods. The complete program is given.

PROGRAM QUICKSORT

```
DIM name$(11) OF 3, comp$ OF 3, temp$ OF 3
FOR k := 1 TO 11 DO READ name$(k)
EXEC quick(1,11)
FOR k := 1 TO 11 DO print name$(k); " ";
PROC quick(lend,rend)
  left := lend; right := rend
  comp$ := name$((left+right) DIV 2)
  REPEAT
    WHILE name$(left) < comp$ DO left := left+1
    WHILE name$(right) > comp$ DO right := right-1
    IF left <= right THEN EXEC swap
  UNTIL left > right
  IF lend < right THEN EXEC quick(lend, right)
  IF left < rend THEN EXEC quick(left, rend)
ENDPROC quick
PROC swap
  temp$ := name$(left)
  name$(left) := name$(right)
  name$(right) := temp$
  left := left+1; right := right-1
ENDPROC swap
DATA "JIM", "BEN", "ALF", "PAT", "VAL", "KEN", "RON",
  "HAL", "LEN", "ZOE", "TOM"
```

OUTPUT

The lines below show the state of the array and the values of the left and right pointers after every exchange.

```
JIM BEN ALF HAL VAL KEN RON PAT LEN ZOE TOM  5  7
JIM BEN ALF HAL KEN VAL RON PAT LEN ZOE TOM  6  5
ALF BEN JIM HAL KEN VAL RON PAT LEN ZOE TOM  2  2
ALF BEN HAL JIM KEN VAL RON PAT LEN ZOE TOM  4  3
ALF BEN HAL JIM KEN VAL RON PAT LEN ZOE TOM  3  1
ALF BEN HAL JIM KEN VAL RON PAT LEN ZOE TOM  5  3
ALF BEN HAL JIM KEN LEN RON PAT VAL ZOE TOM  7  8
ALF BEN HAL JIM KEN LEN PAT RON VAL ZOE TOM  8  7
ALF BEN HAL JIM KEN LEN PAT RON VAL ZOE TOM  7  5
```

ALF	BEN	HAL	JIM	KEN	LEN	PAT	RON	TOM	ZOE	VAL	10	10
ALF	BEN	HAL	JIM	KEN	LEN	PAT	RON	TOM	ZOE	VAL	9	7
ALF	BEN	HAL	JIM	KEN	LEN	PAT	RON	TOM	VAL	ZOE	11	10
ALF	BEN	HAL	JIM	KEN	LEN	PAT	RON	TOM	VAL	ZOE	10	8
ALF	BEN	HAL	JIM	KEN	LEN	PAT	RON	TOM	VAL	ZOE	11	9

COMMENTS

(1) The total number of exchanges is seen to be 14.

(2) Even though the partitioning continues after the array is sorted efficiency is still high.

(3) The end of a run is signified by the first number exceeding the second indicating that the pointers have crossed. These pairs are underlined.

CHAPTER 11

Internal Data Structures

> There are many classes of structuring related to computer programming (data structures being a rich enough subject for a separate discussion).
>
> Tom Gilb, *Computer Weekly* 24.9.81.

11.1 THE CONCEPT OF STRUCTURED DATA

If ten names are thrown into a hat in order to decide who shall win a prize the names are deliberately placed in an unstructured situation. All the names have equal status. None bears any special relationship to any other. On the other hand we can have exactly the same names as a list.

JIM BEN ZOE PAT VAL KEN RON ALF TOM LEN

Now we can say that JIM is first or LEN is last or PAT is to the left of RON or TOM follows ALF. Simply by writing the names down in an arbitrary sequence we have added relationships where there were none before. The particular sequence is obtained by writing if our data storage medium is paper, but how could we achieve the same effect if we store data in a computer? The answer is obvious now because we have already used arrays. Placing the names in a string array imposes exactly the same sequential relationships as making a list on paper. If instead we were to store the names in ten ordinary string variables we would not get the same effect.

The concept of a simple array as an ordered sequence of variables is of fundamental importance. It happens to reflect both the requirements of computing problems and the internal design of computer memories and so it has the double advantage of being very useful and easy to organise efficiently internally. It turns out that the efficient solution of problems depends as much on the way data can be structured as on good algorithms. Indeed the two are inseparable. How could one conceive a good sorting algorithm of the kind we have discussed without the concept and facility of an array? As the problems we solve become more realistic and practical so we must spend more time on designing the best data structures for the job. Such designs may be prepared early in the development of a program design and they may be altered or extended as the work proceeds.

Some parts of this chapter will do little more than put into context ideas which have already been used but the simpler ideas will be extended to provide an introduction to the simpler data structures and their use.

11.2 SIMPLE DATA STRUCTURES OF COMAL

11.2.1 Numeric Arrays

Numeric arrays may be *real* or *integer* and must be declared with DIM statement before use. For example

```
DIM whole%(6), table(10)
```

would cause storage to be reserved for six integer elements and ten real elements. In both cases the subscripts would start at 1. However, it is possible to start array subscripts at a number other than one by declaring the first and last subscript values.

DIM whole%(−3:4) reserves eight elements

with subscripts −3,−2,−1,0,1,2,3,4

Numeric arrays may be two dimensional in which case a little care is needed with the subscripts. It is conventional to visualise a two dimensional array as shown in Fig. 11.1.

Suppose we wish to store information about five stock items. It is necessary to declare an array consisting of five rows and three columns,

```
DIM stock(5,3)
```

	STOCK CODE	PRICE	QUANTITY IN STOCK
row 1	123	6.40	78
row 2	349	8.50	82
row 3	496	2.20	300
row 4	608	4.10	132
row 5	813	17.50	35

SUBSCRIPTS

(1,1)	(1,2)	(1,3)
(2.1)	(2,2)	(2,3)
(3,1)	(3,2)	(3,3)
(4,1)	(4,2)	(4,3)
(5,1)	(5,2)	(5,3)

Fig. 11.1 – Stock information.

The fifteen elements so declared must be referenced by two subscripts as shown. Thus we can think of an element referenced as

```
stock(row,column)
```

Having carefully established a suitable data structure for the data we wish to store it is easy to write a short program which places the data in the structure.

```
DIM stock(5,3)
FOR row := 1 TO 5 DO
   FOR column := 1 TO 3 DO
      READ stock(row,column)
   NEXT column
NEXT row
DATA 123,6.40,78,349,8.50,82,496,2.20,300
     608,4.10,132,813,17.50,35
```

A one dimensional array is sometimes called a *vector.* A two dimensional array is sometimes called a *table.*

It is worth noting that the order of numbers in the DATA statement is determined by the need to match the way the algorithm works. If we consider each set of three numbers as a record the form of the DATA is a sequence of records – a structure of widespread application.

Now that the data is stored in a sensible structure we can make use of it. For example the total value of the stock is easily computed. Clearly it is the sum of the products of price and quantity.

```
value := 0
FOR rec := 1 TO 5 DO
   value := value + stock(rec,2)*stock(rec,3)
NEXT rec
PRINT value
```

Example 11.1 The Life Game

The life game is a simplified simulation of the way bacteria might develop from generation to generation. The simulation works on a square grid and there is an initial pattern of life representing the first generation. The grid size can be whatever is convenient. Good patterns can be made with 11 X 11 cells or 15 X 15 but we will show only the inner (9 X 9) part of a grid for reasons of simplicity. The initial pattern may be chosen in a variety of ways but we will start with Fig. 11.2.

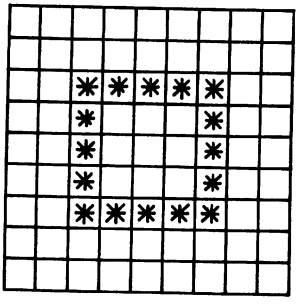


Fig. 11.2 – Starting Life.

Each cell which is not on an outside edge position has eight neighbouring squares which can be marked as North West, North, North East, etc. as in Fig. 11.3. If one of the eight squares is occupied we say that the cell has a neighbour.

NW	N	NE
W		E
SW	S	SE

Fig. 11.3 – Neighbours of a cell.

RULES

(1) A generation is derived from the previous one by deciding which bacteria shall die, which shall survive and in which cells new ones will be born.
(2) The changes from one generation to next occur at the same instant so that only the bacteria of the previous generation determine the pattern of the current one.
(3) A bacterium with one neighbour or none will die of loneliness.
(4) A bacterium with exactly two or three neighbours will survive.
(5) If a cell has exactly three neighbours and is not an edge cell a new bacterium is born in it.
(6) A bacterium with four or more neighbours will die from environmental pollution.
(7) Bacteria cannot be placed or born in cells at the edges of the grid.

Following the rules we can derive the second generation by drawing a circle in every cell which has a surviving bacterium or a newly born one. (Fig. 11.4).

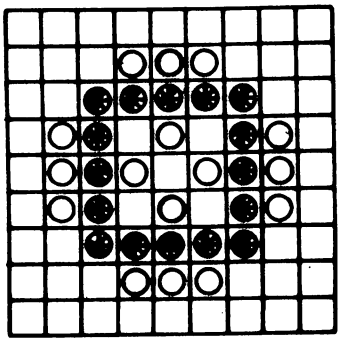


Fig. 11.4 – Second Generation.

There is suddenly a pattern of severe overcrowding and the third generation expands outwards. We show it by first placing stars wherever there was a circle with or without a star in the second generation. Then we draw more circles to show survivors and new bacteria. (Fig. 11.5).

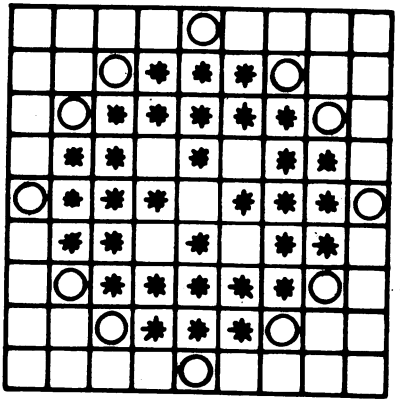


Fig. 11.5 – Third Generation.

It is easy to see that there are no survivors, and no new bacteria after the third generation so the colony ceases to exist.

We have been able to solve problems in a loosely defined framework of method, problem analysis/program design, program, because the problems have involved only the simplest of data structure requirements. Indeed some problems have been so simple that just writing the program also displayed the method, analysis and design. In some cases we have also shown a structure diagram. Now we need to add one more category but, as always, there is no advantage to be gained from trying to write something in all categories for all problems.

Method
Data Structures
Problem Analysis/Program Design
Structure Diagram
Program
Testing
Comments

It will be seen that the choice of appropriate data structures is crucial to the life game program.

METHOD

(1) An array will record each generation of the life cycle of the colony.
(2) The initial state of the grid will be presented as a set of pairs of (row,column) values.
(3) Each generation will be printed as stars though internal storage will be zero for empty and one for life present.
(4) The life cycle will stop when all bacteria have died. It may cycle repetitively through a sequence of states and stop only when interrupted by the operator.

DATA STRUCTURES

(1) DATA statements showing pairs of values will take the form

DATA (sequence of pairs), −1, −1

(2) A 15 X 15 numeric array will store the current generation : *life*(15,15).
(3) A second array, *change*(15,15) is crucial because any change must be based only on previous generation data and must not be influenced by any changes already computed. Thus all changes will be computed and then implemented.

PROBLEM ANALYSIS/PROGRAM DESIGN

(1) The most difficult part of the program is the testing of each cell to make a count of its neighbours. In a 15 X 15 array we need only test the cells with rows 2 TO 14 and columns 2 TO 14 because the edges must remain free of bacteria.
(2) For each of these 13 X 13 = 169 cells we must test all eight neighbours. These are listed relative to a cell life(row,col) in Fig. 11.6.

row−1,col−1	row−1,col	row−1,col+1
row,col−1	row,col	row,col+1
row+1,col−1	row+1,col	row+1,col+1

Fig. 11.6 – Cells for testing.

A neat way to do this is to test all three rows of three columns, making a count of bacteria, then:

(a) IF life(row,col) = 1 THEN count:= count−1
(b) IF count < 2 OR count > 3 THEN change(row,col) := 0
(c) IF count = 3 THEN change(row, col) := 1
(d) IF count = 2 AND life(row,col) = 1 THEN change(row,col) := 1

This anlysis follows the given rules but we can immediately see a more efficient sequence which is logically equivalent. It is important to use the fastest method because it will be used 169 times in each generation. The following is a better alternative.

```
count := 0
FOR c:= col-1 TO col+1 DO
   FOR r := row-1 TO row+1 DO
      count := count+life(r,c)
   NEXT r
NEXT c
count := count-life(row,col)
change(row,col) := 0
IF count=3 OR (count=2 AND life(row,col)=1) THEN
                                        change(row,col) := 1
```

The above sequence will be made into PROC test(row,col)

(3) Move the data from *change* to *life*.

(4) Print using EXEC cursor.

PROGRAM

```
DIM down$ OF 24, right$ OF 80
DIM life(15,15), change(15,15)
FOR k:= 1 TO 24 DO down$(k) := CHR$(17)
FOR k:= 1 TO 80 DO right$(k) := CHR$(29)
FOR row := 1 TO 15 DO
   FOR col := 1 TO 15 DO
      life(row,col) := 0
      change(row,col) := 0
   NEXT col
NEXT row
READ row,col
WHILE row >0 AND col >0 DO
   life(row,col) := 1
   READ(row,col)
ENDWHILE
EXEC printit
REPEAT
   flag := 0
   FOR row := 2 TO 14 DO
         FOR col := 2 TO 14 DO
            EXEC test(row,col)
         NEXT col
      NEXT row
      IF flag := 0 THEN
            PRINT "All dead"
            STOP
      ENDIF
```

```
    FOR row := 1 TO 15 DO
      FOR col := 1 TO 15 DO
        life(row,col) := change(row,col)
      NEXT col
    NEXT row
    EXEC printit
  UNTIL 2=1
  PROC printit
    PRINT CHR$(147)
    FOR row := 1 TO 15 DO
      gridex := 20; gridy := 4
      FOR col := 1 TO 15 DO
        IF life(row,col) = 1 THEN
          EXEC cursor(gridex + col,gridy + row)
          PRINT "*"
        ENDIF
        gridex := gridex + 2
      NEXT col
    NEXT row
  ENDPROC printit
  PROC test(row,col)
    count := 0
    FOR c := col-1 TO col+1 DO
    FOR r := row-1 TO row+1 DO
      count := count+life(r,c)
    NEXT r
  NEXT c
  count := count-life(row,col)
    change(row,col) := 1
    IF count=3 OR (count=2 AND life(row,col) = 1) THEN
      change(row,col) := 1
      flag :=1
    ENDIF
  ENDPROC test
  PROC cursor(ac,dn)
    PRINT CHR$(19)
    PRINT down$(1 : dn-1), right$(1 : ac-1),
  ENDPROC cursor
  DATA 6,6,6,7,6,8,6,9,6,10,7,6,7,10,8,6,8,10,9,6,9,10,10,6,10,7
  DATA 10,6,10,7,10,8,10,9,10,10,-1,-1
```

11.2.2 String Arrays

A string variable in COMAL is a data structure. Essentially it is a character array because any element in the string can be accessed by means of an index. The

notation is extended to give the facility of accessing more than one element because that is how we sometimes wish to handle such things but, in essence, the process is the same as accessing elements of a numeric array by means of subscripts. This can be demonstrated quite easily by writing programs to search or sort the characters in a string. Such programs need differ from their numeric array counterparts only in minor points of syntax. The method and program structure can be identical.

Further, we can break a string variable into various fields to represent various elements of a record (see section 2.2.4).

This provices a very flexible arrangement for storing certain common types of data and the ease with which strings may be handled in COMAL or the better versions of BASIC is almost unrivalled amongst computer languages. Some languages have more facilities for manipulating character data, fields and records and more eaxacting tasks can be performed without much trouble, but it is not easy to find languages which enable the simpler tasks to be done in such simple ways.

If a string variable is really a one dimensional array of characters or fields (sub-strings) then a string array is really a two dimensional structure. However, the two dimensions are not identical in form or facility and the difference is reflected in the string array declaration

```
DIM rec$(10) OF 40
```

which reserves ten string variables each capable of holding 40 characters. Here COMAL differs slightly from the equivalent in BASIC which would be

```
DIM rec$(10,40)
```

Indexing into a string array element is not different in principle from indexing into an ordinary string, but, of course, there must be an indication which array element is involved

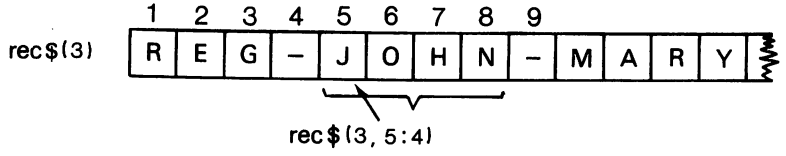


Fig. 11.7 – Indexing a string array element.

Figure 11.7 shows the notation for indexing with a string array element. For example we could write

```
name$ := rec$(3,5:4)
```

which would place "JOHN" in the string *name*$.

The Life game can be a little slow when numeric arrays are used. One way to speed up both computation and plotting on the screen is to use string arrays instead.

Example 11.2
Convert the Life game program (Example 11.1) so that string arrays are used to store the data.

```
PROGRAM
        DIM life$(15) OF 15,change$(15) OF 15
        FOR row:= 1 TO 15 DO
          life$(row,1:15):=" "
          change$(row, 1:15):=" "
        NEXT row
        READ row,col
        WHILE row > 0 AND col > 0 DO
          life$(row,col):= "*"
          READ row,col
        ENDWHILE
        EXEC printit
        REPEAT
          flag:=0
          FOR row:= 2 TO 14 DO
            change$(row,1:15):= " "
            FOR col:=2 TO 14 DO
              EXEC test(row,col)
            NEXT col
          NEXT row
          IF flag=0 THEN
            PRINT "All dead"
            STOP
          ENDIF
          FOR row:= 1 TO 15 DO
            life$(row,1:15):= change$(row, 1:15)
          NEXT row
          EXEC printit
        UNTIL2=1
        PROC printit
          PRINT CHR (147)
          FOR row:= 1 TO 15 DO
            PRINT TAB(20),life$(row, 1:15)
          NEXT row
        ENDPROC printit
```

```
PROC test(row,col)
  count:= 0
  FOR c:=col-1 TO col+1 DO
    IF life$(row-1,c)= "*" THEN count:=count+1
    IF life$(row+1,c)= "*" THEN count:=count+1
  NEXT c
  IF life$(row,col-1)= "*" THEN count:=count+1
  IF life$(row,col+1)= "*" THEN count:=count+1
  IF count=3 OR (count=2 and life$(row,col)="*")THEN
    change$(row,col):= "*"
    flag:=1
  ENDIF
ENDPROC test
DATA 6,6,6,7,6,8,6,9,6,10,7,6,7,10,8,6,8,10
DATA 9,6,9,10,10,6,10,7,10,8,10,9,10,10,-1,-1
```

11.3 SIMPLE DERIVED DATA STRUCTURES

11.3.1 Conceptual Structures

The simple structures provided by COMAL are strings and arrays. (We shall consider files, as more complex COMAL data structures, later). But sometimes a programmer finds it very helpful to use structures which are invented in order to reflect the solutions to certain problems more easily. For example a queue can be represented as an array plus certain rules. The concept of a queue is not a part of COMAL but various types of arrays are suitable structures for the internal representation of the external concept. Of course, a programmer could do the same things to solve a queue problem without mentioning the word queue but it and other concepts occur often enough in problem-solving to be taken seriously and worthy of study in certain more or less standard forms of internal representation and rules for manipulation.

Anything more than an introduction to data structures is beyond the scope of this book but the subject is a big one and a comprehensive understanding of such things is one of the distinguishing features of a good professional programmer or computer scientist.

11.3.2 Queues

A queue is usually represented by an array (see Problem 4.3.5).

Figure 11.8 shows a queue with six items. A variable is needed to mark the front of the queue and either the end or the next empty place must also be marked. If we use *front* and *nep* (next empty place) these variables would have the values 1 and 7.

If an item leaves the queue then

```
front := front+1
```

and if an item joins the queue then

```
queue(nep) := item
```

and

```
nep := nep+1
```

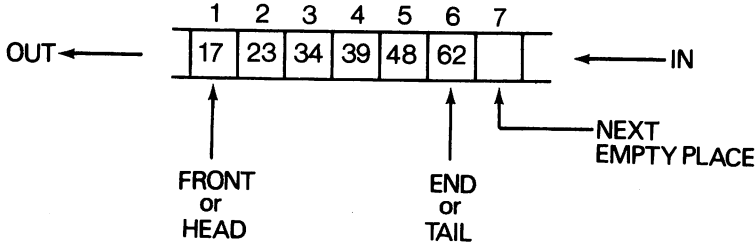


Fig. 11.8 – A queue.

If queue is an array of ten elements we may reach the situation of Fig. 11.9.

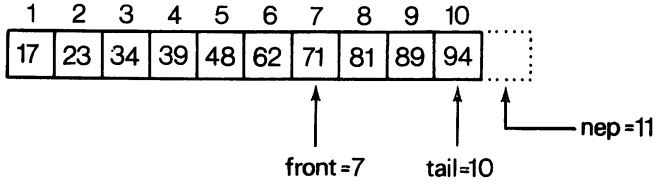


Fig. 11.9 – A queue of four items.

Note that the queue is defined by the pointers *front* and *nep*. The data in elements 1 to 6 is left there but it is no longer part of the queue and it may be overwritten. It can also be seen that the queue has "walked" along the array. One method of dealing with this situation is to move all the items up each time one leaves the queue, but a more economical method is to let the queue walk and reset the printers when they exceed the length of the array. Thus the array becomes, conceptually, a circle. When something leaves the queue we execute

```
front := front+1
IF front > 10 THEN front := 1
```

If something joins the queue we execute

```
queue(nep) := item
```

```
nep := nep+1
IF nep > 10 THEN front := 1
```

The condition for an empty queue is

```
front = nep
```

and the condition for overflow (too many items for the array size) is

```
front = nep
```

If it is necessary to guard against the danger of overflow this can be done by introducing a variable, *length*, which will record the length of the queue. The overflow condition is now

```
length > 10
```

Example 11.3

Write a program to allow a user to add or remove items from a queue. Each item will be a random number in the range 1–100. The array size is six. The user should also be able to request at any time that the queue be listed.

METHOD

(1) A 'menu' will be displayed inviting addition, removal, display or finish.
(2) Messages refusing action will be given if an attempt is made to remove from an empty queue or add to a queue of length six.
(3) If a list of an empty queue is requested the message "Queue empty" will be displayed.

PROGRAM

```
DIM queue(6)
length := 0; front := 1; nep := 1
REPEAT
  PRINT TAB(20) " 1 – Add an item."
  PRINT TAB(20) " 2 – Remove an item."
  PRINT TAB(20) " 3 – List items in queue."
  PRINT TAB(20) " 4 – Finish."
  INPUT choice
  CASE choice OF
  WHEN 1
    EXEC add
  WHEN 2
    EXEC remove
  WHEN 3
    EXEC display
```

```
    WHEN 4
      STOP
    OTHERWISE
      PRINT "Error in input"
    ENDCASE
  UNTIL 2 = 1
  PROC add
    IF length < 6 THEN
      queue(nep) := RND(1,100)
      nep := nep + 1
      IF nep > 6 THEN nep := 1
      length := length + 1
    ELSE
      PRINT "Queue is full."
    ENDIF
  ENDPROC
  PROC remove
    IF length > 0 THEN
      PRINT "Item removed is"; queue(front)
      front := front+1
      IF front > 6 THEN front := 1
      length := length-1
    ELSE
      PRINT "Queue is empty."
    ENDIF
  ENDPROC
  PROC display
    IF length > 0 THEN
      FOR k := front TO front+ length-1 PRINT queue(k MOD 6);
    ELSE
      PRINT "Queue is empty."
    ENDIF
  ENDPROC
```

COMMENT

The above program simulates an actual queue. The numbers stored in the array might represent arrival times or they could be job numbers of certain items placed on one side for later attention. A large computer system would have some method of placing large or low priority jobs in a queue for later execution. However, the representation of the physical queue is not always the best solution to a queue problem. Consider again the questions asked on page 95 about analysing problems and designing programs.

A common use of the queue concept occurs in simulation. Ships queue at ports, aeroplanes queue to land, people queue in supermarkets, patients in a hospital, cars on an assembly line. Sometimes an expensive and time consuming simulation study is made so that better informed decisions can be made about the actual plans. The kind of information required from such studies is average and maximum queue-length, average waiting time for jobs, total 'idle' time for the service and other statistics. There is a branch of mathematics devoted to such ideas, but we are really concerned with choosing the right data structures and programming methods for a computer solution to a typical queue problem.

Example 11.4

A TV repair shop starts an eight hour day (no breaks) with no sets to repair. TV sets arrive in a random manner at an average rate of one every forty minutes. Service times are exponentially distributed with a mean of thirty minutes.

METHOD

One might set up some kind of queue model in which sets arrive and either go into service or join a queue as they would in the real world. An alternative approach is to fix one's attention on the data one needs.

It is easy to compute random arrival times and service times according to the conditions of the problem, but a complete solution requires times of entry into service. These are not so straightforward because the computation is dependent on whether or not the set has had to wait. But once the entry-into-service times are computed, completion times are simply entry-time plus service time.

job = job number
arr(job) = arrival time
ser(job) = service time
ent(job) = entry into service time
comp(job) = job completion time

DATA STRUCTURES

The first two rows of the following arrays are easily computed using any suitable data.

job	1	2	3	4	5	6	7	8	9
arr (job)	37	50	96	108	137	158	187	237	241
ser (job)	32	21	20	28	34	36	38	27	23
ent (job)	37	69	96	116					
comp (job)	69	90	116						

The completion times comp(job) help the argument but they are not essential because it is always true that:

```
comp(job) = ent(job) + ser(job)
```

PROBLEM ANALYSIS/PROGRAM DESIGN

We start the computation for job 1 by writing:

```
ent(1) := arr(1)
```

and `comp(1) := ent(1) + ser(1)`

This avoids difficulties at the start and is valid if one assumes an empty queue and an empty service station as initial conditions.

For the other jobs there are two possibilities:

(1) Arrival *after* completion of previous job as in job 3 above.

(2) Arrival *before* completion of previous job as in jobs 2 and 4 above.

If case (1) applies the job goes straight into service and we can write:

```
IF arr(job) > comp(job–1) THEN
   ent(job) := arr(job)
```

If case (2) applies the job goes into service when the previous job is completed so we write:

```
ELSE
   ent(job) := comp(job–1)
ENDIF
```

The case where arrival time is the same as completion time for the previous job is covered by the ELSE clause, and this seems reasonable. It will also be noticed that data is used from either the top two rows or the previous column so we never try to use data which has not yet been computed.

Once we know the entry time for a job we can compute its completion time so that the routine for job number 2 and subsequent ones could be as follows.

```
FOR job := 2 TO . . . DO
   IF arr(job) > comp(job–1) THEN
      ent(job) := arr(job)
   ELSE
      ent(job) := comp(job–1)
   ENDIF
   comp(job) := ent(job) + ser(job)
NEXT job
```

We can now complete the problem analysis and program design. For simplicity we will deal with exactly twenty jobs.

(1) Set up arrays and initialize.
(2) Compute 20 arrival times.
(3) Compute 20 service times.
(4) Complete column for job 1.
(5) Compute entry and completion time for jobs 2 to 20.
(6) Use the four arrays to calculate the required statistics.

Arrival Times
There is a choice of methods available. We will use the following,

```
arr(1) := –40 * LOG(RND(1))
FOR job := 2 TO 20 DO
   inter := –40*LOG(RND(1))
   arr (job) := arr(job–1) + inter
NEXT job
```

Service Times
This is quite simple.

```
FOR job := 1 TO 20 DO
   ser(job) := –30*LOG(RND(1))
NEXT job
```

Statistics
(1) *Average Queue Length* – Each job either does not wait (arr(job) > comp (job–1)) or its total waiting time in minutes is ent(job) – arr(job). The average queue length is the sum of these quantities divided by the total time under consideration which will be taken to be from t = 1 to t = ent(20).
(2) *Average Waiting Time* – This is the total waiting time as above divided by the number of jobs.

```
PROGRAM
        PRINT "TV REPAIR SIMULATION"
        DIM arr(20), ser(20), ent(20), comp(20)
        arr(1) := –40*LOG(RND(1))
        FOR job := 2 TO 20 DO
           inter := –40*LOG(RND(1))
           arr(job) := arr(job–1) + inter
        NEXT job
        FOR job := 1 TO 20 DO
           ser(job) := –30*LOG(RND(1))
        NEXT job
        ent(1) := arr(1) ; comp(1) := ent(1) + ser(1)
```

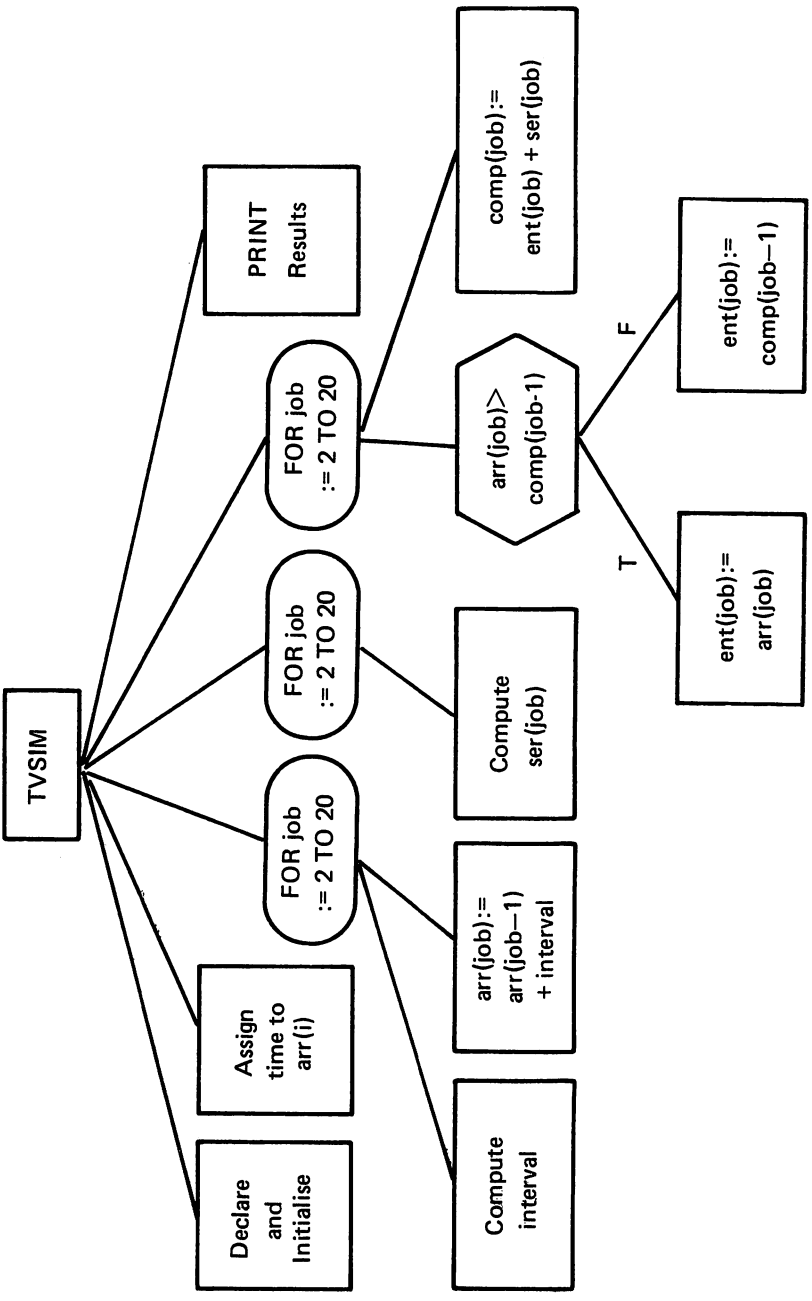


Fig. 11.10 – Structure Diagram for TV Simulation.

```
FOR job := 2 TO 20 DO
   IF arr(job) > comp(job-1) THEN
      ent(job) := arr(job)
   ELSE
      ent(job) := comp(job-1)
   ENDIF
   comp(job) := ent(job) + ser(job)
NEXT job
ZONE := 8
PRINT "Job    Arrival Service Entry   Completion"
FOR job := 1 TO 20 DO
   arr(job) := INT(arr(job) + 0.5)
   ser(job) := INT(ser(job) + 0.5)
   ent(job) := INT(ent(job) + 0.5)
   camp(job) := INT(comp(job) + 0.5)
   PRINT job, arr(job), ser(job), ent(job), comp(job)
NEXT job
```

OUTPUT

TV REPAIR SIMULATION

Job	Arrival	Service	Entry	Completion
1	17	13	17	30
2	63	86	63	149
3	112	10	149	159
4	130	33	159	193
5	225	40	225	265
6	246	9	265	275
7	322	2	322	323
8	373	15	373	388
9	418	50	418	469
10	491	21	491	512
11	518	36	518	554
12	589	64	589	653
13	629	18	653	671
14	678	49	678	726
15	733	27	733	760
16	757	38	760	798
17	845	45	845	890
18	863	2	890	892
19	893	65	893	959
20	939	13	959	972

Queue Statistics

(1) *Average Queue Length* – We can add the segment:

```
IF arr(job)<comp(job-1) THEN sumwt:=sumwt+ent(job)-arr(job)
```

together with initialisation:

```
sumwt := 0
```

This sum needs to be divided by the total number of minutes – approximately given by ent(20).

```
PRINT "Average Queue Length is ";sumwt/ent(20)
```

(2) *Average Waiting Time* – This is the sum of the waiting times divided by the number of sets.

```
PRINT "Average Waiting Time is";sumwt/20
```

COMMENTS

(1) The analysis and program are more straightforward than the direct simulation of physical events.
(2) FOR loops have been used for convenience because they seem to reflect better the 'Event driven' approach. This causes slight difficulty in deciding when the simulation actually ends.
(3) A "clock" could be incorporated and a REPEAT/UNTIL loop could be terminated at a particular time.
(4) The emphasis in this simulation is on intervals between events and data gathering rather than "clock ticking" and queues.
(5) The simplest method for determining random events in time is the testing of each minute. Another method given by K. D. Tocher (*The Art of Simulation*, EUP, 1972.) is the multiplicative method. This is naturally slower than addition and also seems more sensitive to weaknesses in the RND function. The inverse transformation (–M*LOG (RND(1)) is used because it is easy to use, reliable and faster than either of the other two methods. Tocher gives a theoretical justification for it.

The important lesson from the above example is that we should try to establish precisely what information we require and what data structures and methods will enable us to get it in the most straightforward way. Such an approach led to more appropriate data structures and simpler program designs than we would get from a more direct simulation of the physical queue.

11.3.3 Stacks

The fundamental idea of a queue is 'First in – first out' sometimes called FIFO. In contrast the stack follows the rather unfair rule 'Last in – first out' or LIFO. The stack is not very important in introductory work, but it has become a crucial

idea in both hardware and software and it is a simple concept, so it will be treated briefly.

Suppose there is a pile of doormats for sale in a hardware shop. Figure 11.11 shows how the structure works.

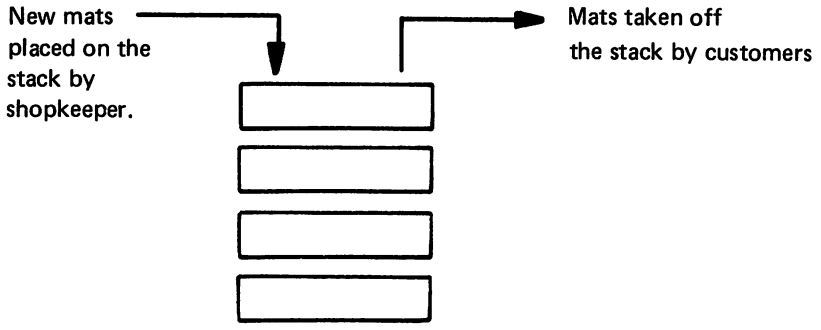


Fig. 11.11 – A stack of doormats.

The stack is like a queue with only one accessible end and it is natural to realise such a concept with an array and a set of rules. The words *pop* and *push* are used as shown in Fig. 11.12.

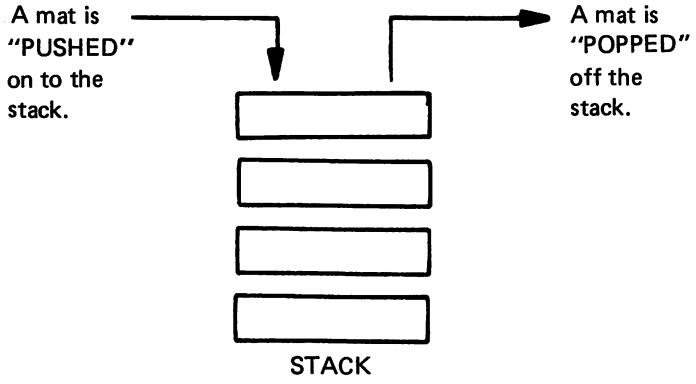


Fig. 11.12.

Example 11.5

Suppose we wish to mark substrings or sub-arrays by storing the left and right indices or subscripts on a stack. Write a program which will 'push' a pair of numbers onto a stack and 'pop' a pair off when requested.

METHOD
The stack will work as shown in Fig. 11.13.

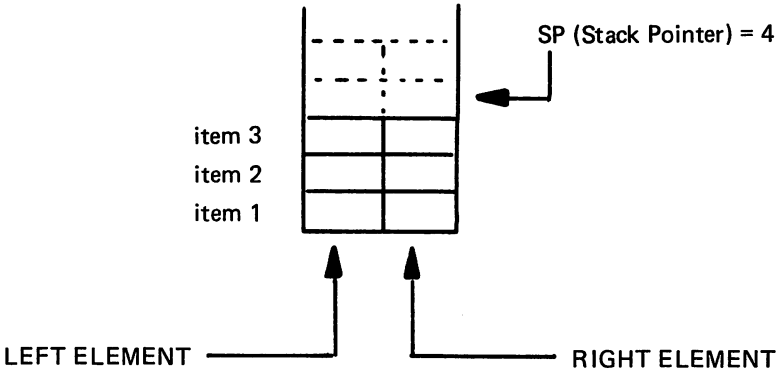


Fig. 11.13.

DATA STRUCTURE
The stack pointer (SP) will always point at the next empty place on the stack which will be a two dimensional array,

stack(20,2)

PROBLEM ANALYSIS/PROGRAM DESIGN
This short program shows the working of a stack in such a way that the user can PUSH pairs of numbers on to it, POP them off, or finish by running out of numbers on the stack. There will be two procedures *push* and *pop*.

```
PROC push
   INPUT "enter two numbers": left,right
   stack(sp,1) := left
   stack(sp,2) := right
   sp := sp+1
ENDPROC
PROC pop
   IF sp = 1 THEN STOP
   sp := sp-1
   left := stack(sp,1)
   right := stack(sp,2)
   PRINT left,right
ENDPROC
```

The two procedures may be called by the main program which will

(1) establish the stack

(2) set the stack printer

(3) accept commands from the user

```
DIM stack(20,2),op$ OF 4
sp := 1
REPEAT
   INPUT "POP or PUSH?": op$
   IF op$ = "POP" THEN EXEC pop
   IF op$ = "PUSH" THEN EXEC push
UNTIL 2=1
```

COMMENT

This stack is used essentially as a storage device. In other uses of stacks the order in which data comes off the stack is crucially important.

11.3.4 Trees

We have met the concept of a tree as a structure diagram. A much more widespread example of the use of the tree (hierarchical) concept is a filing system. A headmaster might organise his files on the lines seen in Fig. 11.14. The 'tree' would be developed further in the actual filing cabinet.

Tree structures are often useful in computer work particularly in more advanced contexts. To discuss data structures without mentioning trees would seem odd and we will examine one illustrative example.

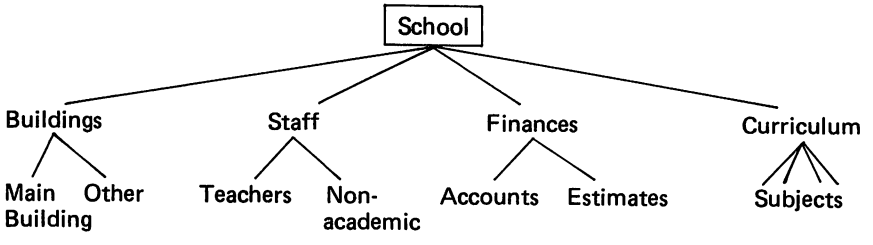


Fig. 11.14 – Files as a tree structure.

Example 11.6

Consider the structure diagram (simplified) which would represent a program segment from the queue problem of Example 11.4. We shall take only the right hand segment (Fig. 11.15). The content of each box has also been simplified but the structure is the same.

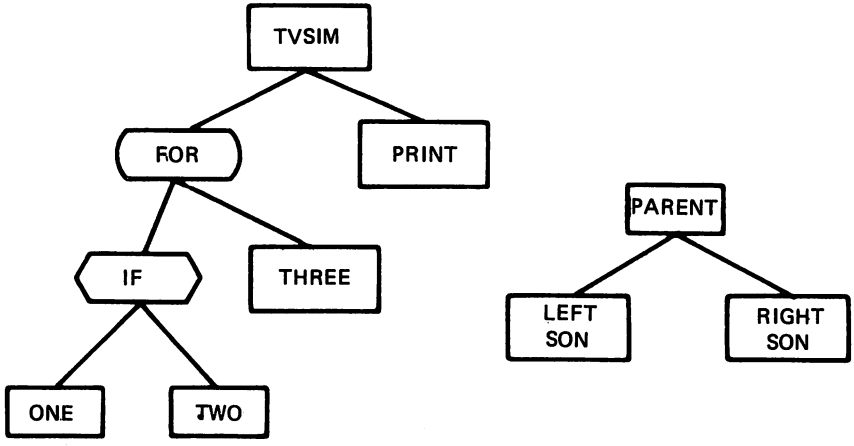


Fig. 11.15(a) – Simplified Program Structure.

This is a **binary tree** because each **node** is either a **terminal node** (or **leaf**) or else it has two **branches.**

We shall also use the terminology **parent, left son** and **right son** in the obvious sense of Fig. 11.15(a). The top node is special and is called the **root.**

It is clear that a tree structure involves more complex relationships between items of data than we have encountered before but while the conceptual structure is very useful externally we cannot translate it directly into an internal data structure. Instead we must represent it as in Fig. 11.15(b) with arrays. We number the nodes arbitrarily except that the root should be 1.

Node	*Data*	*Left Son*	*Right Son*	*Parent*
1	TVSIM	2	7	0
2	FOR	3	4	1
3	IF	5	6	2
4	THREE	0	0	2
5	ONE	0	0	3
6	TWO	0	0	3
7	PRINT	0	0	1

Fig. 11.15(b).

A natural walk around the tree according to the rules of section 6.3 would produce the output:

```
TVSIM
FOR
  IF
    ONE
  ELSE
    TWO
  ENDIF
  THREE
NEXT
PRINT
```

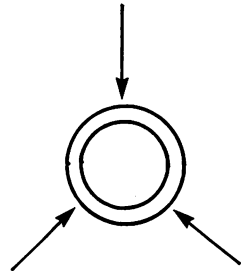
As a further simplification we shall not attempt to provide indentation.

Example 11.6 Tree Walk

Write a program to store the tree of Fig. 11.15(a) and do a natural walk and output the data in a way which reflects the correct program structure.

METHOD

(1) The data will be placed in arrays *info*$, *left*, *right*, *parent*.
(2) A visit counter for each node will be needed to record the number of previous visits. This is because the non-terminal nodes are visited more than once and the action is different on each visit.
(3) The walk will start at the root with *node*=1 and finish at the root when an attempt is made to go to the parent of the root by making *node* = 0.
(4) As the walk proceeds nodes may be approached from three possible directions as shown.



(5) When going *down* the tree the rules are more straightforward. If it is a first visit we simply print the data, update the visit count and decide where to go next.
(6) When going up the tree (using a parent pointer) the rules are more complex and depend on the number of previous visits.

PROBLEM ANALYSIS/PROGRAM DESIGN

(1) Place data in arrays and initialise visit counters.

(2) Start at root node (node = 1) and finish when an attempt is made to go to the parent of the root (node = 0).

(3) IF first visit THEN PRINT data
Increase visit count
IF left son exists AND not visited THEN
go left
ELSE
IF right son exists AND not visited THEN
go right
ELSE
IF parent exists THEN
go to parent
EXEC parentnode
ENDIF
The ELSE-IF combinations will be turned into ELIF.

(4) PROC parentnode
WHEN one previous visit
IF "IF" THEN PRINT "ELSE"
Increase counter
go right
WHEN two previous visits
IF "IF" THEN PRINT "ENDIF"
IF "FOR" THEN PRINT "NEXT"
Increase counter
go to parent
ENDPROC

```
PROGRAM
        DIM info$(7) OF 5, left(7), right(7), parent(7), visit(0:7)
        FOR k:=1 TO 7 DO READ info$(k), left(k), right(k), parent(k)
        FOR k:= 1 TO 7 DO visit(k) := 0
        node := 1
        WHILE node <> 0 DO
          IF visit(node) = 0 THEN PRINT info$(node)
          visit(node) := visit(node) + 1
          IF left(node) <> 0 AND visit(left(node)) = 0 THEN
            node := left(node)
          ELIF right(node) <> 0 AND visit(right(node)) = 0 THEN
            node := right(node)
```

```
    ELIF parent(node) <> 0 THEN
      node := parent(node)
      EXEC parentnode
    ENDIF
  ENDWHILE
  PROC parentnode
    CASE visit(node) OF
    WHEN 1
      IF info$(node) = "IF" THEN PRINT "ELSE"
      visit(node) := visit(node)+1
      node := right(node)
    WHEN 2
      IF info$(node) = "IF" THEN PRINT "ENDIF"
      IF info$(node) = "FOR" THEN PRINT "NEXT"
      visit(node) := visit(node)+1
      node := parent(node)
    ENDCASE
  ENDPROC
  DATA "TVSIM", 2, 7, 0
  DATA "FOR", 3, 4, 1
  DATA "IF", 5, 6, 2
  DATA "THREE", 0, 0, 2
  DATA "ONE", 0, 0, 3
  DATA "TWO", 0, 0, 3
  DATA "PRINT", 0, 0, 1
```

RESULT

The output is:

```
TVSIM
FOR
IF
ONE
ELSE
TWO
ENDIF
THREE
NEXT
PRINT
```

COMMENT

The program is rather difficult for an introductory book. There are simpler tree manipulation programs but this one has clear connections with what has been

done so far. A fairly simple case has been treated but it is not difficult to see how the program could be made more general.

There are more subtle ways of dealing with tree structures, involving the use of stacks and recursion, but this is not the place for more than an introductory example.

PROBLEMS

(1) Write a program which will extract words from text and place them into an alphabetical list, to be output on request.
(2) Alter the TV simulation program so that it computes:
    (a) average waiting time for twenty jobs
    (b) average queue length for twenty jobs
(3) Make a conjecture about possible outcomes of the life game program with various starting conditions and test your expectations.

CHAPTER 12

External Data Structures-Files

If circumstances lead me, I will find
Where truth is hid, though it were hid indeed
Within the centre.

Polonius to Hamlet.

12.1 TYPES OF FILES

The two simpler types of file are those which provide only serial access, rather like the DATA statement or musical items on a cassette, and simple direct access by means of a record number. Just as one moves directly to a chosen track on a long-playing record, without necessarily searching through the preceding tracks first, one can go directly to a record in a direct access file whether its number is one or one thousand.

Various types of external file devices exist, the most common being tapes and discs. Obviously tape-based systems can only provide essentially serial access because it is not possible to get to a record without first winding past its predecessors. At best this will be a fast search process – at worst a very slow one. Magnetic discs, on the other hand, are accessed by a sliding read/write head which can be directed to any track on the disc by the system and its software.

Serious data processing is hardly possible without direct access files and even then it can involve substantial investment in effort and equipment if it is to be successful. Of course, it is possible to demonstrate certain principles with small systems but one principle is that the regular, successful processing of realistically large files requires appreciably higher standards of hardware, software and expertise than a demonstration with only token amounts of specially prepared data.

For these reasons COMAL 80 implementations include direct access files thereby implying a disc-based system or something as powerful. The system software for COMAL 80 is likely to occupy a substantial amount of main memory and this suggests that systems with rather more than 32K bytes of main memory are needed. Some of this may be of the "Read only" (ROM) type but the total should be not too far short of 64K. Even this figure is arbitrary in a sense because it is based on the maximum straightforward addressing capability

of eight-bit microprocessors. As the technology improves, much larger main memories will be available at reasonable cost and they will be very useful.

Just as internal data structures are based on a few fairly simple concepts so it is with files. We shall deal with a few simple cases, but the reader should be aware that there is a great deal more to be learned about file handling. It is not necessary to attempt complex problems involving data retrieval or data bases in large systems to become aware of difficulties in this area. Appreciable problems can arise in seemingly straightforward tasks such as establishing mailing lists, inventories or sorting files which are not large, but too large for the main memory. As usual we shall deal with problems specifically chosen for their comparative ease of solution. The real world is sometimes like that, but mostly it is not.

12.2 FILE CONCEPTS

Record

A record is data in a particular format. It can be anything from a single character or number to a large record giving information about, for example, a book: Author, title, publisher, year, accession number, classification, book number, description, list of keywords of subject matter.

Records can be fixed length or variable length. The former are easier to handle, but the latter are more economical in storage.

File

A file is, in effect, an array of records. If the file is sequential there are constraints on the way the array is accessed. If the file is a direct-access (sometimes called random) file the analogue is very close because just as one can access any array element directly by using a subscript, so one can access any record directly by using the record number.

A file must have a name, preferably meaningful, and there are rules in COMAL about choosing filenames but these are not severe constraints.

Create

Before any file can be used it must be created. This is roughly equivalent to reserving a named space in a filing cabinet so that it may be used when a clerk starts to follow a procedure for putting data into the file or getting it out. It will be useful, conceptually, to think of each file having a separate drawer or box.

Open

If a file exists and is to be used by a program it is necessary to inform the system that the file is going to be used. This is analogous to opening a drawer or box so that a clerk may gain physical access to the records in the file. In the case of COMAL 80, it is necessary to give further information about the way in which the file will be used.

READ from a file

Computers READ data from a file, the smallest unit for reading often being a record. If one small part of a record is required it may be necessary to read the whole record as a unit into suitable internal storage so that the necessary detail can be extracted.

Records may be read serially from a serial or sequential file. There is no choice but to read from the beginning of the file until the desired record is found – even if there are many thousands of records in the file.

A much more flexible type of structure is the direct access or random access file from which any record may be selected directly by using its record number. In practice a name or some other field of a record is known as the identifying feature of a particular record and some intermediate process is often used to convert this into a record number.

There may be different types of fields in a record using different data types. The most commonly used are numeric and string fields.

WRITE to a file

Data is usually written to a file a record at a time. If one part of a record is to be changed the whole record must be assembled, including the amendment, and written to the file as a unit.

In order to write to a particular record of a serial file it is necessary to access all its predecessors first, but there are, in some systems, exceptions to this. The APPEND facility allows the writing of new records which are added to the end of the existing file.

Records may be written to any part of a direct-access file easily and quickly without wasting time searching.

Close

When the use of a file is finished the system must be informed. Certain routine tasks are performed by the system to keep everything in order for future use. This is analogous to closing the box or drawer to protect the file from accidental interference and possibly to enable access to another file.

Catalogue or Directory

A catalogue of all files is kept by the operating system which may be integrated with COMAL or it may be separate. For this purpose any named set of data on a system is called a file. The system should provide various facilities for handling such things as are listed.

SAVE program	Write to a disc file.
LOAD program	Read from a disc file.
ENTER program	Read from a disc file, adding to any program already in main memory.
DELETE program	Remove program from catalogue.

DELETE file	Remove data file from catalogue.
COPY file	Copy from one device to another.
COPY disc	Copy whole disc.
FORMAT disc	Prepare new disc for use.

Such facilities are usually called utility programs or utilities. The first five are incorporated in standard COMAL 80, but the others vary from system to system.

Implementations of Files

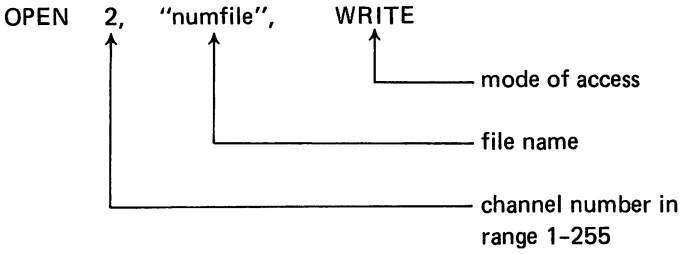
The COMAL 80 proposal did not specify the notation of file systems and there is some variation between implementations. However, the variations are matters of detail. All COMAL 80 implementations up to the time of writing have serial and direct-access files. Records may contain numeric or string data. The file handling routines of the following sections are those of the Commodore PET version of COMAL 80. The style and notation of other implementations (RC Piccolo, Metanic COMAL 80, DDE SPC1, etc.) is similar but manufacturers' documentation should be examined for precise details.

12.3 SERIAL OR SEQUENTIAL FILES

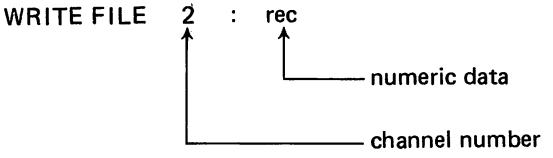
12.3.1 File operations

In order to understand file notation we shall consider two simple files. The first will have a record structure of one real number and the second, of a number and a three-character string.

PET COMAL 80 does not require a separate CREATE statement. When a file is OPENed for the first time it is also created. The following statement will do both jobs. Obviously the first time a file is used it will be opened for writing and this is indicated.



We can place a record in the file by writing.



The statement

```
CLOSE
```

will close the file after operations are finished.

A complete program is shown.

```
OPEN 2, "numfile", WRITE
FOR recnum := 1 TO 10 DO
   rec := RND(1,100)
   WRITE FILE 2 : rec
NEXT recnum
CLOSE
```

The next task is to check that the records are correctly written. The form of the program is very similar.

```
OPEN 2, "numfile", READ
FOR recnum := 1 TO 10 DO
   READ FILE 2 : rec
   PRINT rec;
NEXT recnum
CLOSE
```

String data can be handled in the same way and a record can be constructed according to requirements. The next file will have a record format

```
number , name
```

The two programs below will establish a file, "mixedfile", of ten such records and read back the information as a check.

WRITE PROGRAM

```
DIM name$ OF 3
OPEN 2, "mixedfile", WRITE
FOR recnum := 1 TO 10 DO
   rec := RND(1,100)
   READ name$
   WRITE FILE 2 : rec,name$
NEXT recnum
CLOSE
DATA "JIM", "BEN", "TOM", "ALF", "PAT", "JAN", "SUE",
                                        "TOM", "LEN", "LIZ"
```

READ PROGRAM

```
DIM name$ OF 3
OPEN 2, "mixedfile", READ
FOR recnum := 1 TO 10 DO
   READ FILE 2 : rec, name$
   PRINT rec; name$
NEXT recnum
CLOSE
```

An alternative form of the READ program makes use of the EOF (End-of-file) function. This takes the value TRUE when a sequential file is opened for reading and it remains true until the last record of the file has been read. It then becomes FALSE.

```
DIM name$ OF 3
OPEN 2, "mixedfile", READ
REPEAT
   READ FILE 2 : rec,name$
   PRINT rec; name$
UNTIL EOF(2)
CLOSE
```

12.3.2 Searching a Sequential File

A small file of five records will be set up. The format of each record will be as shown.

Roll number	:	4 digit number
Surname	:	up to 20 characters
Forenames	:	up to 20 characters
Sex	:	one character (M or F)
Birth date	:	six digits in form DDMMYY
Height	:	number giving height in centimetres
Check	:	zero, indicating end of record

The end of the file will be indicated by a separate record containing only –1.

Example 12.1

Set up five records as suggested in DATA statements and transfer the data to a serial file "PERSONS".

Write a second program to search the file and output the roll number and name of all people who are:

(1) male
and (2) born before 1926
and (3) over 170 centimetres in height

METHOD

The programs are simple and self explanatory.

PROGRAM "SETFILE"

```
DIM surnam$ OF 20, fornam$ OF 20,sex$ OF 1,birth$ OF 6
OPEN 2, "PERSONFILE", WRITE
FOR rec := 1 TO 5 DO
   READ roll,surnam$,fornam$,sex$,birth$,hite,check
   WRITE FILE 2 : roll,surnam$,fornam$,sex$,birth$,hite,check
NEXT rec
READ endmark
WRITE FILE 2 : endmark
CLOSE
DATA 0007, "WILLIAMS   ","SHIRLEY     ","F","030429",168,0
DATA 0008, "JENKINS    ","ROY         ","M","210925",171,0
DATA 0009, "STEEL      ","DAVID       ","M","131233",173,0
DATA 0010, "THATCHER   ","MARGARET    ","F","160726",170,0
DATA 0011, "BENN       ","ANTHONY     ","M","110223",177,0
DATA -1
```

PROGRAM "SEARCHFILE"

```
DIM surnam$ OF 20,fornam$ OF 20,sex$ OF 1,birth$ OF 6
OPEN 2, "PERSONFILE",READ
FOR rec := 1 TO 5 DO
   READ FILE 2 : roll,surnam$,fornam$,sex$,birth$,hite,check
   IF check <> 0 THEN
      PRINT "Error in record number"; rec
   ELIF hite > 170 AND sex$="M" AND birth$(5:2)<"26" THEN
      PRINT roll; fornam$;surnam$
   ENDIF
NEXT rec
READ FILE 2 : endmark
IF endmark <> -1 THEN PRINT "End file error."
CLOSE
```

12.3.3 Data Validation

When moderate or substantial quantities of data are processed it is necessary to be confident that the data is correct. Certain errors such as a slightly inaccurate date or a mis-spelt name are not easy to detect but each part (field) of a record can be checked for certain errors. The extent of checking depends on circumstances but in general everything that can be checked should be checked to ensure that a file is 'clean'. For example, character strings can be checked for

length and types of character, digits can be checked, numbers should be in the correct range.

Checking can be done as data is entered ensuring that only correct data goes onto the file. The data might be entered first and validated later. The validation process might merely report errors or it might also correct them.

As an example consider the following numbers which are intended to be six figure birth dates in the form DDMMYY where:

DD is a two-digit day
MM is a two-digit month
YY is a two-digit year

Example 12.2
Test the above data, which is to be checked in every way possible. Report errors by printing the incorrect date with asterisks under the offending pair of digits. Assume the year must be in range 30 – 60 inclusive. If data is non-numeric print "Non-numeric".

METHOD
(1) We must first identify the necessary criteria for correctness
   (a) DD must be in range 01-28 for month 2
                or 01-29 in a leap year
                   01-30 for months 04,06,09,11
                   01-31 for other months
   (b) MM must be in range 01-12
   (c) YY must be in range 30-60
   (d) Data must consist entirely of digits.
(2) Procedure, *numeric*, will test each digit in turn. If the string is not entirely numeric a message will be printed and there will be no further testing.
(3) Procedure, *maxdays*, will determine the correct maximum value for the day depending on the month.
(4) Procedure, *leapyear*, will test the year when the month is February.
(5) Testing will proceed as follows, assuming data is numeric,
   (a) Month in range 01-12
   (b) Day in correct range
   (c) Year in correct range
(6) The date will be printed with asterisks if necessary.

```
PROGRAM "VALIDATE"
        DIM max$ OF 2,d$ OF 2,m$ OF 2,y$ OF 2
        DIM surnam$ OF 20,fornam$ OF 20,sex$ OF 1,birth$ OF 6
        OPEN 2,"PERSONFILE",READ
        FOR rec := 1 TO 5 DO
          READ FILE 2:roll,surnam$,fornam$,sex$,birth$,hite,check
```

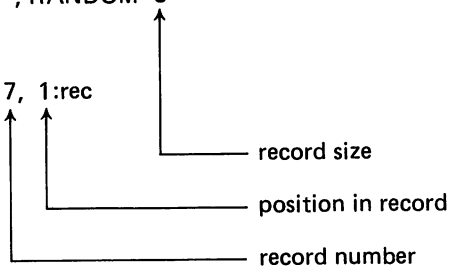
```
    IF check <> 0 THEN
      PRINT "Error in record number";rec
    ELSE
      EXEC numeric
      IF date = FALSE THEN
        PRINT "Non-numeric data or wrong length. Record";rec
      ELSE
        d$:=birth$(1:2);m$:=birth(3:2);y$:=birth$(5:2)
        IF m$ => "01" AND m$ =< "12" THEN
          EXEC maxday
          IF d$ < "01" OR d$ > max$ THEN d$ := "**"
        ELSE
          m$ := "**"
        ENDIF
        IF y$ > "60" OR y$ < "30" THEN y$ := "**"
      ENDIF
      PRINT birth$
      IF birth$ <> d$+m$+y$ THEN PRINT d$+m$+y$
    ENDIF
  NEXT rec
  CLOSE
  PROC numeric
    date := TRUE
    IF LEN(birth$) <> 6 THEN date := FALSE
    FOR k:= 1 TO 6 DO
      ch := ORD(birth$(k:1))
      IF ch < 48 OR ch > 57 THEN date := FALSE
    NEXT k
  ENDPROC
  PROC maxday
    max$ = "00"
    IF m$ IN "04060911" THEN max$ := "30"
    IF m$ = "02" THEN EXEC leapyear
    IF max$ < "28" THEN max$ := "31"
  ENDPROC
  PROC leapyear
    year := ORD(y$(1:1))-48)*10 + ORD(y$(2:1))-48
    IF year/4 = year DIV 4 THEN
      max$ := "29"
    ELSE
      max$ := "28"
    ENDIF
  ENDPROC
```

12.4 DIRECT ACCESS OR RANDOM ACCESS FILES

A file may be created and opened for random access mode. It is not necessary to specify whether reading or writing is intended because, once a random file is properly opened, both are allowed. However, it is necessary to state the record size. The file size is measured in bytes each of which can accommodate one character. Five bytes are necessary for a real number.

The following program will open a random file with a record length of 5 bytes, place a number in record number 7 and read it back as a check.

```
OPEN 2, "ranfile", RANDOM 5
rec := 999.99
WRITE FILE 2, 7, 1:rec
```




```
READ FILE 2, 7, 1:check
PRINT check
CLOSE
```

The "position in record" number, 1, is not necessary when data is being placed in or read from a record starting at the first byte. However, it is possible to alter part of a record in this way (but not two non-contiguous parts). Obviously the user must be careful to specify the record structure carefully and ensure that the correct parts are accessed.

A text file can be treated in a similar way. The following program places ten records, "abcdef", in a random file.

```
DIM rec$ OF 6
OPEN 2, "textfile",RANDOM 6
FOR recnum := 1 TO 10 DO
   rec$ := "abcdef"
   WRITE FILE 2,recnum,1:rec$
NEXT recnum
CLOSE
```

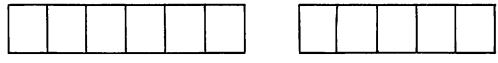
The record size, 6, is the length of text required.

The number, 1, in the WRITE FILE statement indicates that the data is to be placed in the record starting at position 1.

To check that the file is correct we can use the following program,

```
DIM text$ OF 6
OPEN 2,"textfile",RANDOM 6
FOR recnum := 1 TO 10 DO
DIM text$ OF 6
OPEN 2,"textfile",RANDOM 6
FOR recnum := 1 TO 10 DO
   READ FILE 2,recnum,1:text$
   PRINT text$;
NEXT recnum
CLOSE
```

Before proceeding to some more realistic work we should see that it is possible to mix different data types. The following program uses a record structure as shown.



string real number

```
DIM rec$ OF 6,text$ OF 6
OPEN 2,"mixfile",RANDOM 11
FOR recnum := 1 TO 10 DO
   rec$ := "abcdef"
   num := 999
   WRITE FILE 2,recnum,1:rec$,num
   READ FILE 2,recnum,1:text$,check
   PRINT text$;check;
NEXT recnum
CLOSE
```

It is now possible to examine some basic file-handling techniques by means of examples. In order to enable non-trivial amounts of data to be handled without the labour of data entry we shall generate some record data in a random manner. While this data will have no meaning it will serve to demonstrate the principles and techniques without the need to set up large file manually.

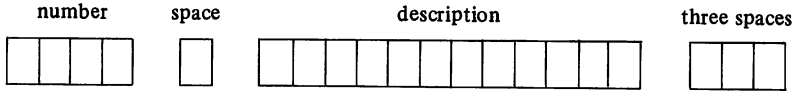
12.5 FILE HANDLING

Example 12.3

Generate a file consisting of one hundred records. The format for a record is

inventory number, description

The number will be a four digit one in the range 0001 – 9999. The description will be a sequence of twelve random upper case letters. Print the file on the screen in the format



METHOD

(1) The records will be generated in the usual way and stored in two arrays. While this is not essential it may be good practice to prepare a block of records and output them rather than deal with the generation and output of records individually.
(2) When the records are ready and stored they will be output to the random access file, INFILE.

PROBLEM ANALYSIS/PROGRAM DESIGN

(1) Since all the above operations are sequential there is no need for any special arrangements at this stage.
(2) The techniques are all straightforward, if detailed, and the program can be written immediately.

PROGRAM

```
DIM num(100), descr$(100) OF 12, word$ OF 12
EXEC fillarrays
OPEN 2,"INFILE",RANDOM 17
FOR r:=1 TO 100 DO WRITE FILE 2,r,1:num(r),descr$(r)
FOR r:=1 TO 100 DO READ FILE 2,r,1:num(r),descr$(r)
CLOSE
FOR r:=1 TO 100 DO PRINT num(r)," ",descr$;"  ";
PROC fillarrays
  FOR k:= 1 TO 100 DO
    num(k) := RND(1,9999)
    word$ := " "
    FOR ch := 1 TO 12 DO word$ := word$+CHR(RND
                                            (193,218))
    descr$(k) := word$
  NEXT k
ENDPROC
```

COMMENT

The file is now well established but it is not sorted in any way and there is no end-of-file marker provided by the user. Nor is any other information, such as

number of records in the file, provided. Such information could be stored at the beginning or end of a file or in an associated small file. For example we could use the last record for file information. "0000" or "ZZZZ" could be placed in the first field as a user's end-marker, and "99" could be placed in the first two places of the second field as a note of the number of records. (The hundredth record would be sacrificed, or the file size increased).

The central concern in using files, after ensuring their correctness, is probably how to access records. For example, we may wish to obtain the record with the inventory number 0734. These numbers are not in any particular order so we can only do a linear search. If we only search for records which exist this means, on average, we search half the records to find the one we want. If some of our searches are for non-existent records the whole file will be searched sometimes and the average search time will be pushed up.

A better solution would be to extract the *search keys,* in this case the inventory numbers, sort them along with the master file record numbers, and use a binary search. Having found an inventory number, we use the associated record number to access the main file directly. (See Fig. 12.1).

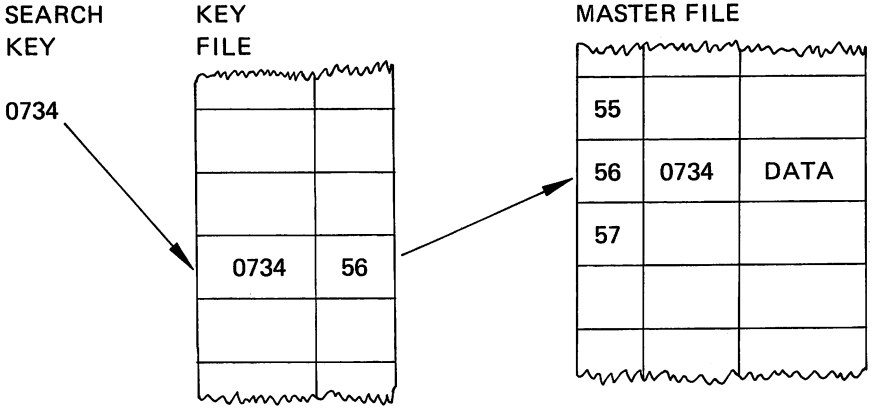


Fig. 12.1 – Binary Search of a Key File and Direct Access.

The advantage of such a method becomes greater as record sizes or file sizes increase. Search times are not affected by the size of master file records and only slightly by increasing the number of records. For example the binary search time is only doubled if the number of records increases from 128 to about 16000.

Example 12.4

Read all the keys (inventory numbers) from the master.file in a separate key file. Use quicksort to sort them into order together with associated record numbers. Check that the key file is properly established by reading it and displaying on the screen.

METHOD

(1) We will construct KEYFILE by reading each key from the master file and associating with it a record number.
For example,

key | 0734 | | 56 | recnum

(2) The ninety nine records will be sorted into key order using Quicksort. Comparisons will be based on the four-digit key but moves will include the complete record of key and recnum. There might be a problem if a key is repeated but this will be ignored if it is only due to the unusual way of generating the records.
(3) A read/print loop will display the sorted keyfile.

PROBLEM ANALYSIS/PROGRAM DESIGN

(1) "INFILE" will be opened and the 99 keys read into array key(100) and the file will be closed.
(2) Quicksort will be a procedure adapted from the program in section 10.4.3. The key and record number will be exchanged when necessary, otherwise the program is only slightly altered.
(3) After sorting the arrays of keys and record numbers will be placed in "KEYFILE".
(4) As a check the data from KEYFILE will be read and printed.

PROGRAM

```
DIM key(100), recno(100)
OPEN 2, "INFILE",RANDOM 17
FOR r:=1 TO 99 DO
   READ FILE 2,r,1:key(r)
   recno(r) := r
NEXT r
CLOSE
EXEC quicksort(1,99)
OPEN 2,"KEYFILE",RANDOM 10
FOR r:= 1 TO 99 DO WRITE FILE 2,r,1:key(r),recno(r)
FOR r:= 1 TO 99 DO READ FILE 2,r,1:key(r),recno(r)
CLOSE
FOR r := 1 TO 99 DO PRINT key(r);recno(r);" ";
PROC quicksort(lend,rend)
   left := lend; right := rend
   comp := key((left+right) DIV 2)
   REPEAT
      WHILE key(left) < comp DO left := left+1
```

```
        WHILE key(right) > comp DO right := right-1
        IF left <= right THEN EXEC swap
    UNTIL left > right
    IF lend < right THEN EXEC quicksort (lend,right)
    IF left < rend THEN EXEC quicksort (left,rend)
ENDPROC
PROC swap
    temp1 := key(left);temp2 := recno(left)
    key(left) := key(right); recno(left) := recno(right)
    key(right) := temp1; recno(right) := temp2
    left := left+1; right:=right-1
ENDPROC
```

COMMENT

The problems of adding and deleting items from the inventory file have not been considered. In practice the addition of items might well be done in such a way as to preserve the order of the key file. The same method might well be used to set up the file, building it, in parallel with a master file. The total computer time of such a method would be greater than the time taken by a method which uses quicksort but a large number of fairly small time 'slices' are usually more acceptable to a computer system than less frequent larger slices for sort operations. Methods which are not the most efficient for a complete sort may be the best in practice for continuously updating a file.

Example 12.5

Write a program which will accept as input four-digit keys. For each key the program will either print the complete record, including the key and the record number, or it will report that the item has not been found. If the user inputs zero the program should halt.

METHOD

(1) A binary search will either return a key and associated number or report "Record not found".
(2) If the key and record number are found direct access will be made to the master file in order to obtain the complete record which will then be displayed.

PROBLEM ANALYSIS/PROGRAM DESIGN

(1) The data from "KEYFILE" will be read into arrays key(100) and recno(100).
(2) "KEYFILE" will be closed and "INFILE" opened.
(3) A correct key will be obtained and the procedure, *binsearch*, will be essentially the program of example 4.7.

PROGRAM

```
DIM key(100),recno(100),descr$ OF 12
OPEN 2,"KEYFILE",RANDOM 10
FOR r:= 1 TO 99 DO READ FILE 2,r,1:key(r),recno(r)
CLOSE
OPEN 2, "INFILE",RANDOM 17
REPEAT
  REPEAT
    INPUT "What is the search key (1-9999)?":seek
  UNTIL seek <= 9999 AND seek > = 0
  IF seek = 0 THEN
    CLOSE
    STOP
  ELSE
    EXEC binsearch(1,99)
  ENDIF
UNTIL 2=1
PROC binsearch(low,top)
  found := FALSE
  REPEAT
    mid := (low+top) DIV 2
    IF key(mid) = seek THEN
      found := TRUE
    ELSE
      IF key(mid) > seek THEN
        top := mid-1
      ELSE
        low := mid+1
      ENDIF
    ENDIF
  UNTIL found OR low > top
  IF found THEN
    READ FILE 2,recno(mid), 1 : num,descr$
    PRINT num; descr$
  ELSE
    PRINT seek; "not found"
  ENDIF
ENDPROC
```

COMMENT

The method works perfectly well for a *static* file which does not change. Deletions would not cause any problem because the record number in the key file could be

made "00" to indicate deletion. The addition of records would need a new routine which would place the complete record in any convenient position in the master file (a new file showing deleted record numbers might be needed). The corresponding key and record number would have to be placed in order in the key file.

It should be clear that file handling, even in relatively simple applications, requires careful planning. The record structures and files need to be planned in detail according to the way they will be used. This implies that all the uses of the files should be well-defined in advance of detailed design of the records and files. But the users of such a system may not be computer people. They may be suspicious of computers. They are very likely to be unaccustomed to specifying their requirements in precise detail, and will be unlikely to know what precision of detail is necessary. In this situation the programmer either gets help or becomes a systems analyst. His job is to persuade the potential user to give the necessary information or help him to define what is necessary. This is a vital pre-requisite for a successful outcome because the final data processing system cannot be better than its specification. If that omits something important there could be a need for time-consuming and embarrassing re-designing of files and programs.

Nevertheless the reader who has read carefully and worked conscientiously through a reasonable proportion of the examples and problems should be in a position to tackle useful projects. He should have acquired some essential techniques and sensible habits in problem analysis and program design. He should also have acquired, or at least be in a position to acquire, rational and realistic attitudes to the whole business of making computers work effectively.

PROBLEMS

11.1 Set up an inventory which will accept items of up to 10 characters (CHAIR, CAR, TABLE, etc.) The system should allow additions, deletions and alterations, as well as search for a particular item.

11.2 Modify the solution to problem 11.1 so that items are added in such a way as to preserve alphabetical order and allow the user to request a printed alphabetical list.

11.3 A mailing file contains names and addresses terminated by the "=" sign. The end of file mark is the "@" sign. Sticky labels are able to accept up to six lines of up to 32 characters. Write a program to validate the file and either:

(a) place a name and address on a new file altering it if necessary to fit the conditions

or (b) if it cannot fit the required format output a message such as

"line count exceeded in record 13"

Bibliography

[1] Abrahams, P. W., Structured Programming Considered Harmful, *SIGPLAN*, **10**, No. 4, April, 1975.

[2] Alagic, S. and Arbib, M. A., *The Design of Well-Structured and Correct Programs*, Springer-Verlag, 1978.

[3] Anderson, R. B., *Proving Programs Correct*, Wiley, 1979.

[4] Atherton, R. (Chairman), *Requirements for a General Purpose High Level Programming Language for Schools*, Working Party of BCS Schools Committee, Computer Education, No. 38, June, 1981.

[5] Atkinson, G., The Non-Desirability of Structured Programming in User Languages, *SIGPLAN*, **12**, No. 7, July, 1977.

[6] Bailey, B., *Data Structures*, Blackie, 1980.

[7] Barron, D. W., *An Introduction to the Study of Programming Languages*, C.U.P. 1977.

[8] Barron, D. W., *Recursive Techniques in Programming*, MacDonald, 1975.

[9] Beech, G., *Successful Software for Small Computers – Structured Programming in BASIC*, Sigma Technical Press, 1980.

[10] Bowles, K., *Problem Solving Using Pascal*, Springer-Verlag, 1977.

[11] Bramer, M., *Adding Structure to BASIC – the Programming Language COMAL 80*, Computer Education No. 37, February, 1981.

[12] Burns, A., *The Microchip as an Appropriate Technology*, Ellis Horwood Ltd., 1981.

[13] Christensen, B. R., *The Programming Language COMAL (Denmark)*, International World of Computer Education, **1**, No. 8, April, 1975.

[14] Colin, A., *Programming and Problem-solving in Algol 68*, Macmillan, 1978.

[15] Dahl, O-J., Dijkstra, E. W. and Hoare, C. A. R., *Structured Programming*, Academic Press, 1972.

[16] Denning, P. J. (Ed.), Special Issue: Programming, *ACM, Computing Surveys* **6**, No. 4, December, 1974.

[17] Dijkstra, E. W., *A Discipline of Programming*, Prentice-Hall, 1976.

[18] Dijkstra, E. W., GOTO Statement Considered Harmful, *Comm. ACM 11*, **3**, March, 1968.
[19] Flores, I., *Computer Sorting*, Prentice Hall, 1969.
[20] Grogono, P., *Programming in Pascal*, Addison Wesley, 1978.
[21] Jensen, K. and Wirth, N., *Pascal User Manual and Report*, Springer-Verlag, 1978.
[22] Kernighan, B. W. and Plauger, P. J., *The Elements of Programming Style*, McGraw-Hill, 1974.
[23] Kieburtz, R. B., *Structured Programming and Problem-Solving with Pascal*, Prentice-Hall, 1978.
[24] McGowan, L. and Kelly, J. B., *Top-Down Structured Programming Techniques*, Van Nostrand, 1975.
[25] Meek, B. L. (Ed.) and Heath, P., *Guide to Good Programming Practice*, Ellis Horwood Ltd., 1981.
[26] Osterby, Tom (Ed.), *COMAL 80 (Nucleus) Definition*, Technical University of Denmark, March, 1980.
[27] Page, E. S. and Wilson, L. B., *Information Representation and Manipulation in a Computer*, C.U.P. 1980.
[28] Penney, G. (Ed.), *Data Processing Case Histories*, NCC, 1974.
[29] Smith, I. C. H. (Ed.), *Microcomputers in Education*, Ellis Horwood Ltd., 1981.
[30] Turski, W. M., *Programming Teaching Techniques*, North Holland/American Elsevier, 1972.
[31] Weinberg, G. M., *The Psychology of Computer Programming*, Van Nostrand, 1971.
[32] Wirth, N., *Algorithms + Data Structures = Programs*, Prentice-Hall, 1976.
[33] Wirth, N., Program Development by Stepwise Refinement, *CACM*, **14**, No. 4, April, 1971.
[34] Wirth, N., *Systematic Programming – An Introduction*, Prentice-Hall, 1973.
[35] Yourdon, E. N., *Classics in Software Engineering*, Yourdon Press, N.Y., 1979.

Appendices

APPENDIX I

PROBLEM GRADING SYSTEM

The following attempt to grade problems is based on their length and structural complexity. Some purely subjective judgements about the relative difficulty of the various structures have been made so the grading is only a rough guide.

Points are added in respect of the level of nesting (two per level) so that, for example, two successive FOR loops would count four points but one nested within the other would get six.

The length of a program does add to its difficulty. For example, string or file manipulations would add to the length of a program irrespective of any particular structures which may be used. However, it is recognised that purely sequential instructions are not usually a cause of serious difficulty and the points rating for length is just the total program length divided by five.

Simple procedures are straightforward in themselves but there is a concealed nesting in the sense that the use of procedures prevents indentations from getting very deep. In recognition of the anlysis which leads to this simplification a simple procedure gets three points split into two for the definition and one for each EXEC statement. Formal parameters count extra.

The system is far from perfect and other factors will need to be considered. Elegant simplifications, notably recursion, get low points in this scheme. But generally the grading of a problem is a reasonable guide, particularly in the earlier and simpler types of problem where algorithmic structures are the dominant theme. Later, as data structures play an increasingly important role the system fails to reflect directly the work necessary to develop the right data structures for a solution. However, there appears to be a correlation between the need for more elaborate data structures and the length and general complexity of a solution to a problem. In any case the system has its greatest value in the early stages when students need to be guided carefully with well-chosen problems.

Clearly it breaks down when programs of hundreds or thousands of lines are written. Such problems are certainly more difficult but not so much more as the grading system would indicate.

POINTS FOR STRUCTURE

Keyword	*Type of structure*	*Points*
FOR	(loop with exit on count)	2
REPEAT	(loop with exit on condition)	2
WHILE	(loop with exit on condition)	3
IF/THEN	(binary decision)	2
IF/THEN/ELSE	(binary decision)	3
ELIF	(nested binary decision)	4
CASE	(multiple decision)	4
PROC	(modularity)	2
EXEC	(procedure call)	1
GOTO	(unstructured control transfer)	4
FORMAL PARAMETERS –	Each	1
NESTING	– For each level Add	2
PROCEDURE	– as a function Add	1

POINTS FOR LENGTH

Total number of lines divided by five, rounded.

Note

The use of system functions or specially supplied procedures will not merit extra points except that the use of EXEC will merit the usual extra point. The use of such things as LEN, ABS will not merit extra points.

APPENDIX II

NOTES ON COMAL SYNTAX

INTRODUCTION

Definitions of computer languages range from the vague and often misleading simple lists of keywords which manufacturers may use in advertisements to long and precise detailed reports which software writers may need. The ordinary user usually needs something in between in order to answer the questions:

Which keyword or usage will enable me to do what I want?
What is the correct syntax?

The material of the various chapters should have answered many such questions. The following pages will probably answer a few more and provide an organised guide. An attempt is made to take account of several definitions or implementations but others are known to be at various stages of development. In all cases the final word must be the manufacturers' documentation for which the following is no substitute.

SYMBOLS USED

This appendix will deal with certain concepts in an informal way. An alternative treatment might be more precise but rather technical. The concepts explained in this section will be used in subsequent definitions.

< >	enclose a syntactic element.
{ }	enclose something which is optional and may occur zero, one or several times.
[ ]	enclose something which is optional and may occur zero times or once.
|	means 'or'
=	means 'is defined as' where the context makes this sensible. The symbol is also part of COMAL.

EXPRESSIONS

The concepts and usage of Boolean and numeric expressions are usually quite different. However, numeric expressions can be used as conditional expressions, as in

```
IF tree THEN EXEC treechop
```

because zero is taken to mean FALSE and any other value is TRUE.

Boolean expressions can also be used in a numerical context because a TRUE Boolean expression is taken as one and a FALSE Boolean expression is taken as zero. Thus

```
code := (value>64)*8
```

would make *code* eight if *value* is greater than sixty-four, otherwise zero.

In order to arrive at sensible definitions we will start with four informal concepts.

⟨integer expression⟩	evaluates to an integer
⟨real expression⟩	evaluates to a real number
⟨Boolean expression⟩	evaluates to 0 or 1 (FALSE or TRUE)
⟨string expression⟩	evaluates to a string

From the previous discussion it can be seen that
⟨numeric expression⟩=⟨integer expression⟩|⟨real expression⟩|⟨Boolean expression⟩
⟨conditional expression⟩=⟨integer expression⟩|⟨real expression⟩|⟨Boolean expression⟩

Although the usage of a numeric expression is usually different from that of a Boolean expression the definitions are identical. For this reason the definitions:

⟨expr⟩=⟨numeric expression⟩|⟨conditional expression⟩
⟨sexpr⟩=⟨string expression⟩

will be used. Although the COMAL 80 proposal does not use this notation the effect is similar.

IDENTIFIERS

The COMAL 80 proposal defines ⟨identifier⟩ as

⟨letter⟩{⟨letter⟩| digit|—}

and specifies that the maximum number of characters should be at least sixteen. There are slight variations between systems but we shall use ⟨name⟩ in the sense of ⟨identifier⟩.

```
          ⟨real variable⟩=⟨name⟩
       ⟨integer variable⟩=⟨name⟩%
                  (or =⟨name⟩#)
        ⟨string variable⟩=⟨name⟩$
        ⟨array variable⟩=⟨variable⟩(⟨expr⟩[,⟨expr⟩])
⟨string array variable⟩=⟨string variable⟩(⟨expr⟩)
     ⟨procedure name⟩=⟨name⟩
              ⟨label⟩=⟨name⟩
```

For definitions of filenames and program names see system manuals.

THE COMAL 80 PROPOSAL (NUCLEUS)

The proposal of March 1980 was intended to define a minimal set of features. It was intended that further recommendations for extending the language would appear later. However, some implementations have already extended the language. Some of these extensions are fundamental and cannot be ignored in any sensible appraisal of COMAL 80. In particular the use of integer variables and expressions are common to the major implementations and the writer regards them as part of COMAL. Apart from these items, what has gone before in this appendix and the rest of this section is consistent with the COMAL 80 Proposal, though the notation has been simplified to make it easier to read.

System Functions

ABS(⟨expr⟩)	Absolute value of⟨expr⟩
ATN(⟨expr⟩)	Angle(Radians) whose tangent is⟨expr⟩
COS(⟨expr⟩)	Cosine of⟨expr⟩measured in radians
EXP(⟨expr⟩)	e to the power of⟨expr⟩
LOG(⟨expr⟩)	The power (log) of e which makes⟨expr⟩
SIN(⟨expr⟩)	Sine of⟨expr⟩ measured in radians
SQR(⟨expr⟩)	Square root of positive⟨expr⟩
TAN(⟨expr⟩)	Tangent of⟨expr⟩measured in radians
INT(⟨expr⟩)	Integer part of⟨expr⟩
ORD(⟨sexpr⟩)	The character code of the first character of the evaluated⟨sexpr⟩
LEN(⟨sexpr⟩)	The length (number of characters) of the evaluated⟨sexpr⟩
CHR(⟨expr⟩)	The ASCII character corresponding to⟨expr⟩

Some systems insert or expect a $ at the end of function names of functions which return string values.

TAB(⟨expr⟩)	used in a PRINT statement for tabulation.

Relational Operators

>	greater than
>=	greater than or equal to
=	equals
<>	not equal to
<=	less than or equal to
<	less than
IN	(not strictly a pure relational operator but often used as such)

Boolean Operators

NOT
AND
OR

Relational operators have equal first priority in conditional expressions and the Boolean operators are in order of priority. Brackets should be used to override these priorities or to remove any doubt.

Arithmetic Operators

The priority of arithmetic operators is

First: monadic+, monadic–
Second: ↑
Third: *,/,DIV(integer division),MOD(modulus)
Fourth: dyadic+,dyadic–

Real Numbers

Real numbers may be expressed as integers, decimals or in exponential form, for example,

21
21.762
2.762431E+6
2.762431E–6

The range of numbers to be fixed by each implementation.

String Variables and Substrings

Simple string variables and string array variables are defined. Substrings are defined as used in this book using the⟨start position⟩and⟨substring length⟩both of which may be numeric expressions. Rounding will be applied if necessary in subscripts used in string variables, substrings or DIM statements.

Simple Statements

The term simple statement is used to define those statements which may be used in 'one-line' structures

⟨simple statements⟩=⟨assignment statement⟩|
⟨io statement⟩|
⟨goto statement⟩|
⟨procedure call statement⟩|
⟨end statement⟩|
⟨stop statement⟩

⟨io statement⟩=⟨read statement⟩|⟨restore statement⟩|
⟨input statement⟩|
⟨print statement⟩|⟨print-using statement⟩
⟨select output statement⟩

⟨restore statement⟩= RESTORE

This resets the pointer for DATA to the first item in the first DATA statement.

⟨print-using statement⟩= PRINT USING⟨sexpr⟩:⟨expr⟩{,⟨expr⟩}[;]

Example

PRINT USING form$:num;
where form$ = "###.##" and num=234.567
would be output as 234.57

⟨end statement⟩ = END

⟨stop statement⟩ = STOP

For precise description of effects see manuals.
Line numbers are in the range 1 to 9999.
Lines may end with a comment which begins with // and is followed by any printable characters (cf. REM of BASIC)

Structure and Control

In this section the syntactic element

⟨statements⟩

will be taken to mean any legitimate statement or block of statements. For example, it may include any properly opened and closed structure or statements in sequence. A precise definition would be rather involved but intuition and common sense are usually reliable guides to what is permissible.

REPETITION

FOR ⟨var⟩:=⟨expr⟩TO⟨expr⟩[STEP⟨expr⟩] DO⟨simple statement⟩
FOR ⟨var⟩:=⟨expr⟩TO⟨expr⟩[STEP⟨expr⟩] DO
  ⟨statements⟩
NEXT⟨var⟩
⟨var⟩=⟨simple numeric variable⟩
REPEAT
  ⟨statements⟩
UNTIL⟨expr⟩
WHILE⟨expr⟩DO⟨simple statement⟩
WHILE⟨expr⟩DO
  ⟨statements⟩
ENDWHILE

DECISIONS (Selection or Choice)

Binary Decisions

IF⟨expr⟩ THEN ⟨simple statement⟩
IF⟨expr⟩ THEN
  ⟨statements⟩
[ELSE
  ⟨statements⟩]
ENDIF

Where

ELSE
IF

occurs

ELIF

may be substituted.

Multiple Decisions

CASE ⟨expr⟩ OF
WHEN ⟨expr⟩{ , ⟨expr⟩}
⟨statements⟩
{WHEN⟨expr⟩{ , ⟨expr⟩}
⟨statements⟩}
[OTHERWISE
⟨statements⟩]
ENDCASE

PROCEDURES AND FUNCTIONS

The precise definition of procedures, functions and parameter handling is not easy to simplify without serious loss of meaning. The section below is quoted directly from the COMAL 80 (nucleus proposal) with a slight change in notation.

Procedure Call

EXEC ⟨procedure name⟩ [(⟨actual parameter list⟩)]

⟨actual parameter list⟩ =
⟨actual parameter⟩ {,⟨actual parameter⟩}

⟨actual parameter⟩ =⟨simple variable⟩|
⟨simple string name⟩|
⟨numeric array name⟩|
⟨string vector name⟩|
⟨arithmetic expression⟩|
⟨string expression⟩

Notes

(1) The number of actual parameters must be the same as the number of formal parameters in the procedure declaration.

(2) The rules for substitution of the formal parameters by actual parameters are the following:

Formal parameter spec.	*Actual parameter allowed*
⟨simple variable⟩	⟨arithmetic expression⟩
REF ⟨simple variable⟩	⟨simple variable⟩
⟨simple string name⟩	⟨string expression⟩
REF ⟨simple string name⟩	⟨simple string name⟩
REF ⟨numeric array name⟩	⟨numeric array name⟩
REF ⟨string vector name⟩	⟨string vector name⟩

(3) An actual parameter which is a ⟨numeric array name⟩ must have the same number of indices as specified for the formal parameter.

Procedure Declaration

⟨procedure declaration⟩ =
⟨procedure head⟩
⟨statement list⟩
⟨ENDPROC part⟩
⟨procedure head⟩ =
PROC ⟨procedure name⟩ [(⟨formal parameter list⟩)]
⟨ENDPROC part⟩ = ENDPROC [⟨procedure name⟩]
⟨procedure name⟩ = ⟨name⟩
⟨formal parameter list⟩=
⟨formal parameter specification⟩
{ , ⟨formal parameter specification⟩}
⟨formal parameter specification⟩ =
[REF] ⟨simple variable⟩|
[REF] ⟨simple string name⟩|
REF ⟨numeric array name⟩ ([,])|
REF ⟨string vector name⟩ ( )

Notes

(1) A procedure declaration specifies that ⟨statement list⟩ is treated as a unit named ⟨procedure name⟩.

(2) A procedure can be activated only by an EXEC statement or by a function call in an arithmetic expression. Return from the procedure occurs when an ENDPROC statement is executed.

(3) Transfer of data between the calling program and the procedure or vice versa can be done using parameters. The transfer of data can also be done using global variables.

(4) Formal parameters can be used in ⟨statement list⟩ as simple or subscripted variables, simple or subscripted string variables, or actual parameters.
Formal parameters used as numeric array names or string vector names must specify the dimension of the array in the following way:
( ) : one-dimensional array
(,) : two-dimensional array

(5) When the procedure is activated, formal parameters in ⟨statement list⟩ will will be assigned the values of (call by value) or replaced by (call by reference) corresponding actual parameters.
Formal specification in the procedure head determines the choice based on the following rules:

⟨formal parameter spec.⟩	*equal to*
⟨simple variable⟩	call by value
REF ⟨simple variable⟩	call by reference
⟨simple string name⟩	call by value
REF ⟨simple string name⟩	call by reference
REF ⟨numeric array name⟩	call by reference
REF ⟨string vector name⟩	call by reference

The REF before ⟨numeric array name⟩ and ⟨string vector name⟩ must be specified to allow for possible future extensions.

(6) Activation of a procedure can take place as a function call in an arithmetic expression. A procedure used in this way should contain at least one assignment statement with the procedure identifier on the left side of ':='.

(7) ⟨procedure identifier⟩ after ENDPROC must be the same as ⟨procedure identifier⟩ after PROC, otherwise an error will occur.

(8) If the procedure was activated by an EXEC statement, execution of an ENDPROC statement will cause execution to continue with the statement after the EXEC statement. If the procedure was activated by a function call, the value of the function will be used in evaluation of the expression in which the call occurred.

(9) Procedures may be called recursively.

IMPLEMENTATIONS OF COMAL 80

The first three implementations of COMAL 80, Metanic, Commodore and RC, extend the procedure concept in important ways to allow CLOSED procedures and GLOBAL variables. Metanic COMAL fails to meet the standard in certain structural features: certain uses of procedures are not possible and the one-line structures such as:

```
FOR k:= 1 TO 100 DO box(k) := 0
```

are not allowed either. This implementation does build cursor control into its system so that

```
CURSOR 10,3
```

could be used instead of

```
EXEC cursor(10,3)
```

There are several more built-in functions, particularly in the area of string handling and data-type conversion. However, the IN and substring notation of COMAL 80 are unifying and quite powerful concepts and perhaps preferable to the jumble of non-standard string-handling functions in many BASICs.

Certain features of the operating environment, while not in the COMAL 80 proposal have been implemented in most COMAL and COMAL 80 systems and

are regarded by the writer as being of substantial value particularly in educational use.

(1) Automatic provision of line numbers.
(2) Acceptance by the system of certain keywords or constructs in incomplete forms (e.g. CHR without $, FOR statement without DO, = instead of :=) and subsequent correct listing of these items.
(3) Automatic indenting.
(4) Forcing of keywords into upper case and other program words into lower case.
(5) Judicious insertion of spaces and deletion of inappropriate spaces.

There is a world of difference between the ease of entry and appearance of listed programs in a system with the above aids, for example, Commodore PET (all except forcing of keywords), RC Piccolo (all), and systems which have merely added some structural keywords without any serious attempt at language design, for example, BBC BASIC. The difference between the systematic design and careful field-testing of a language and the quick assembling of a few selected features is not always apparent at the first glance at a language vocabulary but it becomes apparent within days or weeks of testing. This book could not have been written without access to well implemented versions of COMAL and COMAL 80.

RML SBAS is in special category. Although it is a low-cost enhanced BASIC the designers have been conscious of the need to help the programmer to write and present readable programs. Though it is not COMAL it is a creditable and valuable contribution, see Appendix IV.

NOTES ON METANIC COMAL

Introduction

Like the Commodore and RC implementations, Metanic COMAL considerably extends the original nucleus definition of COMAL 80. This does not alter the uses to which this book can be put in relation to the use of Metanic COMAL except in a few details. Metanic COMAL contains a considerable number of extra features, details of which can be obtained from manufacturers manuals. There are some points of difference which should be mentioned. These represent the most that a reader might have to take account of when using the book with a Metanic COMAL implementation.

Operating Environment

(1) System-forced indentation works with single space increments.
(2) Listing a sub-sequence of a program is done by LIST 80, 150. Similarly for the RENUM (renumber) command.

Data Storage and Manipulation

(1) The user should check to establish whether numeric array elements are set to zero after being declared. Similarly a test should be made on string arrays.

(2) The substring notation in Metanic COMAL is

```
word$(5:8)
```

would indicate four characters from the fifth to the eighth inclusive. The notation

```
word$(5,8)
```

is also allowed and means the same thing. The COMAL 80 standard specifies the former notation and means 'Starting with the fifth, refer to eight characters'.

Output

(1) There is no ZONE statement in Metanic COMAL. TAB must be used instead.

(2) The effect of the semi-colon in a PRINT statement is not consistent with the COMAL 80 standard.

Control

(1) The one-line FOR and WHILE loops specified in COMAL 80 are not allowed in Metanic COMAL.

Examples

(a)
```
FOR k:= 1 TO 100 DO box(k):=0
```
must be written in the extended form
```
FOR k:= 1 TO 100 DO
  box(k):=0
NEXT k
```

(b)
```
WHILE left < right DO left := left+1
```
becomes
```
WHILE left < right DO
  left := left+1
ENDWHILE
```

(2) Functions in COMAL 80 are treated in the same way as procedures except that the procedure name may be used as the parameter carrying a result back to the calling module where it may be used directly in an expression. The definition of a function in Metanic COMAL is specific, for example:

```
DEF FNran
  -
  -
  -
ENDDEF
```

would be the correct notation for the random number function in section 2.4.1.

The function could be used as specified but the variables in a Metanic COMAL function are made local by the system. It would be necessary in this case to declare the variable x as GLOBAL. This must be done inside the function definition. All functions in Metanic COMAL have names which start with FN and the whole name must be used in function calls.

(3) In Metanic COMAL a label for a GOTO statement takes the form:

```
LABEL again
```

Since GOTO statements have not been used in this book, except to show how they work and in the chapter about BASIC, this should not be a difficulty.

Files

Metanic COMAL uses

```
OPEN FILE
```

and

```
CLOSE FILE
```

where COMAL 80 specifies OPEN and CLOSE.

Filenames used in program statements must be in quotes, but in direct commands quotes are not required.

APPENDIX III — ASCII CODES

The characters generated by the COMAL function CHR(N) are given below. N is a denary number.

N	CHR(N) Piccolo	CHR(N) PET	N	CHR(N) Piccolo	CHR(N) PET	N	CHR(N) Piccolo	CHR(N) PET	N	CHR(N) PET
32	SPACE	SPACE	64	@	@	96	\	SPACE	193	A
33	!	!	65	A	a	97	a	!	194	B
34	"	"	66	B	b	98	b	"	195	C
35	£	#	67	C	c	99	c	#	196	D
36	$	$	68	D	d	100	d	$	197	E
37	%	%	69	E	e	101	e	%	198	F
38	&	&	70	F	f	102	f	&	199	G
39	'	'	71	G	g	103	g	'	200	H
40	(	(	72	H	h	104	h	(	201	I
41	)	)	73	I	i	105	i	)	202	J
42	*	*	74	J	j	106	j	*	203	K
43	+	+	75	K	k	107	k	+	204	L
44	,	,	76	L	l	108	l	,	205	M
45	–	–	77	M	m	109	m	–	206	N
46	.	.	78	N	n	110	n	.	207	O
47	/	/	79	O	o	111	o	/	208	P
48	0	0	80	P	p	112	p	0	209	Q
49	1	1	81	Q	q	113	q	1	210	R
50	2	2	82	R	r	114	r	2	211	S
51	3	3	83	S	s	115	s	3	212	T
52	4	4	84	T	t	116	t	4	213	U
53	5	5	85	U	u	117	u	5	214	V
54	6	6	86	V	v	118	v	6	215	W
55	7	7	87	W	w	119	w	7	216	X
56	8	8	88	X	x	120	x	8	217	Y
57	9	9	89	Y	y	121	y	9	218	Z
58	:	:	90	Z	z	122	z	:		
59	;	;	91	[	[	123	{	;		
60	<	<	92	\	\	124	:	<		
61	=	=	93	]	]	125	}	=		
62	>	>	94	↑	↑	126	∿	>		
63	?	?	95	_	←	127	■	?		

Notes

(1) The codes 0 to 32 are used for various purposes in different systems.

(2) Codes in the range 128-255 are used for graphics and other purposes which vary between systems.

APPENDIX IV

NOTES ON SBAS–STRUCTURED BASIC FOR RML 380Z

INTRODUCTION

SBAS is a modified version of BASIC produced by Software Production Associates of Leamington Spa, England. It is a positive response to the fact that some thousands of 380Z systems exist in British schools. Many of these are cassette based and do not support the CP/M operating system under which Metanic COMAL runs. SBAS will run under CP/M or COS 3.0 and subsequent RML monitors.

SBAS has the main structures of COMAL and at the time of writing serious efforts are being made to bring a second version closer to COMAL in notation as well as in spirit. Users should check with the documentation provided to ascertain precisely what features are available. The following notes cannot replace suppliers documentation which should cover RML BASIC version 5.0 and the SBAS extensions.

OPERATING ENVIRONMENT

The user may choose the degree of indenting. The system will enforce the user's choice. The default value is one space. Rules for indenting are slightly different and are not adjusted for the number of digits in a line number.

Keywords will not be automatically forced into upper case nor will other words be forced into lower case but the system will not alter what the user does. The user of a special command CAPS will cause system-forced lower case, for keywords, and upper case for other words.

Automatic line numbering is not provided.

DATA STORAGE AND MANIPULATION

Long variable names for variables, labels and procedures are allowed.

The rules for using string variables, arrays and substrings are those of RML BASIC. The appropriate manual should be consulted.

INPUT/OUTPUT

The rules for I/O are the rules of RML BASIC.

CONTROL

The following structures are allowed and follow COMAL closely:

(1)
```
FOR
  –
  –
  –
NEXT
```

(2)
```
REPEAT
  –
  –
  –
UNTIL ⟨condition⟩
```

(3)
```
WHILE ⟨condition⟩ DO
  –
  –
  –
ENDWHILE
```

(4)
```
CASE ⟨expression⟩ OF
  WHEN ⟨list of values⟩
    –
    –
    –
  OTHERWISE
    –
    –
    –
ENDCASE
```

In the CASE construction extra features are allowed and an alternative notation is possible in this and other structures. However, there is much to be gained from following the COMAL standard.

In order to preserve compatibility with RML BASIC the structure for a binary decision has a slightly different notation:

```
IF ⟨condition⟩ THEN BEGIN                    (one line)
  –
  –
  –
END
ELSE BEGIN                                   (one line)
  –
  –
  –
END
```

The ELIF construction is not allowed but ranges of values, as well as lists of values, can be dealt with in a CASE statement.

```
CASE v OF
  WHEN >1
    -
    -
  WHEN <3
    -
    -
    -
ENDCASE
```

Procedures may be defined by PROC . . . ENDPROC and are called by stating the procedure name

```
printit
```

rather than

```
EXEC printit
```

Rules for the scope of variables and parameter passing are different from COMAL. The manual should be consulted. The main rule is that procedures are normally closed by the system. Global variables are allowed.

Labels are allowed as in COMAL but a colon is required at the target label.

```
GOTO start
  -
  -
  -
  -
start:
```

FILES

Direct Access files are not available in BASIC 5 and therefore not in SBAS.

Solutions to Problems

CHAPTER 1 (page 13)

1.1 susan
24

1.2 susan
jane
jane susan

1.3 40 3 120

1.4 10

1.5 8 Q or other combination of digit, letter.

1.6 A string variable may be allocated any amount of storage from sufficient for one character to whatever the system may allow, for example 255 or 4095. The system needs to know what the programmer requires so that sufficient, but not a wasteful amount of storage can be reserved. Numeric variables are allocated storage in a consistent way and the programmer cannot influence this.

1.7 A previously used program may be in the memory. If NEW is not typed the new program will be treated as amendments to the existing one which is likely to produce unwanted effects.

1.8 (a) DEL LIST AUTO RENUM EDIT
(b) RUN CON
(c) PRINT DIM OF

1.9 Without RANDOMIZE at the beginning of a program any sequence of random numbers generated will be the same each time the program is executed.

1.10 The form:

⟨variable⟩ ← ⟨expression⟩

suggests that the right hand side must receive attention first, the result going into the variable on the left hand side.

Taking things a stage further and recognising that we normally read from left to right an even more sensible notation might be:

⟨expression⟩ → ⟨variable⟩

1.11 (a) If the AUTO command is used the user need not bother about line number provision any further. A line number deliberately chosen out of sequence will cause the associated line to be inserted according to the chosen line number. Line numbers may be edited thus enabling lines to be moved to new positions. Lines may be renumbered using the RENUM command.

(b) The user may type an assignment statement using = only. The system will change it to :=. The system will not accept a line which attempts an invalid assignment. For example

```
area := "susan"
```

would be rejected with a syntax error.

(c) When indenting becomes appropriate with more complex programs the user need not provide it. After entering a program, the command LIST will show all necessary indenting.

CHAPTER 2 (page 25)

2.1 The data is not helpful unless the helmsman knows how to interpret it. He must known whether the raised right arm means 'Steer to the right' or 'there is a rock on the right.' or possibly something else.

2.2

(a)	temperature	18.5
	position	4
(b)	count%	3
	k%	2
(c)	soft	1 (interpreted as TRUE)
	found	0 (interpreted as FALSE)
(d)	text$	"Any characters"
	char$	"A"

2.3 Integer variables occupy less computer storage, are not susceptible to rounding errors and are processed faster.

2.4 '+' can mean 'add' or 'join' according to the context.

2.5
```
plant$  := test$(3:4)
meat$   := text$(14:9)
DIM text$ OF 23, plant$ OF 4, meat$ OF 9
```

2.6 (a) syntax error : *total* is a numeric variable

(b) *total*$ will take the value "6550".

2.7 12 56 68 9 5

2.8 (a) TRUE (b) TRUE (c) FALSE (d) NEITHER

2.9 (a) TRUE (b) FALSE (c) FALSE (d) TRUE
(e) FALSE (f) TRUE (g) FALSE (h) TRUE
(i) NEITHER (j) TRUE

2.10 As seen by a human being a computer program is static text on paper or screen but it represents a process which is a dynamic sequence of probably many more operations than there are statements in the program. It is not easy to establish clear association between two such different things.

2.11 Twenty 20

2.12 An INPUT statement causes the program to pause in order to allow it to accept data from the keyboard.

CHAPTER 3 (page 40)

3.1 The important point here is that in a well-implemented system the statement:

```
FOR post := 1 TO 0 DO
```

should cause no executions of the content of the FOR loop. It will be seen later that a programmer sometimes finds it necessary to provide in repetitive sequences the possibility of one or zero executions.

3.2
```
FOR half := 1 TO 2 DO
   PRINT "Half a game."
NEXT half
```

3.3 The results can be tabulated by omitting the final semicolon in the PRINT statement.

3.4
```
FOR down := 10 TO 14 DO
   FOR across := 30 TO 49 DO
      EXEC cursor(across,down)
      PRINT "*"
   NEXT across
NEXT down
```

3.5
```
FOR across := 30 TO 49 DO
   FOR down := 10 TO 14 DO
      EXEC cursor(across,down)
      PRINT "*"
   NEXT down
NEXT across
```

In version 1 the first FOR statement is executed once and the second FOR statement is executed 5 times making a total of six. Each time a FOR statement is executed a variable is established and initialised, a final value is

noted. This work is done 1 + 5 = 6 times in version 1 but 1 + 20 times in version two. Such considerations can sometimes make a significant difference in speed of execution.

3.6
```
FOR code := 65 TO 90 DO
   PRINT CHR(code);
NEXT code
```
Note. In PET COMAL the codes 65,90 change to 193,218 for upper case letters.

3.7
```
num:= 1
FOR k := 1 TO 12 DO
   num := num*2
NEXT k
PRINT num
```

3.8 METHOD 1.
```
bact := 10
FOR hour := 1 TO 12 DO
   bact := bact*4
NEXT hour
PRINT bact
```
METHOD 2.
```
bact := 10
PRINT bact * 4 ↑ 12
```

3.9
```
RANDOMIZE
REPEAT
   die1 := RND(1,6)
   die2 := RND(1,6)
   PRINT die1 + die2
UNTIL die1 = die2
```

3.10
```
RANDOMIZE
DIM char$ OF 1
REPEAT
   code := RND(65,90)
   char$ := CHR(code)
   PRINT char$
UNTIL char$ = "Z"
```

3.11
```
RANDOMIZE
DIM char$ OF 1
count := 0
REPEAT
   code := RND(65,90)
```

```
    char$ := CHR(code)
    PRINT char$;
    count := count + 1
UNTIL char$ = "Z"
PRINT count
```

3.12

```
count := 0
million := 1000000
num := 1
REPEAT
   num := num*2
   count := count +1
UNTIL num > million
PRINT count
```

3.13

```
extra := 0
tri := 0
REPEAT
   extra := extra +1
   tri := tri + extra
   PRINT tri
UNTIL tri > 1000
```

3.14

```
sum := 0
term := 0
REPEAT
   term := term +1
   sum := sum + 1/term
UNTIL sum > 6
PRINT term
```

3.15

```
RANDOMIZE
DIM word$ OF 3, ch1$ OF 1, ch2$ OF 1, ch3$ OF 1
REPEAT
   ch1$ := CHR(RND(65,67))
   ch2$ := CHR(RND(65,67))
   ch3$ := CHR(RND(65,67))
   word$ := ch1$ + ch2$ + ch3$
   PRINT word$; " ";
UNTIL word$ = "CAB"
```

3.16

```
count := 0 after DIM in solution 3.15
count := count +1 after PRINT
PRINT count at end.
```

3.17
```
year := 1979
oldbats := 840
upstart := 250
REPEAT
   oldbats := oldbats – 40
   upstart := upstart +50
   year := year +1
UNTIL upstart > oldbats
PRINT year; upstart; oldbats
```

3.18
```
old := 4000
previous := 5000
PRINT old; previous
FOR gen := 3 TO 10 DO
   newval := previous + old – 2000
   PRINT gen; newval
   old := previous
   previous := newval
NEXT gen
```

3.19
```
DIM word$ OF 5, char$ OF 1
INPUT word$
num := 0
FOR ch := 1 TO LEN(word$) DO
   char$ := word$ (ch:1)
   newval := ORD(char$) – 48
   num := num*10 + newval
NEXT ch
PRINT num
```

3.20
```
FOR hour := 7 TO 11 DO
   FOR bus := 1 TO 4 DO
      PRINT hour; "."; 15* (bus–1)
   NEXT bus
NEXT hour
```

3.21 METHOD

It is necessary to recognise that the first star on each line is always at across position 41-line and the final star on a line is always at across position 39+line.

```
PROGRAM
      FOR line := 1 TO 5 DO
         down := line + 4
         FOR across := 41-line TO 39 + line DO
```

```
            EXEC cursor (across,down)
            PRINT "*"
         NEXT across
      NEXT line
```

3.22
```
      sum := 0
      FOR ex := 1 TO 20 DO
         count := 0
         REPEAT
            die := RND(1,6)
            count := count + 1
         UNTIL die = 6
         sum := sum + count
      NEXT ex
      average := sum/20
      PRINT average
```

3.23
```
      sum1 := 1000
      sum2 := 1000
      FOR year := 1 TO 7 DO
         sum1 := sum1 * 1.11
         FOR quarter := 1 TO 4 DO
            sum2:= sum2 * 1.025
         NEXT quarter
         PRINT year; sum1; sum2
      NEXT year
```

3.24 If the original use of count is to be preserved we need another variable, sum, to accumulate the sum of each separate count from each experiment.

```
      RANDOMIZE
      DIM char$ OF 1
      sum := 0
      FOR ex := 1 TO 25 DO
         count := 0
         REPEAT
            code := RND(65,90)
            char$ := CHR(code)
            PRINT char$
            count := count + 1
         UNTIL char$ = "Z"
         PRINT count
         sum := sum + count
      NEXT ex
      average := sum/25
      PRINT average
```

3.25
```
RANDOMIZE
DIM word$ OF 3, ch1$ OF 1, ch2$ OF 1, ch3$ OF 1
sum := 0
FOR ex := 1 TO 30 DO
   count := 0
   REPEAT
      ch1$ := CHR(RND(65,67))
      ch2$ := CHR(RND(65,67))
      ch3$ := CHR(RND(65,67))
      word$ := ch1$ + ch2$ + ch3$
      PRINT word$; " ";
      count := count + 1
   UNTIL word$ = "CAB"
   sum := sum + count
NEXT ex
average := sum/30
PRINT average
```

CHAPTER 4 (page 58)

4.3
```
max := 0
FOR count := 1 TO 3 DO
   READ num
   IF num > max THEN max := num
NEXT count
PRINT "Largest number is "; max
DATA 13, 9, 17
```

4.4
```
DIM name$ OF 3, first$ OF 3
first$ := "ZZZ"
FOR count := 1 TO 5 DO
   READ name$
   IF name$ < first$ THEN first$ := name$
NEXT count
PRINT "The first name is "; first$
DATA "PAT", "HAL", "ALF", "BEN", "ZOE"
```

4.5
```
min := 999
max := 0
FOR count := 1 TO 30 DO
   num := RND(1,100)
   PRINT num;
   IF num > max THEN max := num
   IF num < min THEN min := num
```

```
NEXT count
PRINT "Range of numbers is "; min; "to"; max
```

4.6
```
good := 0
FOR count := 1 TO 400 DO
   IF RND(1,100) > 60 THEN good := good + 1
NEXT count
PRINT "Number of good chips is "; good
```

4.7
```
FOR worker := 1 TO 6 DO
   READ wage, prod
   IF prod > 40 THEN wage := wage +20
   PRINT "Worker number "; worker; wage
NEXT worker
DATA 120, 42, 132, 46, 125, 31, 130, 36, 120, 44, 140, 52
```

4.8
```
DIM message$ OF 7
FOR rail := 1 TO 10 DO
   message$ := "OK"
   READ length
   IF length < 4.997 OR length > 5.003 THEN message$ :=
                                                    "Push off."
   PRINT message$
NEXT rail
DATA 5.0017, 5.0042, 4.9978, 4.9991, 5.0005
DATA 5.0021, 4.9983, 4.9997, 4.9970, 5.0030
```

4.9
```
PRINT "No Bathing"
READ first, second, third
PRINT "No Bathing"
PRINT "No Bathing"
REPEAT
   average := (first + second + third)/3
   IF average > 25 THEN
      PRINT "No Bathing."
   ELSE
      PRINT "Bathing Allowed."
   ENDIF
   first := second
   second := third
   READ third
UNTIL third < 0
DATA 13, 15, 26, 24, 28, 30, 27, 22, 21, 19, 17, 14, -1
```

4.10
```
max := 0
FOR triangle := 1 TO 6 DO
```

```
    READ opp,adj
    hyp := SQR (opp↑2 + adj↑2)
    IF hyp > max THEN max := hyp
  NEXT triangle
  DATA 1,6,2,5,3,4,4,4,3,5,2,6
```

4.11 METHOD

(1) For each of the fifteen possible routes the distance from London to point of joining is *route* * 10. Call this *join.*

(2) The distance across country (*leg*1) is

```
leg1 := SQR(1600 + join ↑ 2)
```

(3) The distance along the motorway is 200 – join. (leg2)

(4) Time for a journey is distance/speed.

```
PROGRAM
    min := 999; minroute := 0
    FOR route := 1 TO 15 DO
      join := route *10
      leg1 := SQR(1600 + join↑2)
      leg2 := 200 – join
      tim := leg1/45 + leg2/60
      PRINT "Time for route "; route; "is" ; tim
      IF tim < min THEN
        min := tim
        minroute := route
      ENDIF
    NEXT route
    PRINT "Route is "– route; "Time is "; tim
```

4.12

```
    DIM letter$(2) OF 1, temp$ OF 1
    FOR k := 1 TO 2 DO letter$(k) := CHR(RND(65,90))
    IF letter$(1) > letter$(2) THEN
      temp$ := letter$(1)
      letter$(1) := letter$(2)
      letter$(2) := temp$
    ENDIF
    PRINT letter$(1); letter$(2)
```

4.13

```
    DIM area$ OF 1
    INPUT "What is the zone ?": area$
    INPUT "What is the weight ?": weight
    extras := weight DIV 10
    CASE area$ OF
```

```
WHEN "A"
   charge := 20 + extras*11
WHEN "B"
   charge := 22 + extras*14
WHEN "C"
   charge := 25 + extras*15
OTHERWISE
   PRINT "Error in input."
   charge := 0
ENDCASE
PRINT "Charge is"; charge
```

4.14

```
DIM play$ OF 1, comp$ OF 1, result$ OF 2
INPUT "Type 1 or 2.": play$
comp$ := CHR(RND(49,50))
result$ := play$ + comp$
CASE result$ OF
WHEN "11"
   PRINT "You pay me one penny."
WHEN "12", "21"
   PRINT "I pay you two pence."
WHEN "22"
   PRINT "You pay me three pence."
OTHERWISE
   PRINT "Error in input."
ENDCASE
```

4.15

```
DIM box(30)
FOR count := 1 TO 30 DO box(count) := RND(1,100)
max := 0; min := 999
FOR count := 1 TO 30 DO
   IF box(count) > max THEN max := box(count)
   IF box(count) < min THEN min := box(count)
NEXT count
PRINT "Range of numbers is "; min; "to"; max
```

4.16

```
DIM queue(12), ch$ OF 1
front := 1; nex := 1
FOR item := 1 TO 10 DO
   READ tim, ch$
   IF ch$ = "A" THEN
      queue(nex) := tim
      nex := nex + 1
   ENDIF
```

```
      IF ch$ = "D" THEN
        IF front >= nex THEN
          PRINT "Queue is empty"
          STOP
        ELSE
          PRINT "Time in queue was"; tim - queue(front)
          front := front+1
        ENDIF
      ENDIF
    NEXT item
    DATA 1, "A", 3, "A", 7, "A", 10, "A", 15, "A", 16, "D", 18, "D",
                                                  20, "A", 21, "D", 23, "D"
```

Note

The reader who tests the program carefully will find that the queue "walks" along the array. This can be countered by resetting both *front* and *next* to 1 when either goes above the size of the array. The direct simulation of a queue can be an awkward program and may be avoided if the necessary data can be extracted in other ways.

4.17

```
DIM stack(12), op$ OF 4
point := 1
REPEAT
  INPUT "PUSH or POP?" : op$
  CASE op$ OF
  WHEN "PUSH"
    INPUT "What number do you wish to stack?" : num
    stack(point) := num
    point := point + 1
  WHEN "POP"
    point := point - 1
    IF point < 1 THEN STOP
    PRINT "Popped item is "; stack(point)
  OTHERWISE
    PRINT "Input error"
  ENDCASE
UNTIL point > 12
```

4.18

```
DIM player$ OF 1, comp$(3) OF 1, result$ OF 2
FOR k:= 1 TO 3 DO READ comp$(k)
INPUT "Type S or P or B.": player$
result$ := player$ + comp$ (RND(1,3))
CASE result$ OF
WHEN "SS"
  PRINT "Both cut. A draw."
```

```
WHEN "SP"
   PRINT "Scissors cut paper. You win."
WHEN "SB"
   PRINT "Brick blunts scissors. I win."
WHEN "PS"
   PRINT "Paper cut by scissors. I win."
WHEN "PP"
   PRINT "Both Paper. A draw."
WHEN "PB"
   PRINT "Paper wraps brick. You win."
WHEN "BS"
   PRINT "Brick blunts scissors. You win."
WHEN "BP"
   PRINT "Brick wrapped by paper. I win."
WHEN "BB"
   PRINT "Both brick. A draw."
OTHERWISE
   PRINT "Error in input."
ENDCASE
DATA "S", "P", "B"
```

4.19

```
DIM price(12)
FOR item := 1 TO 12 DO
   READ num
   price(item) := num
NEXT item
FOR guest := 1 TO 3 DO
   charge := 0
   REPEAT
      READ choice
      IF choice > 0 THEN charge := charge + price(choice)
   UNTIL choice < 0
   PRINT "Charge for guest is "; charge
NEXT guest
DATA 1,1.20,1.80,2,1.50,0.80,0.60,1.30,1,2,1.30,1.40
DATA 2,6,8,-1,1,2,5,7,-1,3,6,8,-1
```

4.20

```
DIM box(10)
FOR count := 1 TO 10 DO READ box(count)
FOR run := 1 TO 9 DO
   FOR pair := 1 TO 9 DO
      IF box(pair) > box(pair+1) THEN
         temp := box(pair)
         box(pair) := box(pair+1)
```

```
            box(pair +1) := temp
          ENDIF
        NEXT pair
      NEXT run
      FOR count := 1 TO 10 DO PRINT box(count);
```

4.21 METHOD

Suppose that ten tons of mix required. The quantities required are:

$$\text{sand} = \frac{2}{2+1+3} \times 10$$

$$\text{cement} = \frac{1}{2+1+3} \times 10$$

$$\text{aggregate} = \frac{3}{2+1+3} \times 10$$

PROBLEM ANALYSIS/PROGRAM DESIGN

(1) A DATA statement will record the nine numbers
DATA 2.5, 1, 4, 2, 1, 3, 3, 1, 0

(2) Three sets of fractions will be stored in an array *part* (9).

(3) A letter A, B or C will be read together with a weight.

(4) A CASE statement will assign a value to *point* indicating the first of the three relevant items in the array.

(5) The computation will follow as described above.

```
DIM part(9), mix$ OF 1
FOR k:= 1 TO 9 DO READ part(k)
READ mix$, weight
CASE mix$ OF
WHEN "A"
   point := 1
WHEN "B"
   point := 4
WHEN "C"
   point := 7
ENDCASE
san := part(point); cem := part(point+1); agg := part (point+2)
tot := san + cem + agg
sand := weight * san/tot
cement := weight * cem/tot
aggreg := weight * agg/tot
PRINT "Sand"; sand; "Cement"; cement; "Aggregate"; aggreg
DATA 2.5, 1, 4, 2, 1, 3, 3, 1, 0, "B", 10
```

Note
The above method is not the neatest or shortest possible but it is fairly straightforward once the method is understood.

CHAPTER 5 (page 83)

5.1
```
num := 1;count := 0;million := 1000000
REPEAT
  EXEC double
  count := count+1
UNTIL num > million
PRINT count
PROC double
  num := num*2
ENDPROC
```

5.2
```
INPUT num
EXEC double(num)
PRINT count
PROC double(num)
  product := num;count := 0;million := 1000000
  REPEAT
    product := product *2
    count := count +1
  UNTIL product > million
ENDPROC
```

5.3
```
FOR count := 1 TO 12 DO
  EXEC ranlet
NEXT count
PROC ranlet CLOSED
  DIM letter$ OF 1
  count := 0
  REPEAT
    letter$ := CHR(RND(193,218))
    count := count +1
  UNTIL letter$ = "Z"
  PRINT count
ENDPROC
```

5.4 (a)
```
DIM alfa$ OF 6, ch$ OF 1
INPUT alfa$
NUM := 0
FOR k := 1 TO LEN(alfa$) DO
  ch$ := alfa$(k:1)
```

```
      newdig := ORD(ch$) – 48
      num := num*10+ newdig
    NEXT k
    PRINT num
```

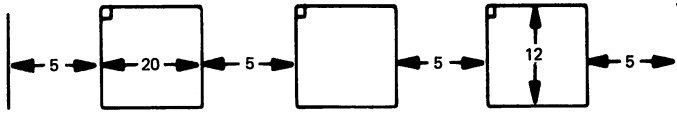
5.4 (b)

```
DIM alfa$ OF 7, ch$ OF 1
INPUT alfa$
EXEC numcon
PRINT num/10↑ dec
PROC numcon
  flag := 0;dec := 0;num := 0
  FOR k := 1 TO LEN(alfa$) DO
    ch$ := alfa$(k:1)
    IF ch$ = "." THEN
      flag := 1
    ELSE
      newdig := ORD(ch$)–48
      num := num*10 + newdig
      IF flag = 1 THEN dec := dec + 1
    ENDIF
  NEXT k
ENDPROC
```

5.5 METHOD

Size of pictures

The screen size of 24 X 80 suggests a size of about twelve down and twenty across to allow space all round and possible messages.



Positions of pictures

The top left-hand positions of each picture will be roughly (6,6), (31,6), (56,6). All positional data will be computed relative to these base positions. The down co-ordinates will always be the same but across co-ordinates will be relative to the variable *basac.*

Picture Data

The square is the simplest picture, being one procedure call with parameters:

```
    ac = basac
    dn = 6
 width = 20
  hite = 12
```

Picture routines

Clearly we require five picture drawing procedures : *square, window, car, cross, face.* Each will require only one parameter which is *basac* or something equivalent. The main program will generate a random number in range one to five to determine which picture and in a FOR loop the value of *basac* will determine the position.

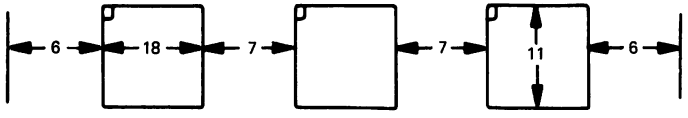
Picture designs

Consideration of the detail of the pictures leads to two immediate rules.

(1) Windows and crosses are more easily drawn with odd numbers of squares for the overall size.

(2) Because of the difference in proportions, vertical lines of two characters width will approximately match horizontal lines of one character in height.

This leads to a modification of the previous numbers.



The *co-ordinates* of the three reference points become (7,7), (32,7), (57,7) and *basac* should take values 7, 32, 57. The pictures can now be drawn on a screen layout chart, which is just a representation of all screen character positions. The cross shown in Fig. S.1 is shown in position 1.

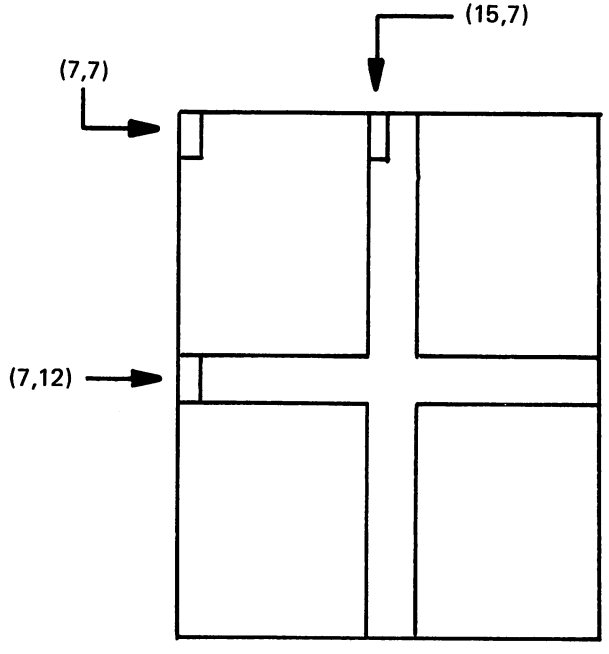


Fig. S.1. Cross in Position 1.

Window. This is most easily seen as three rows and three vertical rectangles.

```
row(ac, dn, width)
   7   7   18
   7  12   18
   7  17   18

box(ac, dn, width, hite)
   7   7    2    11
  15   7    2    11
  23   7    2    11
```

Car. The car consists of two rectangles and two rows.

```
row(ac, dn, width)
  10  15    2
  20  15    2

box(ac, dn, width, hite)
   7  10   11    5
  18  12    7    3
```

Cross. The cross is a row and a rectangle

```
row(ac, dn, width)
   7  12   18

box(ac, dn, width, hite)
  15   7   18   11
```

Square. The square is just one rectangle.

```
box(ac, dn, width, hite)
   7   7   18   11
```

Face. The face is six rows and two vertical rectangles. Four of these elements are the same as those used in the window.

```
row(ac, dn, width)
   7   7   18
   7  17   18
  12  10    3
  17  10    3
  15  12    2
  13  14    6

box(ac, dn, width, hite)
   7   7    2   11
  23   7    2   11
```

The data is correct for a square in position 1.
For pictures in position two or three the following numbers must be added to each *ac* value.

Position 2 25
Position 3 50

PROBLEM ANALYSIS/PROGRAM DESIGN

(1) The display of three pictures will be initiated by the user running the program.

(2) The picture selection routine is essentially,

```
FOR pos := 1 TO 3 DO
basac := 7 + (pos-1) * 25
   pic := RND(1,5)
   CASE pic OF
      WHEN 1
         EXEC window(basac)
      WHEN 2
         EXEC car(basac)
      WHEN 3
         EXEC cross(basac)
      WHEN 4
         EXEC square(basac)
      WHEN 5
         EXEC face(basac)
      ENDCASE
NEXT pos
```

(3) The co-ordinates in data all relate to position 1.

```
PROC window(basac)
   EXEC row(basac, 7,18)
   EXEC row(basac, 12,18)
   EXEC row(basac, 17,18)
   EXEC box(basac, 7,2,11)
   EXEC box(basac + 8,7,2,11)
   EXEC box(basac + 16,7,2,11)
ENDPROC
PROC car(basac)
   EXEC row(basac + 3,15,2)
   EXEC row(basac + 13,15,2)
   EXEC box(basac, 10,11,5)
   EXEC box(basac + 11,12,7,3)
ENDPROC
PROC cross(basac)
```

```
    EXEC row(basac, 12,18)
    EXEC box(basac + 8,7,2,11)
ENDPROC
PROC fac(basac)
    EXEC row(basac, 7,18)
    EXEC row(basac, 17,18)
    EXEC row(basac + 5,10,3)
    EXEC row(basac + 10,10,3)
    EXEC row(basac + 8,12,2)
    EXEC row(basac + 6,14,6)
    EXEC box(basac, 7,2,11)
    EXEC box(basac + 18,7,2,11)
ENDPROC
PROC square(basac)
    EXEC box(basac, 7,18,11)
ENDPROC square
```

Plus
Procedures box, row, point as in example 5.2.
Procedure cursor(ac,dn) as in section 5.3.

PROGRAM INITIALISATION (For Commodore PET)

```
DIM down$ OF 24, right$ OF 80, bar$ OF 1
FOR k := 1 TO 24 DO down$(k) := CHR(17)
FOR k := 1 TO 80 DO right$(k) := CHR(29)
PRINT CHR(147)
```

CHAPTER 7 (page 113)

7.1 The stages in the solution are given to reflect the major developments. Other diagrams were drawn as intermediate stages or as trial analyses but in the interests of saving space and not boring the reader they are not all included.

Stage 1
Figure S.2 gives a first view of the solution.

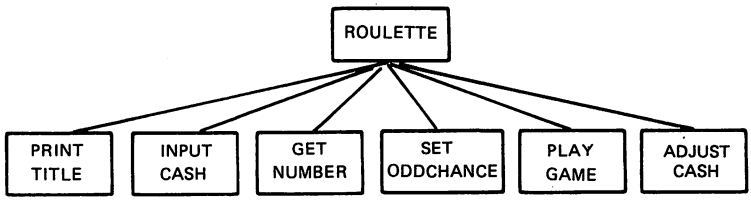


Fig. S.2. Stage 1.

Stage 2

It is immediately obvious that the last four boxes should be in a REPEAT loop which continue until the player quits or runs out of cash. See Fig. S.3.

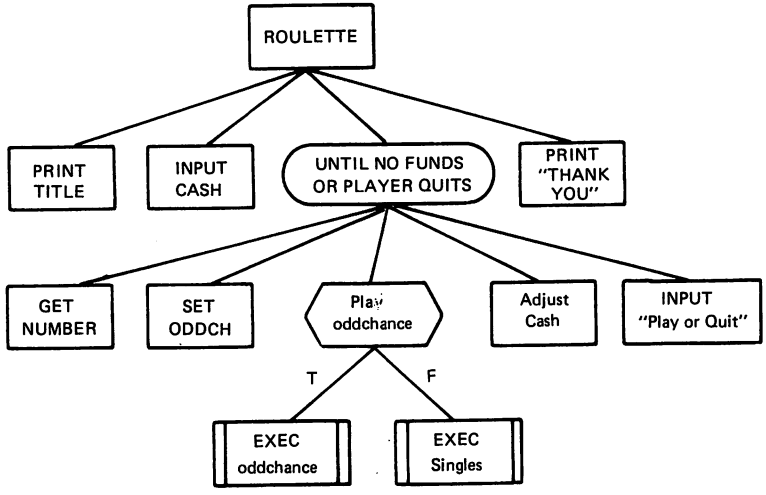


Fig. S.3. Stage 2.

Stage 3

(1) We can now see a fault in the spec. Oddchance should only be set if the player chooses that game. We will change the name to 'oddevens' and calls the other game 'singles'.

(2) The invitation to 'Play or Quit' should only be made if there is cash left.

(3) *Every* INPUT should be placed in a REPEAT loop to ensure that user input is properly validated.

(4) There are five such inputs altogther and the structure is becoming complex. We will therfore define a procedure, *play*, which will handle the gaming. This procedure will call the procedures mentioned in (1) above. See Figs. S.4, S.5 and S.6.

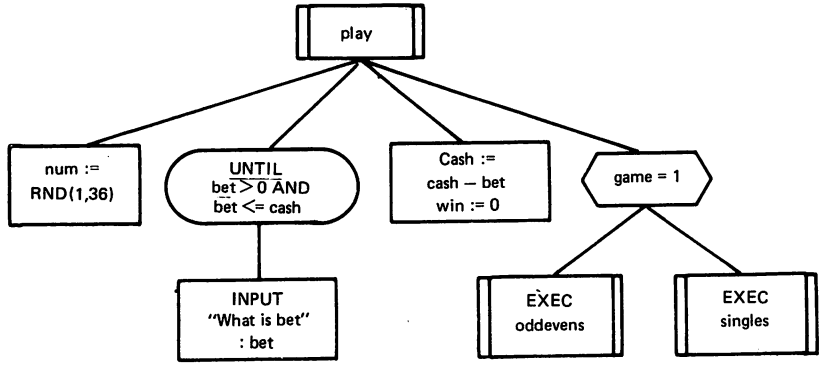


Fig. S.4. Procedure play.

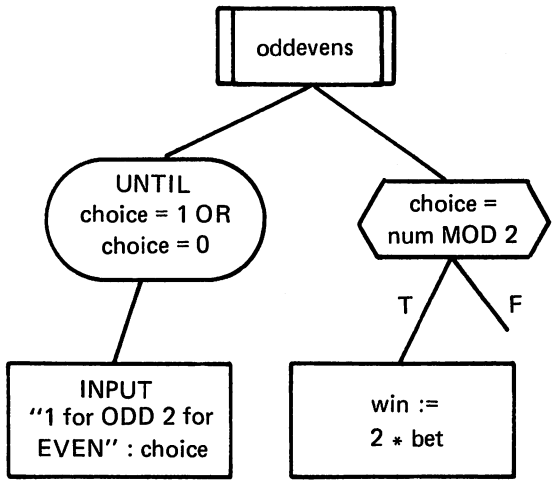


Fig. S.5. Procedure oddevens

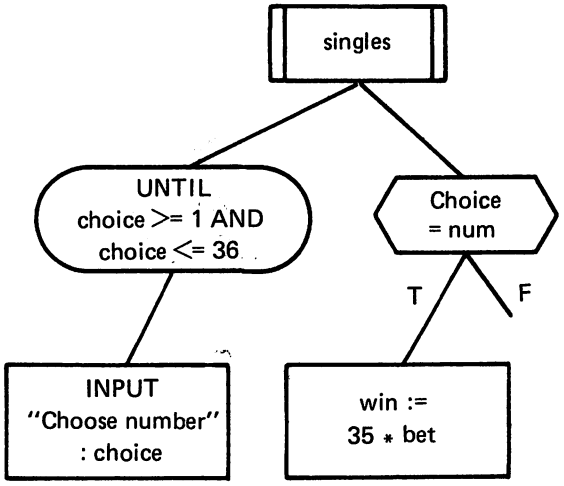


Fig. S.6. Procedure singles.

Stage 4

The original analysis has been modified and can now be expanded to take account of all the details of interaction with the user. See Fig. S.7.

7.2 PROGRAM

```
PRINT "ROULETTE"
REPEAT
  INPUT "How much cash are you playing?" : cash
```

```
UNTIL cash > 0 AND cash <= 100
REPEAT
   REPEAT
      INPUT "Type 1 for oddevens or 2 for singles": game
   UNTIL game = 1 OR game = 2
   EXEC play
   IF win = 0 THEN PRINT "You lose"
   IF win > 0 THEN PRINT "You win"; win
   cash := cash + win
   PRINT "You have"; cash
   REPEAT
      INPUT "Type 0 to quit or 1 to continue.":continue
   UNTIL continue = 0 OR continue = 1
UNTIL continue = 0 OR cash <= 0
PRINT "Thank you for the game."
PROC play
   num := RND(1,36)
   REPEAT
      INPUT "How much do you bet?" : bet
   UNTIL bet > 0 AND bet <= cash
   cash := cash-bet; win := 0
```

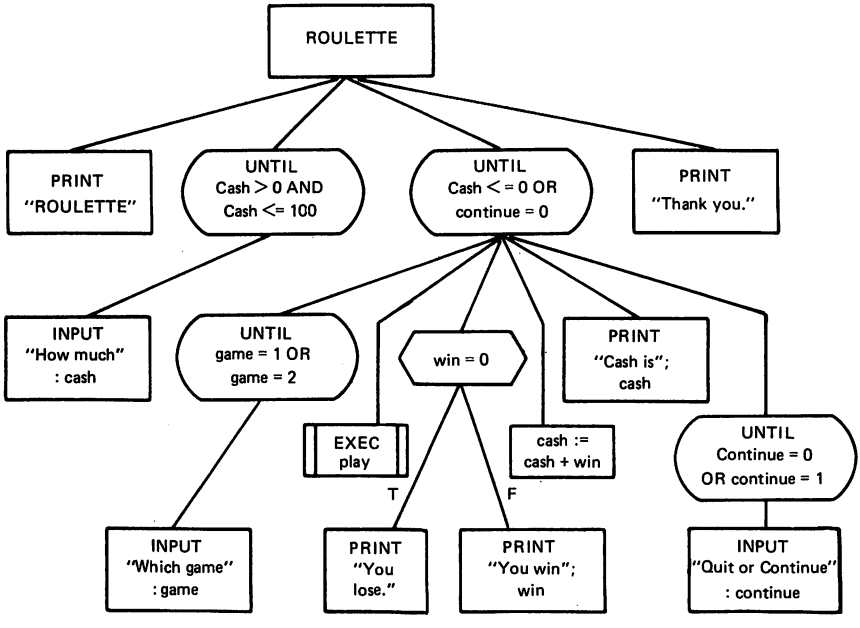


Fig. S.7. Stage 4 of analysis.

```
    IF game = 1 THEN EXEC oddevens
    IF game = 2 THEN EXEC singles
ENDPROC play
PROC oddevens
    REPEAT
        INPUT "Type 1 for ODD or 0 for EVEN.": choice
    UNTIL choice = 1 OR choice = 2
    IF choice = num MOD 2 THEN win := 2*bet
ENDPROC
PROC singles
    REPEAT
        INPUT "Choose from 1 to 36." : choice
    UNTIL choice >= 1 AND choice <= 36
    IF choice = num THEN win := 35*bet
ENDPROC
```

CHAPTER 8 (page 125)

8.1

```
INPUT first, second
WHILE first >= second DO
    first := first – second
ENDWHILE
PRINT first
```

8.2

```
DIM text$ OF 50, ch$ OF 1
READ ch$
count := 0
WHILE ch$ <> "*" DO
    count := count+1
    text$ (count:1) := ch$
    READ ch$
ENDWHILE
FOR k := count TO 1 STEP –1 DO PRINT text$(k:1)
PRINT count
DATA "A", "B", "L", "E", " ", "W", "A", "S", " ",
     "I", " ", "E", "R", "E", " ", "I", " ",
     "S", "A", "W", " ", "E", "L", "B", "A", "*"
```

8.3

```
total := 0
FOR ex := 1 TO 20 DO
    count := 0
    WHILE RND(1,6) <> 6 DO count := count+1
    total := total + count
NEXT ex
PRINT "Average run is "; total/20
```

8.4 *Solution 1*

```
DIM letter$ OF 1
INPUT num
WHILE num >= 1000 DO EXEC printit ("M",1000)
WHILE num >=  500 DO EXEC printit ("D",500)
WHILE num >=  100 DO EXEC printit ("C",100)
WHILE num >=   50 DO EXEC printit ("L",50)
WHILE num >=   10 DO EXEC printit ("X",10)
WHILE num >=    5 DO EXEC printit ("V",5)
WHILE num >=    1 DO EXEC printit ("I",1)
PROC printit (letter$,value)
  PRINT letter$;
  num := num–value
ENDPROC
```

Solution 2

```
DIM letter$ OF 1
INPUT num
FOR k := 1 TO 7 DO
  READ letter$, value
  WHILE num >= value DO
    PRINT letter$;
    num := num – value
  ENDWHILE
NEXT k
DATA "M",1000,"D",500,"C",100,"L",50,"X",10,"V",5,"I",1
```

8.5

```
INPUT first, second
WHILE first <> second DO
  IF first > second THEN
    first := first – second
  ELSE
    second := second–first
  ENDIF
ENDWHILE
PRINT first,second
```

8.6

```
total := 0
FOR game := 1 TO 20 DO
  pay := 0
  WHILE RND(0,1)=0 DO pay := pay+1
  PRINT pay
  total := total + pay
NEXT game
PRINT "Average is "; total/20
```

8.7
```
FOR trial := 1 TO 10 DO
   total := 0
   FOR game := 1 TO 20 DO
      pay := 0;bet := 0.5
      WHILE RND(0,1) = 0 DO
         bet := bet*2
         pay := pay+bet
      ENDWHILE
      PRINT pay
      total := total+pay
   NEXT game
   PRINT total;
NEXT trial
```

8.8
```
total := 0
FOR man:= 1 TO 4 DO
   com := 0
   READ num,item
   WHILE num > 0 DO
      com := num*item + com
      READ num, item
   ENDWHILE
   com := com/10
   PRINT "Commission for salesman"; man; "is"; com
   total := total + com
NEXT man
PRINT "Total is"; total
DATA 2,6,3,15,7,54,-1,-1
DATA 7,13,5,21,13,83,6,94,-1,-1
DATA -1,-1
DATA 8,11,7,34,11,60,-1,-1
```

8.9
```
DIM price(10), tally(10)
FOR k := 1 TO 10 DO READ price(k)
FOR card := 1 TO 5 DO
   cost := 3
   READ item
   WHILE item > 0 DO
      cost := cost + price(item)
      tally(item) := tally(item)+1
      READ item
   ENDWHILE
   PRINT "Total for card"; card; "is"; cost
NEXT card
```

```
FOR item := 1 TO 10 DO PRINT item; tally(item)
DATA 2,4,2,1.5,2.5,3,2,1.5,6,8
DATA 3,5,7,-1
DATA 2,5,6,-1
DATA 5,7,8,9,-1
DATA -1
DATA 1,2,5,9,-1
```

8.10

```
DIM range(6)
FOR k:= 1 TO 6 DO range := 0
FOR game := 1 TO 100 DO
  score := 0
  FOR die := 1 TO 6 DO score := score + RND(1,6)
  IF score <= 10 THEN
    range(1) := range(1)+1
  ELIF score <= 15 THEN
    range(2) := range(2)+1
  ELIF score <= 20
    range(3) := range(3)+1
  ELIF score <= 25
    range(4) := range(4)+1
  ELIF score <= 30
    range(5) := range(5)+1
  ELIF score <= 35
    range(6) := range(6)+1
  ELSE
   PRINT "SPECIAL PRIZE"
  ENDIF
NEXT game
FOR k:= 1 TO 6 DO PRINT k; range(k)
```

Note

If the problem had not specified the use of the ELIF construct the following program might be an alternative. However, it will be noted that the use of mathematical 'tricks' makes the second program less easily readable.

```
DIM range(6)
FOR k := 1 TO 6 DO range(k) := 0
FOR game := 1 TO 100 DO
  score := 0
  FOR die := 1 TO 6 DO score := score + RND(1,6)
  index := (score-1) DIV 5
  IF index <= 6 THEN
    range(index) := range(index) +1
```

```
    ELSE
      PRINT "SPECIAL PRIZE"
    ENDIF
  NEXT game
  FOR k := 1 TO 6 DO PRINT k; range(k)
```

8.11

```
  DIM grade$(5) OF 40, ch$ OF 1,point(5)
  FOR k := 1 TO 5 DO point(k) := 0
  FOR letter := 1 TO 100 DO
    ch$ := CHR(RND(65,90))
    IF ch$ <= "F" THEN
      point(1) := point(1) + 1
      grade$(1,point(1) : 1) := ch$
    ELIF ch$ <= "K"
      point(2) := point(2) + 1
      grade$(2,point(2) : 1) := ch$
    ELIF ch$ <= "Q"
      point(3) := point(3) + 1
      grade$(3,point(3) : 1) := ch$
    ELIF ch$ <= "U"
      point(4) := point(4) + 1
      grade$(4,point(4) : 1) := ch$
    ELSE
      point(5) := point(5) + 1
      grade$(5,point(5) : 1) := ch$
    ENDIF
  NEXT letter
  FOR k := 1 TO 5 DO
    PRINT grade$(k); point(k)
  NEXT k
```

8.12 METHOD

(1) Data from each statement will be READ first into *line*$ and then placed into text$ by joining successive lines.

(2) Each character will then be examined and if it is a u.c. letter the following action will occur,

(a) The letter will be converted in such a way that:

```
A   1
B   2
:
Z   26
```

(b) One of 26 array elements, *tally*, will be incremented,

```
tally(code) := tally(code) + 1
```

(3) The array, *tally*, will be used to plot 26 columns of stars.

```
PROGRAM
    DIM tally(26), line$ OF 80, text$ OF 320, char$ OF 1
  FOR ln := 1 TO 4 DO
    READ line$
    text$ := text$+line$
  NEXT ln
  FOR ch := 1 to LEN(text$) DO
    char$ := text$(ch : 1)
    IF char$ >= "A" AND char$ <= "Z" THEN
      code := ORD(char$)-192
      tally(code) := tally(code)+1
    ENDIF
  NEXT ch
  PRINT CHR$(147)
  FOR ac := 1 TO 26 DO
    FOR dn := 1 TO tally(ac) DO
      EXEC cursor(ac,dn)
      PRINT "*"
    NEXT dn
  NEXT ac
  DATA "HERE WITH A LOAF OF BREAD BENEATH THE
                                                BOUGH,"
  DATA "A BOOK OF VERSE, A GLASS OF WINE AND THOU"
  DATA "BESIDE ME, SINGING IN THE WILDERNESS,"
  DATA "AND WILDERNESS IS PARADISE ENOW."
```

8.13

```
DIM text$ OF 80, ch$ OF 1
count := 0; point := 1
READ text$
ch$ := text$(point : 1)
WHILE ch$ <> "*" DO
  word := FALSE
  WHILE ch$ <> "*" AND (ch$ < "A" OR ch$ > "Z") DO
    point := point + 1
    ch$ := text$ (point:1)
  ENDWHILE
  WHILE ch$ >= "A" AND ch$ <= "Z" DO
    point := point +1
    ch$ := text$(point: 1)
    word := TRUE
  ENDWHILE
  IF word THEN count := count + 1
ENDWHILE
PRINT count
```

```
DATA "SHE IS FOREMOST OF THOSE THAT I WOULD HEAR
                                        PRAISED.*"
```

8.14

```
DIM text$ OF 500, ch$ OF 1, buf$ OF 80
text$ := " "
FOR line := 1 TO 6 DO
   READ buf$
   text$ := text$ + buf$
NEXT line
count := 0; point := 1; dotcount := 0; lettercount := 0
ch$ := text$(point : 1)
WHILE ch$ <> "*" DO
word := FALSE
   WHILE ch$ <> "*" AND (ch$ < "A" OR ch$ > "Z") DO
      IF ch$ = "." THEN dotcount := dotcount + 1
      point := point + 1
      ch$ := text$(point : 1)
   ENDWHILE
   WHILE ch$ >= "A" AND ch$ <= "Z" DO
      point := point + 1
      ch$ := text$ (point : 1)
      lettercount := lettercount + 1
      word := TRUE
   ENDWHILE
   IF word THEN count := count + 1
ENDWHILE
avsent := count/dotcount
avword := lettercount/count
ari := 0.5*avsent + 4.71 * avword – 15.43
PRINT avsent, avword, ari
DATA "THE ICE TALONS SET HARDER IN THE LAND."
DATA "NO TWITTER OF FINCH OR LINNET WAS HEARD"
DATA "ON THE BURROWS, FOR THOSE WHICH REMAINED"
DATA "WERE DEAD. VAINLY THE LINNETS HAD SOUGHT"
DATA "THE SEEDS LOCKED IN THE PLANTS OF THE"
DATA "GLASSWORT. EVEN CROWS DIED OF STARVATION.*"
```

8.15

```
DIM text$ OF 500, ch$ OF 1, buf$ OF 80
EXEC gettext
PRINT ari
PROC gettext
   text$ := " "
   FOR line := 1 TO 6 DO
      READ buf$
      text$ := text$ + buf$
```

```
    NEXT line
    EXEC compute(text$,index)
    ari := index
  ENDPROC
  PROC compute(text$,REF ari)
    count := 0;point := 1;dotcount := 0;lettercount := 0
    ch$ := text$(point:1)
    WHILE ch$ <> "*" DO
      word := FALSE
      WHILE ch$ <> "*" AND (ch$ < "A" OR ch$ > "Z") DO
        IF ch$ = "Z" THEN dotcount := dotcount + 1
        point := point + 1
        ch$ := text$(point : 1)
      ENDWHILE
      WHILE ch$ >= "A" AND ch$ <= "Z" DO
        point := point + 1
        ch$ := text$(point : 1)
        lettercount := lettercount + 1
        word := TRUE
      ENDWHILE
      IF word THEN count := count + 1
    ENDWHILE
    avsent := count/dotcount
    avword := lettercount/count
    ari := 0.5*avsent + 4.71*avword – 15.43
  ENDPROC
  DATA "ONCE YOU CAN APPLY THE USE OF PROCEDURES"
  DATA "INTELLIGENTLY TO YOUR COMPUTING PROBLEMS,A"
  DATA "WHOLE NEW WORLD OPENS UP;POWER AT YOUR"
  DATA "FINGERTIPS ! – BRAILSFORD AND WALKER,"
  DATA "INTRODUCTORY ALGOL 68 PROGRAMMING."
  DATA "ELLIS HORWOOD,1979."
```

8.16 The changes are indicated.

```
  EXEC gettext
  PRINT index
  PROC gettext
    –
    –
    –
    –
    index := ari
  ENDPROC
```

```
PROC ari(text$)
   —
   —
   —
   —
   —
   ari := 0.5*avsent + 4.71*avword – 15.43
ENDPROC
```

Index

The notes on syntax and problem grading in the appendices are not indexed.

Roy Atherton is Senior Lecturer in Computer Studies, and Director of the Computer Education Centre, at Bulmershe College of Higher Education, Reading. He is a graduate of the University of London with a B.Sc. in Mathematics in 1964, to which he added an M.Tech. in Computer Science from Brunel University in 1973. In 1964 he became Head of Mathematics and Computing at Stretford Technical College, where he remained until 1969; he has been adviser in computer education to the Berkshire County Council for three years; and is a Member of the British Computer Society.

COMPUTING USING BASIC: An Interactive Approach

TONIA COPE, University of Oxford Computing Teaching Centre

This comprehensive coverage of modern interactive BASIC is based on a teaching manual successfully used in courses at the University of Oxford since 1977, by a broad spectrum of people ranging from school children to businessmen, doctors and university professors. For this book the presentation has been adapted for a popular educational microcomputer, but its firm basis in other systems means that the essence of the book is appropriate for BASIC on a wide range of microcomputers.

THE MICROCHIP: Appropriate or Inappropriate Technology?

ALAN BURNS, Lecturer in Computer Science, University of Bradford

"Burns' book is better than most" – Tony Cooper in *Computing.*

"A new book which assembles the topics and offers newcomers the opportunity to familiarise themselves" – *Electronics Weekly.*

FOUNDATIONS OF PROGRAMMING WITH PASCAL

LAWRIE MOORE, Head of Computing Services, Birkbeck College, University of London

"an excellent book . . . brings to life the academic principles . . . lucid and easy to follow . . . shows that it is possible to teach programming in a way that is enjoyable and easy, but which does not violate the disciplines of software engineering" – *Personal Computer World.*

"Valuable reading" – Michael Coad, Lecturer in Computer Science, University of Essex, in *Computing.*

"well-written textbook . . . compact and yet thorough . . . effectively-presented diagrams" – *Choice* (USA).

PROGRAMMING LANGUAGE STANDARDISATION

Edited by I. D. HILL, Clinical Research Centre, Harrow and B. L. MEEK, Director, Computer Unit, Queen Elizabeth College, University of London

"This is a consistently readable survey of the subject, and a useful reference, backed up by as complete an index to standard bodies throughout the ADP world as you will find anywhere" – F. H. Little in *Computer Management.*

"A joy of a book" – *Computer Books Review.*

GUIDE TO GOOD PROGRAMMING PRACTICE

Edited by B. L. MEEK, Director, Computer Unit, Queen Elizabeth College, University of London, and PATRICIA HEATH, Plymouth Polytechnic Computing Centre

Brings together, in compact and assimilable form, the many and varied aspects of the programmer's work; structural programming, program-writing techniques, analysis, design, testing, debugging, "tuning" for greater efficiency, coping with the limitations of computer selection of a language and the use of language standards, program documentation, maintenance, transferability, and taking over another programmer's work. A clear picture in one comprehensive volume.

published by
ELLIS HORWOOD LIMITED
Publishers Chichester

distributed by
HALSTED PRESS a division of
JOHN WILEY & SONS
New York Chichester Brisbane Toronto

Halsted Press Library Edition ISBN 0-470-27318-6
Halsted Press Paperback Edition ISBN 0-470-27359-3