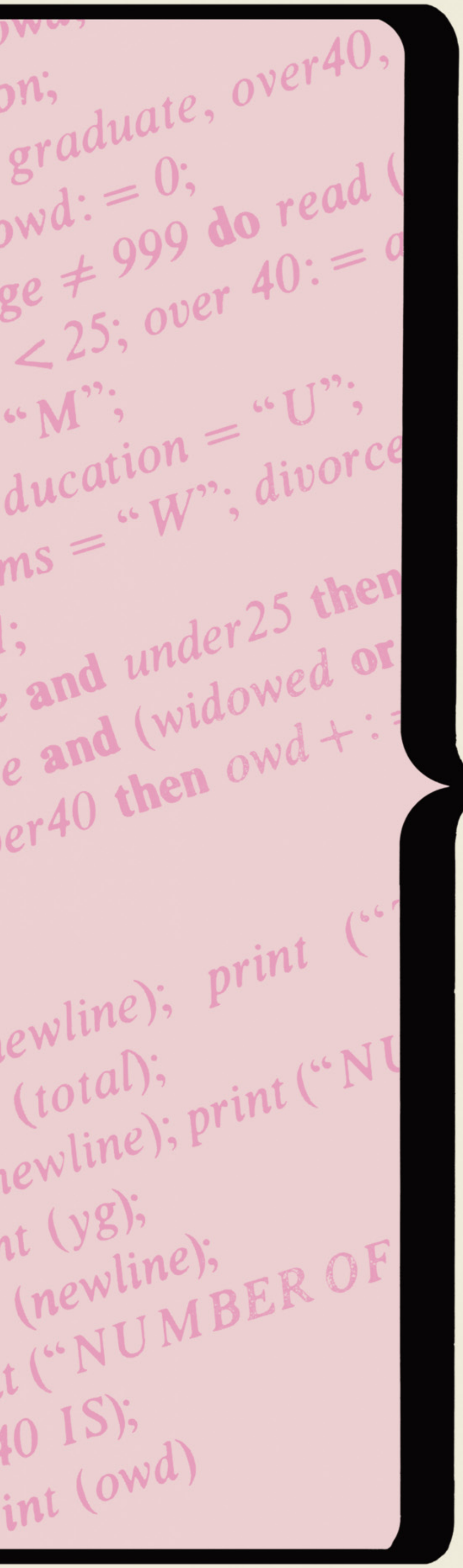


Programming and Problem-solving in Algol 68

Andrew J. T. Colin

Programming and Problem-solving in Algol 68

Macmillan Computer Science Series

Consulting Editor
Professor F.H. Sumner, University of Manchester

S.T. Allworth, *Introduction to Real-time Software Design*

Ian O. Angell, *A Practical Introduction to Computer Graphics*

G.M. Birtwistle, *Discrete Event Modelling on Simula*

T.B. Boffey, *Graph Theory in Operations Research*

Richard Bornat, *Understanding and Writing Compilers*

J.K. Buckle, *The ICL 2900 Series*

J.K. Buckle, *Software Configuration Management*

Derek Coleman, *A Structured Programming Approach to Data**

Andrew J.T. Collin, *Fundamentals of Computer Science*

Andrew J.T. Colin, *Programming and Problem-solving in Algol 68**

S.M. Deen, *Fundamentals of Data Base Systems**

J.B. Gosling, *Design of Arithmetic Units for Digital Computers*

David Hopkin and Barbara Moss, *Automata**

Roger Hutty, *Fortran for Students*

H. Kopetz, *Software Reliability*

A. Learner and A.J. Powell, *An Introduction to Algol 68 through Problems**

A.M. Lister, *Fundamentals of Operating Systems, second edition**

G.P. McKeown and V.J. Rayward-Smith, *Mathematics for Computing*

Brian Meek, *Fortran, PL/I and the Algols*

Derrick Morris and Roland N. Ibbett, *The MU5 Computer System*

John Race, *Case Studies in Systems Analysis*

B.S. Walker, *Understanding Microprocessors*

I.R. Wilson and A.M. Addyman, *A Practical Introduction to Pascal*

*The titles marked with an asterisk were prepared during the Consulting Editorship of Professor J.S. Rohl, University of Western Australia.

Programming and Problem-solving in Algol 68

Andrew J. T. Colin
Professor of Computer Science,
University of Strathclyde

M
Macmillan Education

First edition 1978
Reprinted 1981, 1984

Published by
THE MACMILLAN PRESS LTD
London and Basingstoke
Companies and representatives throughout the world

British Library Cataloguing in Publication Data

Colin, Andrew John Theodore
Programming and problem-solving in Algol 68.
—(Macmillan computer science series).
1. ALGOL (Computer program language)
I. Title II. Series
001.6′424 QA76.73.A24

ISBN 978-0-333-23115-9 **ISBN 978-1-349-03561-8 (eBook)**
DOI 10.1007/978-1-349-03561-8

Contents

Preface

The title of this book is carefully chosen—the main topic is the art of solving problems by programming. The book is intended to appeal to those who believe, like the author, that there is more to this subject than mere knowledge of a programming language. Thus the 'Algol 68' of the title plays a vital but secondary supporting function.

Although much of the book is necessarily about the mechanics of writing programs, I have tried to convey some of the important ideas of programming as a practical discipline — reliability, robustness, economy (considered in a global sense) and structure, both in data and program.

Since the over-all length of the book is limited, I have included only those parts of Algol 68 that would normally be found in a first-year undergraduate course. Omissions — which the reader will be able to fill from other books devoted primarily to Algol 68 — include computing with references, unions, the definition of operators, formats, use of the backing store, and finally labels and **goto** commands. This selection has the additional advantage that the subset that remains is available, with minor variations of dialect, on all existing versions of Algol 68.

Mental models are of great help in understanding complex phenomena. I should perhaps make clear that the 'identifier table' described in chapter 7 is such a model. The reader can safely take this account literally, and it will help him to write correct efficient programs. Later, if he ever studies the design of compilers, he may discover that the same effect can be reached in other more complex ways.

My thanks are due to a great number of people: to two successive first-year classes at Strathclyde University who used preliminary drafts of the book as lecture notes; to Miss Agnes Wisley, who typed the manuscript; to Dr Charles Lindsey, who as referee made many helpful suggestions and to my wife, who drew sketches for the pictures, tracked down the quotations at the head of each chapter, and gave me constant encouragement. The book itself is dedicated to my mother, Mrs Vera Colin, without whom it would not have been.

November, 1976 ANDREW COLIN

Note on Exercises

Each chapter in this book is followed by a set of exercises. The number in brackets at the beginning of each exercise is an estimate of its difficulty. The scale ranges from 0 (trivial) to 10 (hard).

The last exercise is usually marked P for practical. This indicates that an actual computer solution is suggested.

A suitable rate for the exercises is about one set (including the practical) each week.

The exercises marked* have worked solutions at the end of the book.

1 *Computers and Program-solving*

> 'We stand at the brink of a new age, an age made possible by the revolution that is embodied in the computer. Standing on the brink, we could totter either way — to a golden age of liberty or a dark age of tyranny, either of which would surpass anything the world has ever known.'
>
> G. M. Weinberg, *The Psychology of Computer Programming*

Many people who have only a nodding acquaintance with computers regard them as dangerous monsters. Far from being the benign servants that the advertisements would have us believe, they seem more like hostile masters, sending peremptory reminders for TV, car and dog licences. Computers helped to put a man on the Moon but at the same time they can issue a bill for no pounds and no pence and take someone to court for non-payment.

Figure 1.1 is a guide to the real state of affairs. The computer is a machine like any other, and its monstrous appearance is only a mask. All machines have to be driven or controlled, and the driver in this case is the programmer. His job is to give the machine the instructions or rules that make it react to the public in the way it does. You will see that the programmer's seat is not well placed, since his view forward is obscured by the mask. Often he cannot actually see what his rules make the computer do and what effect they have on the people on the other side of the mask.

Of course the programmer does not carry the ultimate responsibility for the rules that he gives the computer. In practice he is told what sort of rules to use by a systems analyst, whose task it is to use the machine to make things run as smoothly and effectively as possible *from the point of view of the organisation that employs him.* Both the analyst and the programmer are watched over by a manager who makes quite sure that they do not forget their place and begin to take the part of the outsider.

The monster's public face is frightening, but its secret behaviour can be even worse. Although it does not yet happen much in practice, it is feasible for the computer to use its knowledge about you in ways that you never intended or even considered possible. For example, if you visit your doctor to discuss a weight problem, and he happens to keep his notes in the memory bank of a computer, you might start receiving glossy advertisements for expensive slimming aids. Unknown to the doctor, some advertising agency might have browsed through the information recorded in the computer and picked out your name, and the names of others like you, for mailing shots as particularly

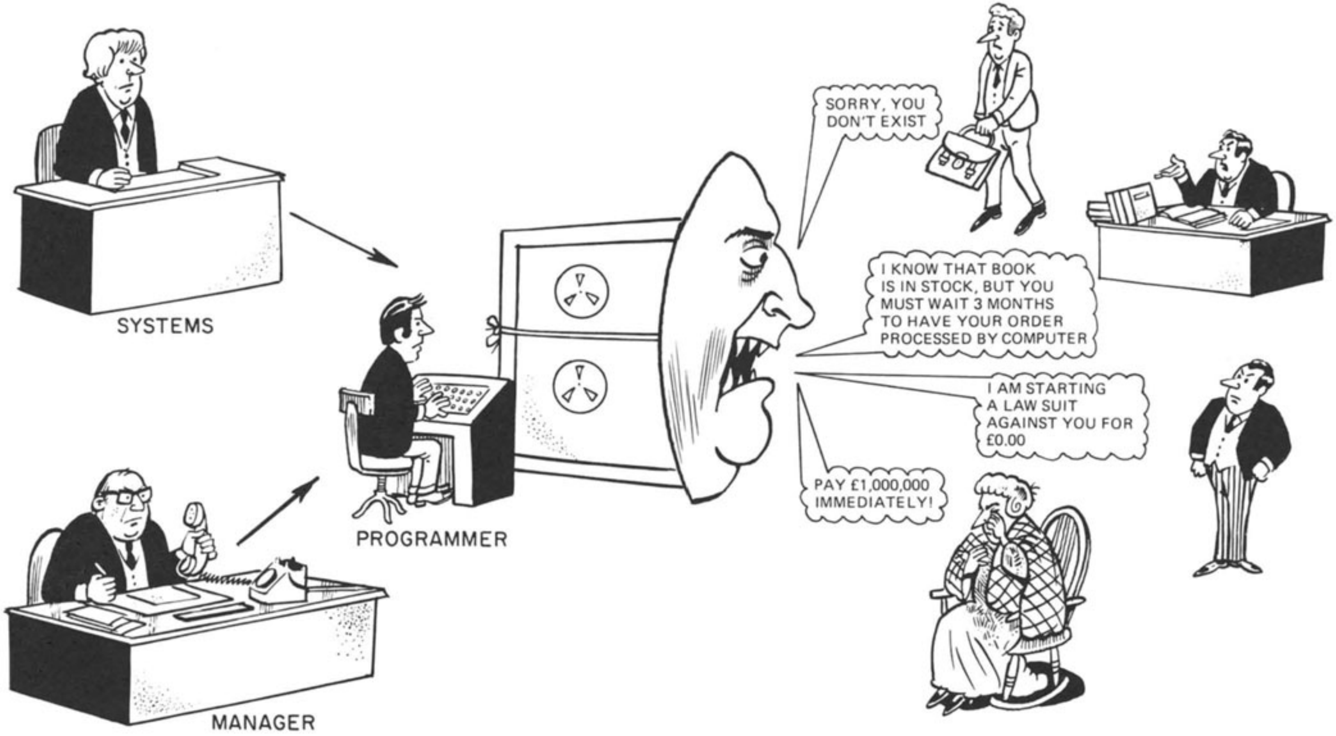

SYSTEMS
MANAGER
PROGRAMMER
SORRY, YOU DON'T EXIST
I KNOW THAT BOOK IS IN STOCK, BUT YOU MUST WAIT 3 MONTHS TO HAVE YOUR ORDER PROCESSED BY COMPUTER
I AM STARTING A LAW SUIT AGAINST YOU FOR £0.00
PAY £1,000,000 IMMEDIATELY!

Figure 1.1

easy 'sells'. As the law stands in the United Kingdom such behaviour would not be illegal.

This is a very mild example, since an unwanted advertisement would not cause you serious annoyance. It does not take much imagination, however, to see the possibilities for blackmail, police harassment, and the theft of commercial secrets, which are opened up as a result of private information being stored in computers. It is true, of course, that much of this information was stored in handwritten notes and ledgers before the computer was invented, and that snoopers could always have searched them and discovered data that they could profitably use. The difference is that, because the information is so systematically arranged in the computer, it can be fully inspected by the machine itself in a few seconds, and there is usually no trace that the search has happened at all.

Most people agree that misuse of information in this way is wrong. The computer itself is morally neutral. Although it makes this kind of abuse feasible, the actual decision to use innocent information for illegitimate purposes would be made by the systems analyst and his manager, and the necessary program would be written by the programmer. The suggestion is not that they are all wicked people; only that they think in terms of generalities like 'expansion of trade', and 'maintenance of law and order' and forget about the individual human beings concerned.

There are a few computer systems where the mask is a kindly one. If such a machine finds you in trouble or difficulty, it will help you as much as it can. If your particular problem is too much for it to sort out on its own, it will never try to stop you from reaching the people actually in control. If you wish to store any confidential information, it gives you every opportunity to safeguard it by using secret cyphers and other protective measures.

On the whole, people are more friendly on the telephone than when answering letters. In the same way, most 'friendly' computer systems allow you to 'talk' directly to the machine on a teletype or a visual display screen with a keyboard. 'Hostile' systems tend to rely on such things as forms, which are filled in and sent through the post, but some also permit a direct dialogue. Here is an example of a possible 'conversation' between a traveller and two computer systems, one friendly and one hostile. The traveller is trying to reserve a ticket, and is about to state his destination. The conversations are imagined as taking place on teletypes — electric typewriters controlled by the computer and connected to it by telephone lines.

Friendly computer systems cost more than hostile ones because it takes more trouble to set them up, but the difference is so slight that in time they may become more common. Of course you must be especially wary of the computer whose friendly mask enables it to extract information, which is then used against your own interests!

```
COMPUTER : WHERE DO YOU WANT TO GO TO?
MAN      : EDNIBURGH
COMPUTER : PARDON?
MAN      : EDINBRGH
COMPUTER : PERHAPS YOU MEAN EDINBURGH?
MAN      : YES
COMPUTER : THANKYOU.  WHICH CLASS WOULD YOU
              LIKE TO TRAVEL?

    . . . . . . . . . . . .
```

(a) Friendly computer

```
    . . . . . . . . . . . .

COMPUTER : DEST?
MAN      : EDNIBURGH
COMPUTER : FAULT 73
MAN      : (Does not know what to do, so he waits.
           After 30 seconds)
COMPUTER : TIMEOUT
           (The teletype now goes dead, and the traveller does
           not know whether his ticket has been booked, or how
           to find out)
```

(b) Hostile computer

Fundamentally, when the mask has been stripped off, a computer is a machine for solving problems. In most cases, the existence of a problem implies that there is something to be done. One speaks, for example, of the problem of clearing the slums, or a students' problem class, or a 'problem' child. In most problems there are three distinct entities

the *solver*, who decides what is to be done
the *client*, who is supposed to benefit by the solution
the *means*, by which the solution is implemented.

The above examples all fit into this mould, as shown in table 1.1.

When he recognises a problem, the would-be solver must consider a number of different factors.

Can the problem be solved, even in principle? Some problems are inherently insoluble because of the laws of physics and mathematics. If modern physics is correct, anyone attempting to make particles travel faster than the speed of light would be wasting his time from the start. Similarly the mathematicians who have tried since ancient times to square the circle or trisect an angle, using only ruler and compasses, were bound to fail—these problems have been proved insoluble.

Problem	*Solver*	*Client*	*Means*
Clearing the slums	Architects and urban planners, who decide to replace tenements with tower blocks	The inhabitants of the slums	Bulldozers, builders, etc.
Problem class	The student, who works through the problems he has been set	The same student, who benefits by the experience	Pencil, paper, logs, etc.
Problem child	Social worker or psychologist	The child, or society, or both	Remand home, Borstal, etc.

Table 1.1

If there is nothing to prevent the problem being solved in principle, are there enough resources for it to be solved in practice? For instance, I would very much like to learn Chinese, and I believe that I could do so if I were to live in China for a couple of years. Unfortunately I have neither the time nor the money to make this possible, and so the problem of my learning Chinese in this way must remain unsolved.

If the problem is soluble in a practical sense, what benefits will the solution bring? Often they are clear and direct. There used to be a problem of endemic cholera in the larger cities; when it was solved by cleaning up the water supply the benefit was immediate and universal. In other cases the results are by no means so clear-cut. If the client is a slum dweller or a problem child, he may not agree that the solution you propose for his 'problem' will benefit him at all. You should be very sure that you are right and he is wrong before you compel him to accept your plan.

What does the solution cost? Every solution has a direct cost that can usually be measured in money; this must be paid by the client, or those who want to benefit him. Unfortunately most proposed ways of solving problems also have indirect costs, which have to be carried by other people. For example, a new machine tool can cause a large reduction in the costs of running a factory and solve the owners' problem of how to make a profit; but it can also put many people out of work. A new road can ease congestion problems, but can also cause a good deal of misery to those who live nearby and have their peace disturbed.

These indirect costs are usually ignored, mainly because there is no easy way of taking them into account. Eventually legislation may force problem-solvers to consider some of them, by making the client pay a very high price (for example, a year in gaol) for adopting a solution that harms other people.

In general, every problem has many different solutions. Clearly any solution that costs more than the benefits it is supposed to bring is simply not worth consideration. From the solutions that remain, the problem-solver must find that which gives the best balance between benefit and cost. If the benefits to be obtained from two possible solutions are exactly the same, then the obvious choice is the cheaper one. In practice this is hardly ever the case, and the solver must use his judgement to decide which solution is the most suitable for the circumstances. The following example shows the kind of thinking that is often useful in selecting one solution out of several that have been suggested.

An engineering company has received an order for a passenger-lift controller, and has decided to charge £1000. It has investigated the situation and discovered a possible market for up to 100 lift controllers of similar design.

The company's design engineers have suggested two approaches to building the controller. The first approach uses conventional relay technology. The initial design work is relatively easy and will cost £5000; thereafter the cost of labour and materials for each controller will be £800.

The second approach relies heavily on modern computer technology, and uses microprocessors and other large-scale integrated circuits. The initial design is expensive (£25 000) because a complex computer program is needed, but once the design is complete the individual construction costs of each unit are much less, only £400.

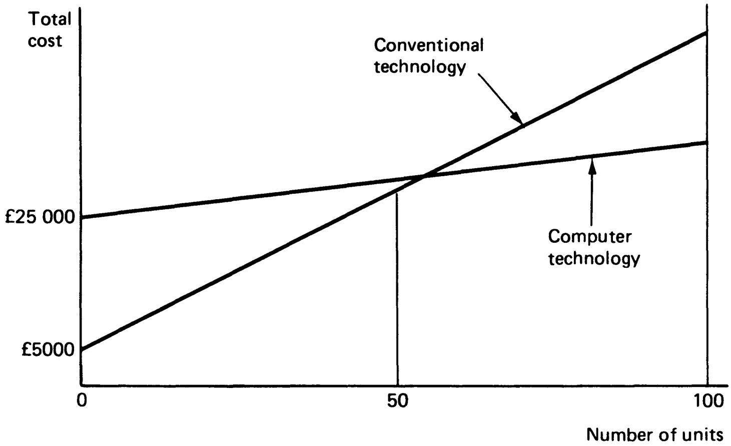


Figure 1.2

Which is the best method to use? There is no clear-cut answer, since the total cost depends on the number of controllers actually produced. The situation is shown in figure 1.2, which shows how the total cost varies with the number produced for each of the two methods. With these figures there is a cross-over point between 52 and 53 units. For a small number the conventional relay technology is the most economical, but for a greater number the computer method wins because of the lower production costs.

If the total number of controllers actually needed were known in advance this graph could be used to select the right method. However, it is more realistic to suppose that the company is trying to sell as many units as possible, and within the limits of 1 to 100 no one knows what the actual sales will be.

Figure 1.3a shows the expected profit (or loss) for various numbers of units built with the conventional technology. There is a break-even point at 25; the maximum loss incurred (if only 1 unit is sold) is £4800, but the greatest possible profit, on sales of 100 controllers, is only £15 000.

Figure 1.3b shows a similar graph for the computerised version. The break-even point is higher at 42, and the maximum loss is £24 400. On the other hand, the maximum profit is a comfortable £35 000.

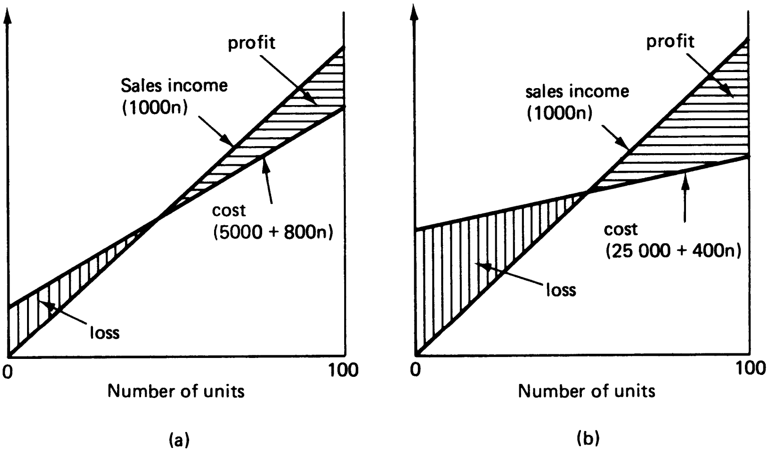


Figure 1.3 (a) Conventional technology; (b) computer technology

The best method for a given company depends on its financial situation. If it has plenty of money, and it can afford to take a risk, then it should choose the computerised solution, since possible profits are so high. On the other hand, a company that is short of capital and hanging on by its teeth to stay in business would be wise to select the conventional technology, even though its final profit may well be less. Indeed this solution may even be forced upon it if its bankers refuse to lend enough money to pay for the more expensive development program needed by the computerised solution.

Suppose, that a general method for solving a problem has been decided. The next stage in the process is for the solver to produce a set of *instructions*. These instructions form the link or *interface* between the solver, who has decided on what is to be done, and the implementor, who actually obeys the instructions and produces the desired results.

The set of instructions can be expressed in various forms, all of them common. Here are some examples.

> 'Place one pint of boiling water in a crock.
> Add 1 teaspoon of Briars' Falsam.
> (If you cannot smell the vapour, add more until you can.)
> Lean over the crock and cover your head with a towel.
> Inhale the vapour for 10—15 minutes.
> Repeat every three hours until your chest is better.'
>
> (From a bottle of Briars' Falsam)

> 'Shape toe as follows:
> 1st round — 1st needle: K to last 3 sts., K2 tog., k1; 2nd needle: k1, s1, k1, psso, K to last 3 sts., K2 tog., kl; 3rd needle: K1, s1, K1, psso, K to end.
> 2nd round — K.
> Rep. these 2 rounds until 26 sts. remain.
> K sts. from 1st needle on to end of 3rd needle.
> Graft sts. or cast-off sts. from two needles tog.'
>
> (From a knitting pattern)

> '. . . . .
> 45) Select the 4K7 resistor (yellow-purple-red)
> Connect it between points J17 and K22
> 46) Select the 2 microfarad electrolytic capacitor, and identify the positive end (marked with + or red) Connect it between L33 (+) and M43 (−)
> . . . .'
>
> (From a do-it-yourself car-radio kit)

These examples cover a wide range of activities. They all assume certain basic skills in the implementor, such as the ability to boil water or to cast on stitches or to use a soldering iron. Some of them make use of a jargon that is related to the area of the problem: K1, P1, 4K7, etc. It is acceptable to use these codes instead of plain English because they are familiar to the people following the instructions and they make the whole list shorter, easier to follow and cheaper to print.

The sets of instructions have other common features as well.

In the first place it is assumed that the implementor will obey them sequentially, that is, one at a time. The order he uses is that in which they are written, except where any other order is explicitly stated (as in 'Repeat every three hours until . . .'). In many sets of instructions the order is vitally important; consider

RICE PUDDING (1)	RICE PUDDING (2)
Boil 1 pint milk	Boil 1 pint milk
Add 2 ounces uncooked rice	Cook for two hours
Cook for two hours	Add 2 ounces uncooked rice
Serve immediately with jam	Serve immediately with jam

Second, the instructions themselves are all relatively simple when compared to the over-all problem to be solved. In the examples given it is easier to make one stitch or solder one joint than to knit a jersey or build a complete radio. Nevertheless this does not mean that the steps are simple in any absolute sense; they might be very complex and require complete sets of instructions on their own. For instance the radio construction leaflet might say

> 'To mount a component.
> Select the component
> Ensure that the leads are straight
> Scrape the leads with a knife or emery paper until they are shiny all over
> Bend the leads with pliers so as to fit the holes in the printed circuit
> Insert the wires into the holes
> Turn the PC board over and bend the wires down
> Cut the leads off leaving 2½ mm on the printed side
> Solder the leads to the copper on the PC board, using resin-cored solder and an electric iron not exceeding 25 watts in power.'

If this puts you off you might like to substitute *exact* instructions for making one knitted stitch (purl) or boiling a pint of milk (remember to pour it out of the bottle into a saucepan and to light the gas!).

The third point of resemblance is that the instructions are all in the *imperative* mood ('Do something'). Some of the instructions may be conditional ('If so and so is the case, then do something') but none of them is indicative ('So and so is the case'). Such assertions are not necessary in a set of instructions, and if they do occur then their only sensible purpose is to check that the other instructions have been followed correctly. Consider the statements

> 'The sleeve is now 28″ long'
> 'The crust is crisp and slightly browned'

If, in the event, the sleeve is actually 48″ long or the crust is white and soggy, you can deduce that you have used the wrong needles or oven temperature, and (if you have the patience and resources) go back to the beginning and start over again.

Sets of instructions of this type are known as *programs*. In many cases the best that can be expected of a program is that it *may* produce the desired result. The Briars' Falsam inhalation may cure the patient if he has a simple cough on the chest, but if he has lung cancer it will do him no good, no matter how long he uses it.

Among programs there is a special category such that, if they are followed correctly, they will produce the required result after a finite number of steps. Such programs are called *algorithms*, and are particularly important where certainty and reliability are to be taken into account. The knitting pattern is an algorithm, because if you follow it precisely you will inevitably finish up with the required article. The radio instructions do not constitute an algorithm because the components themselves may be faulty. It is all too common an experience to follow through the instructions exactly and end up with a set that does not work. The makers of the kit usually convert their program to an algorithm by adding a final instruction

> 'Test the set. If the performance is unsatisfactory return it to Messers Tomb and Son, Mortuary Lane, Gravesend, Kent, who will repair or replace it free of charge.'

EXERCISES

1.1 (1) What is a friendly computer? Describe some of its attributes.

1.2 (1) State clearly who (or what) are the problem, the solver, the client and the means in each of the following situations, where there is something to be done.

(a) You fall and break your leg.
(b) Your domestic heating bills are more than you can afford.
(c) The violent crime rate in your city is too high.
(d) The standard of your local football team needs improvement.

***1.3** (2) You are the landlord of a large student hostel.
You have two methods of buying electricity

(a) tariff 1 imposes a flat charge of 2p per unit
(b) tariff 2 imposes a standing charge of £100 per year + 1p per unit

In the hostel there are 50 rooms. Each resident can use between 0 and 500 units per year. You have two possible ways of charging for electricity

(a) you can impose a flat charge of £5 per year on each resident, or
(b) you can rent a meter for each room at a cost (to you) of £2 per year and adjust it so that the resident pays 3p per unit.

Compare the different ways of dealing in electricity and determine which method (i) can give you the greatest profit, and (ii) ensures the smallest loss (if any). What method would you choose in practice?

1.4 (3) In a shed there is a large pile of bricks. In the yard outside there is a heap of logs. At the door of the shed there lies a particularly malodorous and vicious dog, and you do not want to pass the dog more often than you have to.

Assuming that you can only carry two logs or five bricks at a time, give an algorithm, to be implemented by yourself, for getting the logs into the shed and the bricks into the yard. Your algorithm should ensure that you do not pass the dog more often than you have to. You are not permitted to unchain the dog and take it away.

***1.5** (0) The rules for Snakes and Ladders say, 'To start, shake and throw the dice until a 6 turns up'. Is this an algorithm for starting? Explain your answer.

***1.6** (3) Here is a program for finding the square root of a number n to about two decimal places.

(a) Set a variable x to 1.

(b) So long as the value x^2 differs from n by more than 0.01 repeat the following step

(c) replace the value of x by $\frac{1}{2}(x+n/x)$.

(d) Finally take x as the answer.

Here is an example. Take n = 5.

(a) Set x = 1.

(b) 1^2 (= 1) differs from 5 by more than 0.01, and so

(c) set $x = \frac{1}{2}(1+5/1) = 3$.

(b) 3^2 (= 9) differs from 5 by more than 0.01, and so

(c) set $x = \frac{1}{2}(3+5/3) = 2.333$.

(b) $(2.333)^2$ (= 5.444) differs from 5 by more than 0.01, and so

(c) set $x = \frac{1}{2}(2.333+5/2.333) = 2.238$.

(b) $(2.238)^2$ (= 5.009) differs from 5 by less than 0.01, so

(d) take 2.24 as the square root of 5 to two decimal places.

Use this procedure to find the square roots of 7 and 9, working to three decimal places.

***1.7** (3) Do you think the program given in exercise 1.6 is an algorithm? (Try applying it to the number −2.) If not, what changes would you make to ensure that it is an algorithm?

1.8 (P) Get your course tutor or demonstrator to show you round your local computer installation, to explain what the various parts of the computer actually do and to demonstrate a teletype or visual display unit.

Did you find a programmer in the seat behind the machine? If not, why not?

2 *Information Storage and Processing*

'The Ministry of Truth—Minitrue, in Newspeak—was startlingly different from any other object in sight'

George Orwell, *Nineteen Eighty-Four*

Every machine is designed to handle some particular 'working substance'. Steam engines run on steam, sewing machines sew cloth and cheese graters grate cheese. Some machines, including computers, use *information* as their working substance. In simple terms, information is anything that can be written down. It turns out that, as a working substance, information is unique and has some curious properties.

Most ordinary working substances are physical. They can be weighed by the bushel and measured by the cubit (or pint or millilitre). They are susceptible to being degraded through decay or contamination. Information, however, is abstract. It weighs nothing, and can only be measured by introducing new units. It can still be bought or sold, sometimes (as in time of war) at extremely high prices. Information does not decay in any physical sense, but with the passage of time it tends to become out of date and therefore less *true* than it may have been to start with. This gradual corruption is more insidious than the simple degradation of physical working substances because there are usually no obvious signs that it has occurred.

One of the most important qualities of information is that, once it exists, it can be copied or replicated at very little cost. For example, the effort needed to write a book is prodigious but, once the text has been completed and the print set up, additional copies can be run off very cheaply. This property also makes information particularly liable to theft—all the intruder needs is a camera and, since he need not actually take anything away when stealing information, the theft is that much harder to detect.

A very large fraction of industrial and other endeavours centres on information rather than physical substances. The Post Office, the radio and television companies, the publishers of books and newspapers, and every place of education all use information as their major or only working substance.

Information engineering, like all other sorts of engineering, relies on measurement. There are several different ways of measuring information. A practical unit for everyday situations is the *character*, which is derived from the fact that all information can somehow be written down.

In any information-handling system, there is always available a fixed 'lexicon' of characters. The use of any character outside this set is not permitted. For example, my typewriter has some 88 different signs including the upper- and lower-case Roman alphabet, the 10 decimal digits and various punctuation and mathematical symbols. In writing this book, I must keep within this set; I cannot use Greek letters or integral signs since there is no way to type them.

Other common character sets differ in size. For instance, the set used to send telegrams or punch DYMO tapes only has some 40 different symbols, while the set available to printers is much larger and includes italics, bold type, and characters of various heights. The point remains that the number of different characters in any set is *finite*—that is, there is a limited number of distinguishable signs that can be used.

At first, this limitation may seem a drawback, but this is not so. A message or text composed of characters cannot convey any meaning unless the sense of each character is known both to the sender and to the receiver. If the number of possible characters were unlimited, the sender would be able to make them up as he pleased; but since the new characters would not be familiar to the receiver the message as a whole would be nonsense. The character set must form a *code* known to all who use it, and it must be finite to do so.

Quantities of information are measured in characters. (The artificial word 'byte' is sometimes used instead, and has the merit of being shorter.) For instance, the Bible contains about five million characters, while a typical medical record, pruned of all unnecessary data, might be found to comprise some 8000 characters.

The properties of information machines are also expressed with this unit; one speaks of a data link running at 10 characters a second, or a disc store of 90 megabytes. The prefix 'mega' implies multiplication by 10^6, so that such a disc could store 18 books each the size of the Bible.

Computers are a kind of information machine. They are not the only machines that exist for this purpose, but others like printing presses, sorters and calculators are intended for one application only, whereas computers are designed with great flexibility, so that they can rapidly be switched from one type of job to another, including printing, sorting and calculating.

The various ways in which a computer can process information correspond quite closely to the methods that a human might use. Some of them are given in table 2.1.

As well as similarities, there are important differences between machines and people. Three of them are speed, accuracy and cost.

The speed of a computer in doing simple operations is several million times faster than that of a human. Machines can undertake work that would be impossible or useless if done by people. A good example is found in meteorology. A weather forecast is only useful if it is available before the weather conditions actually arise. There are certain methods by which good 24-hour forecasts can be made from a knowledge of the current state of the weather over a large area. These methods have been known for a long time, but in the past they could not be applied because the calculations involved would have taken a man at least three years. Instead, forecasters tended to use simple rules of thumb, which did not always give correct results. Recently a computer that can do the sum in three hours has been built, and the precision of 24-hour forecasts has improved considerably as a result.

Human activity	Computer activity	Comments
Filing	Information storage	The amount of data stored by a typical computer is quite small—about the same as a couple of shelves in a library
Searching	Searching	Searching through a batch of data for a particular entry
Checking	Checking	Comparing two items of information to see whether they are *exactly* the same. Computers are not nearly as good as people in spotting whether two items are *roughly* similar
Sorting	Sorting	Rearranging groups of items into alphabetical or numerical order. This is an important preliminary to searching
Calculating	Calculating	Selecting certain items of information as numbers and doing sums with them
Writing out reports, etc.	Printing	Producing the results of a given piece of work

Table 2.1

Accuracy is not an attribute common to many people. Humans are very prone to errors, particularly when doing boring and repetitive jobs. On the other hand, machines do not get tired and bored and, while they do occasionally make errors, they do so far less often than people. This means that machines can be used for problems that humans cannot undertake purely because of the 'boredom' factor—such as solving 1000 simultaneous equations in 1000 unknowns, or reading through an entire telephone directory to find all the people who live in a certain street.

The cost of information processing by computer is already much less than that of manual processing, and it is falling further every year. The bulk of information handling is already done by computer. In principle this helps to relieve a large part of the human race of the daily load of drudgery that was so characteristic of earlier industrial times. In practice it may well give rise to the greater evil of unemployment. This is part of the indirect cost of using computers to solve information problems.

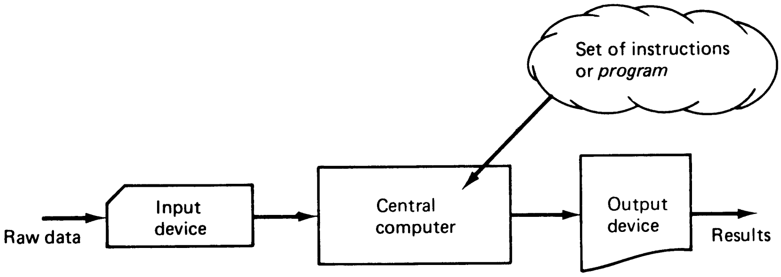


Figure 2.1

Figure 2.1 is a generalised picture of a simple computer system. The computer in the centre is fed with the raw material or *data* for the problem. The data are a stream of characters, which are entered through an input device that converts them into the electrical signals needed by the computer. The input unit can be a typewriter keyboard operated directly by a human, or it can be a device to read information that has already been recorded on some mechanical medium such as cards, paper tape or magnetic tape.

The results of the process also consist of a stream of characters, and they are generally converted from their electrical form and printed by an output device. This again can be a teletype (something like an automatic electric typewriter) or a line printer, which prints a whole line at a time and is very much faster. The results can also be displayed on a screen, or they can control a graph plotter and make it draw a picture, or they can be recorded on a mechanical medium so that they can serve as input to the machine on a later run.

The key to the process itself is the set of instructions or program, which controls the transformation of the data into the results. The program, which always consists of a sequence of relatively simple steps, is written in advance by a human programmer and 'plugged in' to the computer whenever a problem of that particular kind is to be solved. In terms of the ideas set out in chapter one, the programmer is the implementor, or means, by which the solution is actually obtained in any instance. The program may or may not be an algorithm.

To illustrate these ideas, some examples of processes of this type are given in table 2.2.

In all these examples, everything needed to solve the problem is to be found either in the data or in the set of instructions. Unfortunately only a small fraction of practical information problems fit this pattern precisely. Most of them need access to some more or less permanent collection of facts or *database* to provide essential background information.

Program name	Typical input	Typical output	Comments
'Decode'	MTNOT IAAWR WMICS SOENK EERNA OCNET TNTDG SXXXX	MAINATTACKON NWSECTORSEND TWOREGIMENTS	Program to 'unscramble' a military code.
'Simultaneous equations'	3X+2Y = 47 5X+4Y = 90	X = 4, Y = 17.5	Solves simultaneous equations
'Calendar'	1977, 1984	Calendar for 1977 and 1984	Generates a calendar for any specified year

Table 2.2

Typical databases consist of such material as telephone directories, railway timetables, diaries for executives in large businesses and other organisations, and stock records. The services supported by these databases range from straightforward enquiries ('Which train should I catch from Exeter to arrive at Norwich by 5.00 p.m.?') to complex commands like, 'Find the first possible opportunity to hold a two-hour meeting, starting at 10.00 a.m. or after, to be attended by the chairman and any three or more of Messrs Able, Baker, Charlie, Dog and Early'.

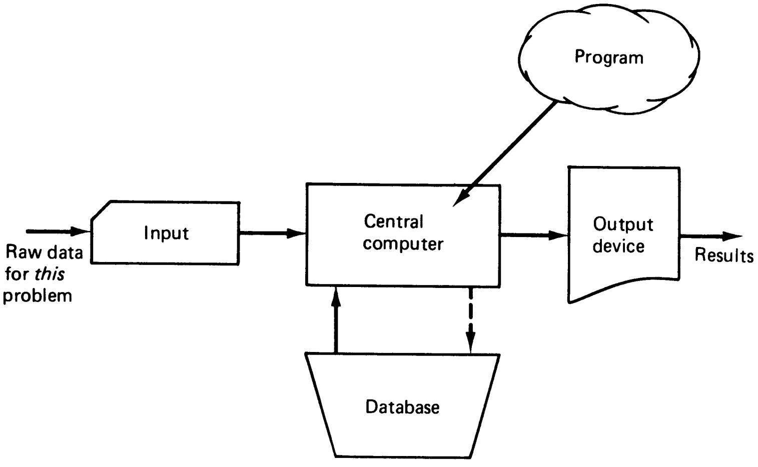


Figure 2.2

The over-all picture of a computing system with a database is shown in figure 2.2. The connecting lines show that in most cases the information flows from the database to the central computer itself. If any information flows the other way at all, its quantity is relatively small.

In many simple instances, the database has the form of a *file*. A file is a collection of records, each one containing information about some specific entity. For example, a university might maintain a file of its graduates and other ex-students. Each record would correspond to one ex-student, and would contain his or her name, status (Mr, Miss, etc.), address, year of graduation, profession, and subject taken at university. Figure 2.3 shows an extract from the file, which in practice might hold some 100 000 entries at any time.

. . . .

SAMUEL WELLER, MR, 2 EXMOOR RD, LYNTON, 1970, SOLICITOR, LAW
SARAH GAMP, MRS, 12 FINCHLEY RD, LONDON NW3, 1968, CHEMIST, PHARMACY
REGINALD WILFER, MR, 1 BEDLAM AVE, BIRMINGHAM 7, 1920, AUTHOR, PHYSICS
CHARITY PECKSNIFF, MISS, 123 BELSIZE RD, LONDON NW3, 1972, TEACHER, MATHS
MARTIN CHUZZLEWIT, DR, SOAP CRESCENT, BATH, 1965, DOCTOR, MEDICINE

. . . .

Figure 2.3 The original database

Once such a file has been set up, the university can use it for a number of different purposes. Some examples are as follows.

(a) The university might wish to substantiate a claim that it produces more engineers (or doctors) than any other. The computer can help by running through the file and counting the number of people in the various professions.

(b) The university might wish to arrange a reunion of old students. To ensure that the people invited have a good chance of knowing one another (and also to restrict numbers to a practical level) it could be decided to limit the issue of invitations to those who graduated at about the same time, say, during three consecutive years. The computer would help by searching all through the file, picking out the ones who were eligible, and printing their names and addresses on labels, which would be stuck directly to envelopes containing copies of a suitable circular letter.

(c) The university might find itself in financial trouble. It would of course appeal to its old students for help. The computer could pick out those whose age and profession made them the most likely to respond and print out personalised letters, duly signed with a facsimile of the Principal's signature. In the following, the personalised sections are underlined; they would be different for each recipient.

> Dear Mr Weller,
>
> Since you left Hardgate University in 1970 there have been many changes, some not altogether desirable, to our financial status. We are being increasingly squeezed between a falling income and rising maintenance costs.' In particular the law school where you spent so many happy hours is gravely in need of redecoration and repair.
>
> Samuel, we need your help in raising the money to keep the University going. Although any amount is acceptable, we feel that you, as a solicitor with several years of practice behind you, could hardly contribute less than £100 to our funds.
>
> Thanking you in advance,
>
> Yours very sincerely,

Of course, a file of this kind would soon lose its usefulness unless the information in it were kept up to date. The pretext of running a graduates' magazine makes an excellent way of collecting news about changes in professions, marriages and deaths. Once a year, all this information would be collected and punched on cards, together with the names and other particulars of new graduates. This data would be used to 'update' the ex-student file; the new graduates would be represented by new records, reports of deaths would cause some of the existing ones to be deleted, and other information would lead to changes in records already on file.

As already mentioned, every job done on a computer is controlled by a program. Every program is initially prepared by a programmer and consists of a suitable sequence of simple steps or instructions. In the remainder of this chapter, the basic nature of these instructions will be examined and some of the elementary operations needed to implement the processing of information will be discussed.

To begin with, it is not hard to see that the machine must be able to handle characters. All data is made up of characters, and so these are, in a sense, the basic units of information.

Secondly, the central processor must have a working store in which information can be retained on a temporary basis. This feature — the need for 'scratch-pad' storage—seems to occur not only in computers but in almost every information-handling situation. Thus a researcher working in a library will keep a pile of paper slips on which to jot down his findings. Again many desk calculators are proudly advertised as being built 'with memory', the implication being that those without memory are very inferior (as in fact they are).

In a computer, the memory serves to retain information over entire sequences of commands. The memory consists of a set of locations, each capable of storing one character. The actual storage is done electrically, but there is no harm, if you like, in imagining the store as a long thin blackboard, divided into squares. You must imagine that each square is permanently labelled with its own number, and is big enough to contain exactly one chalked symbol.

There is an important distinction between working storage on the one hand, and the kind of memory needed to retain databases on the other. Working storage is essentially *volatile*; this means that it can be used to hold information during the running of a program but once a run has ended the working store of the machine is wiped clean and handed over to the next job. By contrast the *backing store* of a computer, which is used to hold files and other databases, is intended to be safe from one run of a program to the next. In a well-built system there is no risk of one job unintentionally wiping out a database created by another.

This difference tends to be reflected in the physical properties of the two types of memory. The working store of a computer is purely electronic (which makes it very fast) but it is liable to lose the information it holds if the machine is switched off for any reason, for example, by a power cut. It is also very expensive and therefore relatively small—tens or hundreds of thousands of characters.

The backing store has the opposite characteristics. Information is recorded magnetically on the surface of a tape or disc, in a place determined by mechanical motion. This makes it slow but cheap, and the information is quite safe even if the machine breaks down. Most large machines have backing stores of hundreds of millions of characters.

So far it has been established that the central processor must include a working store, which can contain characters. Clearly nothing can actually happen unless the characters can be moved or manipulated in some way. The following are some of the basic character operations that the machine must be able to do.

(a) Copy a character from one location to another, erasing any previous contents of the destination cell. For example, if the instruction said 'copy cell 15 to cell 16', and initially cell 15 held an X and 16 a Y, then after the operation both cells 15 and 16 would hold an X, and the Y would have been lost irretrievably.

(b) Read a character from the input stream and put it into a specified location, overwriting the previous contents. When this instruction is obeyed, the computer sends a request to the input device to read one character (or a row of holes in a card, or a key depression). When the character eventually appears as an electrical signal it is duly recorded in the selected cell.

(c) Take the character currently in a selected location and print it. The contents of the chosen cell in the workspace are sent to the output device for printing or display.

(d) Print a specific character (that is, one known in advance).

(e) Compare the character in a given cell with a given character and determine whether they are the same.

It is worth noting that these instructions refer sometimes to 'a given character' and sometimes to 'the character in a given cell'. The difference is important. The given character is always known in advance and is often called a *constant*, but the character in a given cell is not always predictable and may change as the program runs, and so it is called a *variable.*

These instructions are so simple that if computers were restricted to running through any sequence just once they would be of little practical use. There must be some way of ensuring that a group of instructions, which does a particular job, can be repeated as many times as necessary.

A vital property of computers is their ability to count and to do arithmetic. Unfortunately arithmetic with single characters is not a feasible proposition—for example, there is no one character that can express the result of adding 5 and 7. To overcome this problem all computers are equipped to deal with another type of object called an *integer*, which means a whole number.

Integers may reside in store locations just like characters. They can be read, printed and compared in much the same way, but they can also be added, subtracted, multiplied and divided in the ordinary arithmetical sense.

An integer inside a computer is invisible. If the program prints its value, the value always appears as a sequence of characters like+3184. Inside the computer, the value is not represented as a series of characters at all, but in some other notation, which is more suited to arithmetic. The internal representation is of little interest at this stage, except in so far as it imposes limitations on the range of numbers that can be handled. This point will be discussed later.

Characters and integers are just two of the types of object (or *modes*) that a computer can handle. There are several other 'standard' modes, some of which will be introduced in later chapters. Furthermore, in some systems it is possible to make up special modes if for any practical purpose none of the standard ones is entirely suitable.

All computers, without exception, are capable of handling characters and integers, and of obeying a sequence of commands that specifies some program. The notation or *language* used to represent this program forms the subject of the next chapter.

EXERCISES

2.1 (0) Name the working substances of 10 machines familiar to you.

***2.2** (0) What is the 'character set' of an electronic desk calculator? (Consider the display only.)

***2.3** (0) Magnetic tapes are available in standard lengths of 200, 600, 1200 and 2400 feet. A certain type of tape deck can record 800 characters per inch of tape. What size of tape would you buy to store the text of the Bible?

2.4 (1) Describe a system in which a computer uses and maintains a database. Do not use the example given in the text!

2.5 (P) Find out as much as you can about your local computer, and make a table that shows its maker, price, speed and other properties.

3 *Introduction to Algol* 68

'For then I will turn to the people a pure language'

Zephaniah 3: 9

When a computer program is written down it must follow some agreed set of conventions if it is to make sense. Such a collection of conventions or rules is called a *programming language.*

The history of computer science has seen the development of hundreds of different languages. As the art of programming became better understood, newer and more convenient languages gradually replaced the older ones. The most important ones at present are

Fortran and Algol 60	for scientific work
Cobol	for business applications
Basic	for simple problems
PL/I and Algol 68	for general-purpose use (scientific and business)

A language always has two divisions — syntax and semantics. Syntax means the same as grammar, and determines whether any purported 'sentence' is grammatically correct or not. For example, an appeal to the syntax of English would reveal that

'The moon is made of Camembert'

is a correct English sentence, whereas

'Mary eated her little lamb for lunch'

is not.

Similarly, reference to the syntax rules of ordinary algebra shows that

$x+/^2y$

is not a conventional algebraic expression.

The syntax of a language says nothing about the meaning, still less the truth, of any sentence; it can only be used to check whether the sentence conforms to the rules of grammar for that particular language.

Semantics, on the other hand, *are* concerned with meaning. The only sentences to which semantics apply are those that are syntactically correct —

the ones with grammatical errors are automatically meaningless. Traditionally the semantics of a language are explained informally, with references to ideas that the reader already knows and many examples.

In studying Algol 68 in this book, discussions of both the syntax and the semantics of this language will frequently arise. As in the grammar of a natural language, the syntax rules are always arbitrary. They represent a personal choice by the language designers, and do not need to be justified on any other grounds. For instance, the language prefers **begin** and **end** to **start** and **stop**, and this point needs neither argument nor understanding, but only your acceptance. On the other hand the semantics — the meanings of statements in the language — are intended to make very good sense indeed and, as you study this aspect, you should make every effort to understand the purpose of this or that construction.

Algol 68 is written with a specific character set that includes the following.

all lower-case letters
all underlined lower-case letters (in printed material such as this book they are usually indicated by bold type instead)
the ten decimal digits and the decimal point
the arithmetic symbols + − ∗ / ↑
the relational symbols = ≠ < >
the brackets ( ) [ ]
the separators , ; |
the colon : and the double quotes " and " . The quotes mean the same which ever way they are printed.

An Algol-68 program is deemed to consist of one long string of symbols. They can be arranged on lines in any arbitrary way, and spaces can be put in almost anywhere without making any difference to the meaning of the program. This freedom is useful in arranging or 'laying out' the program to make it readable for other people. The rules of good layout will be considered in due course.

As in a natural language, the symbols of an Algol-68 program are grouped into words. The number of words that are, so to speak, permanently in the dictionary is really quite small but, as will become apparent, there is every opportunity for the programmer to invent and define new words as required.

The words fall into four categories.

(1) *Identifiers*: Identifiers are used to distinguish objects of all kinds: characters, integers, constants, variables, formulae and so on. An identifier must start with a *letter*, and then continue with letters or digits only. You can invent identifiers as you please, and most programmers use names with some sort of mnemonic significance. Examples are *count*, *prime*, *factor*, *cell3*, *cost*, *zed37p4q*.

(2) *Literals*: Most programs need to use numbers in various places. When these numbers are known in advance they are written as literals. There are two main ways of writing them

(a) if a number is an integer (that is, a whole number) it is written as a sequence of decimal digits: for example, *3* or *417009*

(b) otherwise, it is written as a series of decimal digits giving the integral part, then a decimal point, then another series of digits giving the fractional part: for example, *3.5* or *0.18007* or *123.001.*

There are other rules for writing numbers that are very small, or very large, but these can be left for consideration elsewhere.

(3) *Character literals*: Again, most programs need to use characters and strings of characters, if only for labelling the results. A character literal is written as a symbol enclosed in double quotes—for example "*A*" or "*:*"—and a string literal comprises several symbols

"*THIS IS A STRING*"

It is important to note that inside a character or string literal a space is significant, and stands for itself. This is the only exception to the general rule that spaces are ignored.

There is some difficulty in putting the double quote symbol itself into a character constant. Presented with, say

"*HERE IS A DOUBLE QUOTE SYMBOL:*" *SEE?*"

as an intended string constant the computer would naturally take the end of it to be at the colon, and would regard SEE?" as an error. To allow a double quote to be used inside a character or string literal the language includes the rule that in such a literal a double quote may be represented by two double quotes side by side. The correct form of the example is

"*HERE IS A DOUBLE QUOTE SYMBOL:*" "*SEE?*"

So as to make programs easier to read, it is conventional to write letters inside character and string literals as capitals.

(4) *Symbols of constant meaning*: These are the symbols of which the meaning is fixed by the rules of the language. They comprise three groups

(a) the single symbols + − * / ↑ = ≠ < > ( ) [ ] , ; | : $

(b) certain compound symbols, made up of two or more single symbols:
<= >= := +:= /:= −:= *:= :=: |:

(c) system words, which consist of sequences of underlined letters. A full list can be found elsewhere but some of the important ones are

int char real bool proc mode if then else elif fi case in esac begin end for from to by do od while co comment ref

Certain of the system words play the part of brackets. The matching pairs are

(	)
[	]
begin	**end**
if	**fi**
case	**esac**
do	**od**

Brackets of various types may be nested inside each other, but every opening bracket must be matched by a closing bracket of the correct type. When a program is set out, its readability is improved if the following rules about brackets are observed.

(i) If possible, a closing bracket (of whatever type) should be on the same line as its corresponding opening bracket.

(ii) Otherwise, a closing bracket should be vertically below its corresponding opening bracket, and the material in between should be indented so that all of it falls to the right of the vertical line joining the two brackets.

The brackets in an Algol-68 program play an extremely important part in determining its meaning, and this arrangement helps both the reader and the writer to match up opening and closing pairs correctly.

The syntax of Algol 68 permits certain contractions. Thus the pairs **begin** . . . **end**, **if** . . . **fi** and **case** . . . **esac** may all be replaced by the round brackets ( ). The replacement must be consistent: **if** cannot be matched with).

A 'skeleton' example of a properly indented program is

```
begin   ——

   ——

   do  ——  od;

   begin   ( ( — ) — )
             ——
   end

end
```

The symbol **comment** is used to set off commentary that is intended for the human reader only. Comments consist of remarks that explain what the program is doing, and they form a useful aid to you and anyone else reading the program (including, of course, your course tutor). Commentary can be inserted between any two system words in a program. It must start with the symbol **comment** and also end with this symbol (not **tnemmoc**), and it may include anything except **comment**, for obvious reasons.

The symbol **co** can be used instead of **comment**, but only in a consistent way; a section of commentary that starts with **co** is not terminated by **comment**, but only by another **co**; and the converse is also true.

For certain practical reasons the Algol-68 systems available on various computers all differ slightly from the language as defined in this chapter. For example, many machines rely on punched cards for input, and the character set that they offer includes one sort of capital letters only and does not permit underlining. The designers of the individual systems have had to find a way round this difficulty, and they have adopted different solutions.

In Algol 68R, the ICL 1900 system written by the R.R.E. at Malvern, identifiers are written in capitals and system words are enclosed in primes, thus: 'BEGIN', 'IF'. A further important difference of the dialect is that **do** is written 'DO' (and **od** is written), while **elif** is written 'ELSF', **co** is written 'c' and ÷ is written '/'.

The next step is to begin to discuss the design, structure and meaning of programs in Algol 68. The first few examples will seem trivially simple, but they are worth careful study, since they illustrate certain principles that are fundamental to all programming.

All programs are written so that they may eventually be submitted to a computer for running. When this is done, the program and its data must be punched up on cards or paper tape and enclosed in a *job description*, just as a letter has to be written out and enclosed in an envelope, which is then addressed and stamped. The job description *is not part of the program*, any more than the envelope is part of the letter it contains; its only purpose is to ensure that the program as a whole is correctly handled by the computer.

The correct form of a job description varies widely from one make of computer to another, and even when using the same machine different installations tend to maintain their own local conventions. In practice, your course tutor will tell you how to write a job description for the particular machine you are using, and you will probably be able to use the same job description for all the programs you submit.

An example of a complete job is given on the next page. The program is in the Algol-68R dialect, and the job description is suitable for an ICL 1900 machine running the OXCAF subsystem under GEORGE 3. The form of the job description that you will have to use may be quite different.

You will see that job-description statements are needed

(a) at the head of the job
(b) at the end of the program
(c) at the beginning of the data
(d) at the end of the job.

Since the job description is not part of the program, and is in any case not written in Algol 68, there will be no further mention of it in this book. Any reference that may be made to a 'complete program' will *not* include the job description.

```
JOB CAFANDREW, :CAQP47    ⎫
ALGOL68                   ⎬  Job description (initial part)
*ALGOL68                  ⎪
EXAMPLE                   ⎭
'BEGIN' 'INT' A, B, C;    ⎫
   READ (A);              ⎪
   READ (B);              ⎬  Program (in the Algol-68R dialect)
   C: = A+B;              ⎪
   PRINT (C)              ⎪
'END'                     ⎭

'FINISH'    ————————————     Job description (which terminates
                             program)
*DATA       ————————————     Job description (introduces data)
37    46    ————————————     The actual data

*END                      ⎫
ENDJOB                    ⎬  Job description (terminates whole
****                      ⎭  job)
```

Consider again the example given above. Rewritten in the standard Algol 68 notation, it is

```
1  begin int  a, b, c;
2     read (a);
3     read (b);
4     c:=a+b;
5     print (c)
6  end
(data)  37    46
```

Note that the line numbers are not part of the program but are included, here and elsewhere in complete programs, for reference only.

According to the conventions of the language, spaces and ends of lines have no significance. Remember also that **begin** and **end** are matching brackets; they always enclose a number of items that are separated by semicolons, so that the over-all structure of the program is

begin -----; -----; -----; -----; ----- **end**

The items between the brackets are *phrases*, or parts of the program. The computer carries them out, one by one, in the order in which they are written. Note that the semicolon is used only to separate one item from the next. It is wrong to put a semicolon before **end**, just as it would be unconventional, in writing a pair of coordinates, to put (5, 3,) instead of (5, 3).

The individual items include a declaration (**int** *a*, *b*, *c*) and four units (*read* (*a*), *read* (*b*), *c*: = *a*+*b* and *print* (*c*)).

It will be worthwhile to examine the effect of these items in detail.

The *declaration* makes the computer set up the working storage it needs for the problem. The system word **int**, which comes first, specifies that the storage required is for integers, not (for example) for characters. Next there follows a list of variable identifiers separated by commas. In the present example *a*, *b* and *c* were chosen for the sake of brevity but longer names would have been quite acceptable, since the programmer is free to choose whatever names he likes. For each identifier in the list the computer sets aside a storage cell for an integer, and notes the special connection between the identifier and the cell.

Strictly speaking the cells are selected from the working store in a way that is of no concern to the programmer, since he cannot influence the choice or even discover exactly what it is. In practice the system is likely to proceed in a simple way, taking cell 1 for *a*, cell 2 for *b*, and so on.

Figure 3.1 shows the state of the working store when the machine has obeyed the declaration.

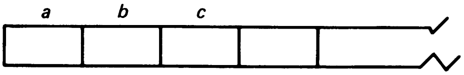


Figure 3.1

Note that although the cells have been set aside and identified, they do not yet actually contain anything – the technical expression is that their contents are *undefined.*

The next item in the sequence is *read* (*a*). This is a command to make the machine read a piece of data from the input stream and put its value into the cell associated with variable *a*. Since *a* has been declared as an integer, the machine scans the input stream, reading characters until it has got a complete integer. It puts the value into cell *a*, not using the two characters 3 and 7, but using instead the internal integer representation of 37. Diagrammatically, the working store will now be as in figure 3.2.

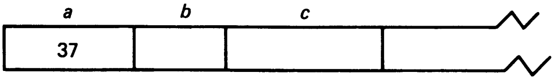


Figure 3.2

The system has now read part of the data stream; the next character available will be the space that follows the 37.

The next item is *read* (*b*). The machine obeys this command in much the same way as the previous one, but only begins to scan the input stream at the point where the preceding *read* ended. The value 46 is read into the variable *b*, and the working store is as in figure 3.3.

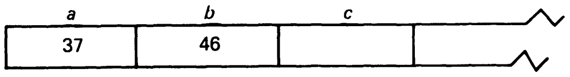


Figure 3.3

The next unit is *c*: = *a*+*b*. This is recognisable as an assignation because it consists of a variable identifier (*c*) and an expression (*a*+*b*) separated by the standard symbol : = (which is pronounced 'becomes').

To obey the unit, the computer works from right to left. First, it evaluates or 'works out' the expression. Since *a* and *b* currently hold the values 37 and 46, the result must be 37+46, or 83. Then it records this value in the cell of the variable mentioned on the left, or *c*. This produces figure 3.4.

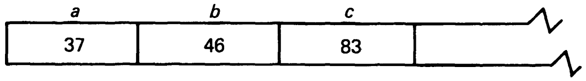


Figure 3.4

The last item is *print* (*c*). Up to now, all the commands have been happening inside the computer itself and there is nothing to show externally. This instruction makes the computer take the value of *c*, convert it to a sequence of decimal digits, and send them to the line printer. Since *c* is an integer, the printed result obtained is something like

```
+83
```

To summarise, here is what the computer has done

(a) It has reserved three cells from the working store to use as integers, and has attached identifiers to them.
(b) It has read two numbers from the data, putting their values into two of the cells.
(c) It has calculated the sum of these two numbers, and stored it in the third cell.
(d) It has printed the value in the third cell.

In short, it has added two numbers.

In the example the two numbers taken were 37 and 46, but the summary suggests that the program should work equally well on any two numbers. This conjecture is easily verified by a few trials. The program turns out to have some measure of generality, since it solves not just one problem (37+46 = ?) but a whole class of problems of a similar kind. This generality is an important feature of all useful programs.

The example illustrated four kinds of phrase—a declaration, a read command, an assignation and a print command. Since these phrases appear in practically every Algol-68 program, they merit consideration in somewhat greater detail.

In its general form, a declaration always starts with a system word giving the *mode* of the variables to be declared. Two modes that have already appeared are **int** and **char**. The mode is followed by a list of variable identifiers, separated by commas if there are two or more. The identifiers can be chosen quite arbitrarily provided that they do not clash with one another. Some further examples of declarations are

int *quick*

char *tiger*, *lion*

It is usual, although not essential, to place declarations immediately after **begin**. It is in order to write several declarations one after the other, including two or more of the same mode. If you forget to declare a variable, you do not need to alter an existing declaration but only to insert a new one. This is usually much quicker.

Declarations are phrases and therefore separated from one another by semicolons. A possible program structure might be

begin **int** *camel*; **char** *mole*; **int** *rat*, *beaver*; ----; --- **end**

The effect of the *read* command depends entirely on the mode of the variable being read. If it is a **char**, then the machine simply takes the next character in the input stream, whatever it may be. If the previous character was the last in a line or card the computer moves automatically on to the next line.

If the variable is an **int**, the system reads the input stream, character by character, looking for the beginning of a number and ignoring such things as spaces and the ends of lines. A number can legitimately start with a +, a − or a decimal digit. Of course it is possible for the machine to find some other character, which may have been punched by mistake, or it may run off the end of the input stream altogether. In either case, the machine reports a dynamic fault, and stops the program from running any further. The whole question of faults is discussed at the end of this chapter.

Once the beginning of a number has been found, the machine reads it and converts it into its internal form, stopping when it comes to a character that cannot be part of a number. Space and end-of-line are allowed in this position, but any other character is interpreted as a mistake and produces a dynamic fault. The simple rule in preparing integers as data for a program is

either put the integers one on á line or card
or put several on each line but separate them only with spaces.

The assignation is particularly interesting because it involves an expression. An expression is an instruction to evaluate something. The rules for expressions in Algol 68 are modelled as closely as possible on those used in ordinary algebra, and any differences arise chiefly because of the limited character set available.

An expression is built up from three types of symbol

operands, which specify the 'raw material' or starting values
operators, which indicate what is to be done with the starting values
brackets, which control the order in which the operators are used.

An operand may be either a literal or a variable. A literal stands for itself and can be written as a decimal integer or a character in double quotes, for example, 308, 9, or "Z". A variable, on the other hand, is written as an identifier and stands for the value stored in the corresponding cell at the time the expression is evaluated. The identifier must of course have been included in a declaration before being used in an expression.

Note that every operand has a certain mode. With variables this is the mode of their declarations, and with literals the mode is **int** for an integer and **char** for a character.

The simplest expressions contain one operand and nothing else, for example

3 or "*Q*" or *quick*

Other expressions include operators like + and −. Operators are of two types, dyadic and monadic. The difference is that a dyadic-operator takes two quantities as its arguments or data, whereas a monadic-operator takes only one.

Five dyadic-operators that use arguments of mode **int** are

+ (addition)
− (subtraction)
* (multiplication; the symbol × is not used because it is so easily confused with the letter x; Multiplication signs must be used wherever multiplication is intended; do not write *ab* meaning $a * b$, because of the risk of confusion with the single identifier *ab*)
÷ (division)
↑ (raising to the power; $x \uparrow y$ is used instead of x^y because there is no way, on a card punch, of writing superscripted text higher up than the surrounding characters).

In general, these operators have their ordinary arithmetical meanings. The division operator simply gives the integer quotient of the two numbers, and discards the remainder. Division by zero is forbidden, and if attempted will produce a dynamic fault.

To ensure that the result is an integer, the second operand of the ↑ must be zero or positive.

All the dyadic operators in Algol 68 are *infix*, that is, they are written between the operands they use just as in conventional algebra. Some expressions with one operator are

$x+1$ $p-q$ $37 * zj$ $k \div j$

These examples assume that *x*, *p*, *q*, *zj*, *k* and *j* have all been mentioned in a declaration like

int *x*, *p*, *q*, *zj*, *k*, *j*

Monadic operators in Algol 68 are always *prefix*, that is, they come immediately before the operands they are going to use. Some monadic operators that use integers are

abs this gives the modulus of a number: **abs** *5* = *5* but **abs** (−*7*) = *7*
sign this gives + 1, 0 or − 1 depending on whether the operand is positive, zero or negative
− minus at the beginning of an expression (or just after an opening bracket) is taken as monadic and simply negates its operand.

If an expression contains more than one operator the question arises as to which one should be applied first. The result will often depend on the choice of operators, as in the expression

3+*2*∗*5* (25 or 13?)

The conventional rule is that, if there are no brackets, multiplication and division must be done before addition and subtraction. Algol 68 adopts this rule and formalises it by the notion of *priorities*. Every operator in Algol 68 has a certain fixed priority, which ranges from 1 to 9 for dyadics and is always 10 for monadics. When expressions without brackets are evaluated the operators are used in descending order of priority, starting with the monadic operator if any. Since the multiplication sign rates 7 on the priority scale, but + only rates 6, the general rule ensures that multiplication does indeed take place before addition. The complete list of priorities for the operators already mentioned is

abs, **sign** and − (if monadic)	10
↑	8
∗ and ÷	7
+ and − (if dyadic)	6

Again conventional algebra decrees that any expression within brackets is evaluated before anything outside. This rule is taken over without modification by Algol 68. The brackets override the priority of the operators and ensure that any expression enclosed by them is evaluated first. If several sets of brackets are nested, evaluation proceeds from the inside outwards. Writing, for example

abs ((*x*+*y*)∗(*x*−*y*))

will ensure that the + and the − are used first, followed by the ∗ and finally the **abs**; as it happens, in the reverse of the defined order of priorities.

Suppose that $x = 5$ and $y = 7$. Then the expression would be evaluated in the following stages

```
  abs ((5+7)*(5−7))
= abs (12*(−2))
= abs (−24)
= 24
```

Now consider the expression

```
abs x+y*x−y
```

which is identical to the first one except for the brackets. The evaluation would proceed

```
 abs 5+7*5−7
=5+7*5−7
=5+35−7
=33
```

A common cause of confusion is an expression that contains two or more operators of equal priority. The general rule is that dyadic operators are used from left to right, but monadic operators from right to left. Thus *36 ÷ 3 * 2* is interpreted as *(36 ÷ 3) * 2* and evaluates to 24 (not to 6). On the other hand −**abs** *5* is taken to mean −(**abs** *5*), not **abs** (−*5*).

Brackets are free. You can afford to ignore the complex rules for deciding the order of evaluation if you use brackets wherever there is any possible doubt in your mind.

So far, all the operators discussed have used integers as operands. There are also some operators for characters, which will be described in the next chapter.

Just as each operand in an expression has a mode, so does the entire expression itself. The mode of an expression is the same as the mode of its result, which is usually (but not necessarily) the mode of the operands it uses.

In any assignation, the mode of the expression must be compatible with the mode of the variable on the left of the : = sign, that is, the variable into which the value is to be stored. It is wrong, for example, to try to assign a character value to a variable of mode **int**. The computer will reject a program that tries to do so.

In many assignations the variable on the left is also mentioned in the expression on the right. The effect is to replace the previous value of the variable by a new one. Some examples are

```
w: = w+1          co add 1 to w co
t: = t*s          co multiply t by s co
```

Algol 68 offers a compact way of writing statements of this type, using the special operators in the list below.

Operator	*Alternative Form*	*Pronunciation*
+: =	**plusab**	'plus and becomes'
−: =	**minusab**	'minus and becomes'
∗: =	**timesab**	'times and becomes'
÷: =	**overab**	'divide and becomes'

If any of these are used, the variable being altered need only be mentioned once. Thus

w:= *w*+*1* **can be written** *w*+: = *1*

and

t: = *t* ∗ *s* may be expressed *t*∗: = *s*

Some of the advanced rules about these operators are rather complex. For the present, use each special operator in a phrase by itself and avoid mixing them with the ordinary operators such as +, −, or : =.

Finally, consider the print command. Its action depends on the mode of the object that it is given to print. If this object is an integer variable or literal, then the print statement will output a decimal number. If it is given a character variable or a character literal, it will output just that character.

The system keeps track of the number of symbols already printed on the current line. If there is not enough room for the next object, a new line is started automatically.

When printing the results of a program, it is important to set them out neatly and to label them so that their significance is clear. The print command offers two useful facilities for doing this

print (*newline*)	will start a new line, irrespective of whether the previous line is full or not
print ("*A STRING*")	will print out the given string literal, no matter how long it may be (up to a full line).

Thus

print ("*HEL LO*")

is a quick way of saying

print ("*H*"); *print* ("*E*"); *print* ("*L*"); *print* ("*L*"); *print*("*O*")

The following is an example of a program that incorporates some of these ideas; it is based on an old folk saying, that a man should marry a wife of half his age, and seven

```
begin int mansage, girlsage;
   read (mansage);
   girlsage: = 7+mansage ÷ 2;
   print ("A MAN OF"); print (mansage); print (newline);
   print ("SHOULD LOOK FOR A WIFE OF"); print (girlsage)
end
```

Programming is always hard. While you are learning you will find that each new example or problem will stretch your knowledge a little further by involving some new ideas. Later, when you have mastered all the basic ideas, you will look back and find these same examples easy; but by then you will be tackling much bigger problems, which are hard by virtue of their size and complexity. The task of writing a program and getting it to work correctly will always remain difficult, laborious and extremely satisfying when successfully completed.

To help you in this task there is a range of aids and techniques. Some of them depend on using the computer, but others are purely pencil-and-paper methods and can be done at your desk.

You will often be given a program and asked to find out precisely what it does. Sometimes it will be a program written by someone else, but usually it will be one of your own programs, which is supposed to do one thing but actually, and unexpectedly, does something different. By finding out exactly what it does do, you can generally pinpoint the phrase at which it goes wrong.

It is usually impossible to tell what a program does just by looking at it. You must *trace* it, or go through it step by step, pretending that you are the computer.

Start with a clear sheet of paper, a pencil and a ruler. Draw two boxes, one for the input stream and one for the output stream. Write the characters of the input stream, in order, into the appropriate box. Then work through the items of the program, as follows.

The first section of any program usually consists of the declarations. Use them to draw a picture of the working store, labelling each cell with its identifier, and also with its mode if there is any risk of confusion. In general, variables can take a whole series of different values during the running of a program; when you record these values, do not erase the previous ones but simply cross them out, so that at the end of the process you have a complete trace of the values taken by each variable. In the picture of the working store it is best to allow plenty of room for each variable (as in figure 3.5).

The rest of the program may contain *read*, *print* and assignation statements, as well as other types, that have not yet been considered. In order to trace a *read* statement, refer to the input stream box and strike off the characters one by one until enough have been collected for the item that you require. (Naturally you must use the characters in order, and you do not take those that have already been crossed off.) You then write the value of the item into the appropriate cell, crossing out any previous value but leaving it legible!

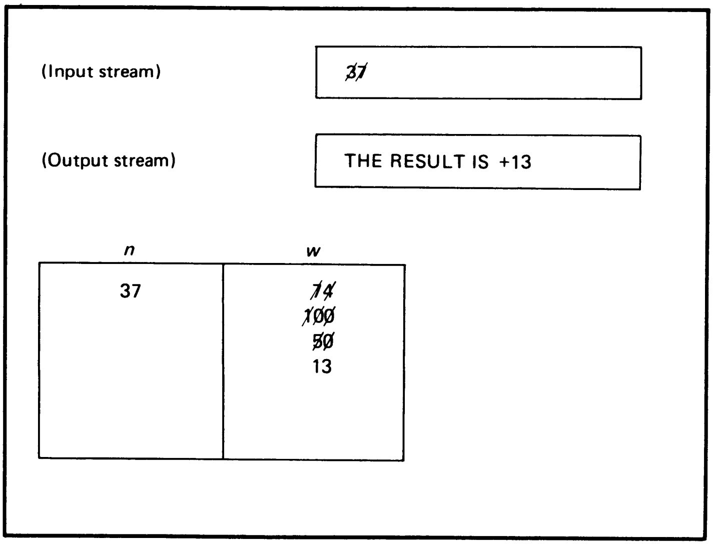
A *print* statement is traced by taking the value of the variable (or the string literal) and writing it into the output stream. Nothing else is changed.

An assignation statement is handled by working out the expression on the right of the : = symbol, using the current values of the variables. When this value has been obtained it is written into the cell identified on the left, crossing out the previous value if any.

The following is a program to be traced

```
begin int n, w;
   read (n);                     co Think of a number      co
   w: = 2 * n;                   co Double it              co
   w + : = 26;                   co Add 26                 co
   w ÷ : = 2;                    co Halve it               co
   w − : = n;                    co Take away the number
                                        you first thought of  co
   print ("THE RESULT IS");      co Tell me the answer     co
   print (w)
end
```

The final version will be as shown in figure 3.5 below.



(Input stream) ~~37~~

(Output stream) THE RESULT IS +13

n	*w*
37	~~74~~ ~~100~~ ~~50~~ 13

Figure 3.5

Another powerful set of aids is provided by the computer's fault-detection system. When a program is wrong, it is nearly always returned by the computer with an *error message*. Unfortunately these messages are often expressed in a strange language, which looks like English but fails to convey much sense as to what is actually wrong.

The cryptic quality of the error message is not altogether the fault of the system designer. A computer is an exceedingly complex machine. The facilities described in this book are only a small subset of the rules governing its use. Nevertheless, when the computer finds an error, it assumes that the rule as a whole has been broken and prints a message accordingly. It is rather as if you were to commit some trivial and unconscious misdemeanour like forgetting to renew your driving licence, and to find yourself charged with

'An offence under the Motor and Pedestrian Controlled Vehicles Act, 1974'

You might be forgiven for not immediately understanding what it was that you had done wrong.

In actual fact the messages are quite easy to interpret if you know what to look for.

When it is handling your job the computer can pass through three different stages, each one producing its own characteristic error reports. First, the machine analyses your job description. If it finds a mistake it abandons the job immediately without even considering the program at all. If, in your results, the program is not even printed out you can assume that your job description was wrong. Check it (with someone else if you cannot find the mistake yourself) and submit the corrected job again.

If it gets past the job description the computer will begin to analyse your program, checking the syntax and translating it into its own internal language. It will print out the program as it goes, attaching a number to each line and reporting any faults as it finds them. The fault messages themselves are all obscure and are best interpreted as 'I found something wrong round about here'. There is no need to give examples because if you are following this course seriously you are certain to have plenty of your own!

Fortunately most syntax errors stem from only a few basic causes, and if you are familiar with them the pointers provided by the error messages will help you find the mistakes quite quickly—sometimes in a few seconds. The most common slips in programs are as follows.

(1) Leaving out semicolons. This is extremely common whenever the next item happens to begin on a new line.

(2) Putting in extra semicolons. Semicolons may only be placed *between* items: never before **end** or a closing bracket.

(3) Not matching brackets (of whatever type) correctly. This fault can be avoided by careful layout, following the rules given in this chapter. It is often detected much further on than it actually occurs. Consider the assignations

$$a := 4 * (a+b) * (a-b); f := \text{“}T\text{”}$$

and suppose that the first closing bracket is omitted, giving

$$a: = 4*(a+b*(a-b); f: = \text{“}T\text{”}$$

The machine cannot find anything wrong with this version until it meets the semicolon, and so this is where it reports the fault. In this example the gap between the error and the report is only a few characters but in practice it is often dozens or even hundreds of lines.

(4) Misspelling *variable* identifiers. If an incorrect spelling occurs, say, in an assignation, the system will take it as a new variable that has not been declared. If the misspelling is in the declaration itself, the fault will not be detected until the first correct spelling that occurs. The following example illustrates the point.

```
begin int boys, gurls;  Fault actually occurs here
   -------;
   -------;
   read (boys);
   read (girls);        Fault reported here
   --------;
   print (boys);
   print (girls);       and here
end
```

(5) Not declaring identifiers. This error is easily found with the clue given by the fault message.

(6) In a dialect where system words are enclosed in primes (like Algol 68R) a common mistake is to leave either or both of them off.

(7) Leaving out double quotes on character literals. If the opening double quote is missing, the computer will probably take the literal itself as an undeclared identifier. If the closing quote is missing the machine is liable to take the whole of the rest of the program as a very long string literal, and not to report any errors until the very end.

(8) There is a group of faults that arises from incorrect expressions. It is a mistake, for example, to omit the multiplication sign, or to write two dyadic operators side by side without an operand in between, or to omit an expression altogether leaving the : = symbol hanging in the air.

(9) Finally a common source of errors lies in the use of variables of unsuitable mode. It is wrong to try to multiply two characters or to assign a value of mode **int** into a character variable.

Sometimes the errors in a program are so involved that the system is, as it were, put off its stride; although it continues checking the program, it reports lots of other errors where there is none. This is particularly common after mistakes with matching brackets and the situation is easily recognised with practice.

If the computer finds any errors of syntax in a program it abandons the job at the end of the translation-and-checking phase, before obeying any of the instructions of the program itself. If there are no errors of syntax the system goes on to the final stage in which it actually executes the program. This phase can have one of three outcomes

(a) The machine is required to do something illegal (like dividing by zero) and this creates a dynamic fault, stopping the run. On the whole the dynamic-fault messages tend to be rather more helpful than the ones generated by the syntax analysis. They may actually say

'Division by zero'

or

'Integer raised to a negative power'

or

'End of input stream reached'

or they may produce reference numbers, which can be looked up in a table that is freely available and describes the faults in clear English. In many systems the current values of the working variables are printed out as well.

(b) The machine prints a set of answers that are not the ones you expect.

(c) The computer actually produces the results you hoped for.

In case (c) the program may be correct, but you cannot assume so unless you have tested it thoroughly in all the conditions in which it can be used. In the other two cases you will have to find the mistakes. This is usually harder than finding syntax errors. One method is to trace the program by hand, and some other useful methods will be discussed later. The best method is not to let dynamic faults or wrong answers happen in the first place! This can often be achieved by careful programming planning.

These remarks show that a computer system is very rigid unforgiving and even arrogant. It never excuses you for the smallest mistake, and it never tries to make an 'intelligent guess' at what you actually meant, even though it may have been quite obvious to a human reader.

Ideally programs should be correct when first submitted. In practice this rarely happens. With practice, and care, people can usually design algorithms correctly, but hardly anyone is capable of writing them out as programs without making any mistakes at all. A good practical aim for simple programs is to use just three computer runs: the first to find and correct all the errors of syntax, the second to check the results and to make any minor adjustments to the layout that may be needed, and the third as the final correct run. If you are taking much more than three runs for a program you are not putting enough care into finding and correcting errors. Do not fall into the common trap of only looking for and correcting one error at a time, hoping that the rest will somehow go away!

EXERCISES

```
begin [1:20] char word; int z; [1:10000] char dict; int pd:= 1;
[1:1250] struct (int back, for, count, pdict, size) tree;
[1:80] char line; int lp:= 100, tp:= 1;
proc charfetch = char: begin char q; if lp > upb line then
clear line; read((newline, line[ ])); print((newline, line));
lp:= 1; “0” else q:= line[lp]; lp+:= 1; if q > “Z”or q < “A”
then “0” else q fi fi end;
proc nextword = bool: begin char j; while j:= charfetch; j = “0” do
skip od; word[1]:= j; z:= 1; while j:= charfetch; j ≠ “0” do
z:= z + 1; word[z]:= j od; word[1:4] ≠ “ZZZZ” end;
proc nodeout = (int x): begin int j = pdict of tree[x], k = size
of tree[x]; print((newline, count of tree[x], space, dict[j:k+j−1]))
end;
proc treeprint = (int x): begin if back of tree[x] ≠ 0 then
treeprint(back of tree[x]) fi; nodeout(x); if for of tree[x] ≠ 0
then treeprint(for of tree[x]) fi end;
proc newnode = void: begin back of tree[tp]:= 0; for of tree[tp]
:= 0; count of tree [tp]:= 1; pdict of tree[tp]:= pd; size of tree
[tp]:= z; dict[pd:pd+z−1]:= word[1:z]; pd+:= z; tp+:= 1 end;
proc make = void: begin int j, k, w; bool f: if not nextword then
print((newline, “NO TEXT”)) else newnode; while nextword do
j:1; f:= true; while f do k:= pdict of tree[j]; w:= size of tree
[j]+k−1; if word[1:z] = dict[k:w] then f:= false; count of
tree[j] +:= 1 elif word [l:z] < dict [k:w] then if back of tree
[j] ≠ 0 then j:= back of tree else f:= false; back of tree[j]
:= tp; newnode fi elif for of tree[j] ≠ 0 then j:= for of tree[j]
else f:= false; for of tree[j]:= tp; newnode fi od od
treeprint(1) fi end; make end
```

Figure 3.6

Figure 3.6 is a correct but poorly laid out Algol-68 program.

*3.1 (1) Make lists of all the ‘words’ used under the various headings.

*3.2 (2) Copy out the program omitting lines 2 to 19 inclusive, and indenting the rest correctly.

3.3 (P) Get your course tutor to give you a complete Algol-68 program. Punch it on cards or tape and run it through your computer.

3.4 (0) Rewrite the program on p. 27 using different (and longer) identifiers for the variables.

3.5 (1) Sketch the working store of a computer after it has obeyed the declarations

```
int oliver, david, philip;
char twist, copperfield, pirrip;
int charles dickens;
```

***3.6**(1) Find two mistakes in the following set of declarations:

```
int susan, pat, emily;
char fred, bill, pat, bob;
int ethel, susan
```

***3.7** (2) Assume that the variables j, k, t and m hold the values 4, 6, 7 and -1 respectively. Evaluate the following expressions

(1) $j+1$ (2) $j*k$ (3) $t \div j$
(4) $m \uparrow k$ (5) $j*k+t$ (6) $t+j*k$
(7) $-k$ (8) $k \div t*j$ (9) **sign** m
(10) **abs** $(t-j)$ (11) $(t-k)\uparrow(t+k)$ (12) $(10-k)*(10+k)$
(13) **abs**(**sign**$(k-j)$ + **sign**$(k-t)$ + **sign**$(k-m)$)

***3.8** (1) Find as many faults as you can in the following program.

```
begin chair x, y;    int w, t, x;
   read (w);
   t: = abs (w+3t-47 ÷w;
   y: = "Q"
   y: = t
   print ("THE ANSWER MAY BE);
   print(t)
end
```

3.9 (1) Trace the program given on p. 35 to decide the correct age for your wife. (If you are a girl rewrite the program so that it calculates the best age for your husband, and then trace it to find out what that age should be.)

***3.10** (1) Trace the following program, using the data stream MOOD and state in general terms (but as briefly as possible) what it actually does.

```
begin char a, b, c, d;
   read (a); read (b); read (c); read (d);
   print (d); print (c); print (b); print (a)
end
```

***3.11** (2) (P) Write and run a program that reads two integers a and b, and prints out $a+b$, $a-b$, $a*b$ and $a \uparrow b$ all suitably labelled.

4 *Conditionals*

'. . . Personally, I never feel that I understand a thing satisfactorily, however fully it may be proved mathematically, unless I also have a clear mental picture of it.'
'Cathode Ray', *Wireless World*, February, 1937

For practical purposes a computer is an abstraction. True, there are certain physical events — such as the flow of electric currents — that are the basis of its function, but they are almost as remote from the actual steps of a program as the switching operations of our individual brain cells are from our thoughts. Philosophers are not generally neurologists, nor are computer programmers required to be experts in electronics.

Most people find it hard to think in entirely abstract terms and prefer to use some kind of picture or mental model of the system that they are considering. To be useful, the model must fulfil two requirements. First, it must mirror some important aspects of the system faithfully and closely; and where it does not, the differences must be so obvious that there is no risk of anyone being misled. Second, the model should be so familiar that it is possible to 'see' in the mind's eye how it works, and what is likely to happen in any given situation. If these two conditions are fulfilled, then the insights obtained by considering the model can be transferred directly to the abstract system.

A useful model of a computer program is a railway. The track represents the sequence of instructions, and the train running along the rails models the operation of the computer as it obeys the instructions one after another. Consider the following program (which was first used in chapter 3).

```
begin int a, b, c;
   read (a);
   read (b);
   c: = a+b;
   print (c)
end
```

The corresponding railway system is shown in figure 4.1 Note that the track starts at a station called **begin** and runs to another station called **end**. There is only one engine, so that there is no need for any signals. The track is divided into a number of sections, one corresponding to each phrase in the program. The engine always starts at **begin** and runs forward until it gets to **end** (in the model it never runs backwards). As it passes the sections in sequence it 'obeys' the phrases it finds there. Like the computer, it has no choice anywhere along the line; it can only pass over existing sections of the railway, in exactly the sequence in which they are put together.

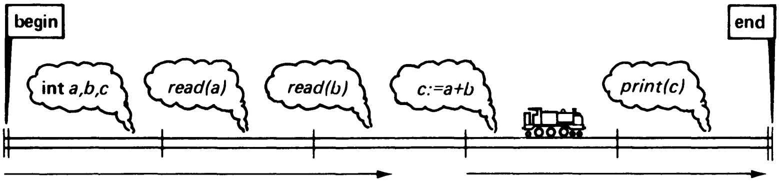


Figure 4.1

Most practical railway systems have branches and points, and the model is no exception. There are two ways in which a train can approach a point, as shown in figure 4.2.

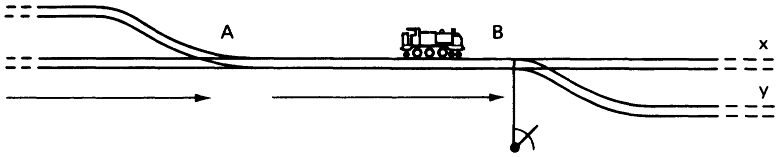


Figure 4.2

When an engine approaches point A from the left there is no possibility of choice; it must continue towards the right. When, on the other hand, it gets to B a decision must be made as to whether to continue straight on towards x or to turn out and go to y. This decision is always made on the basis of information available when the train is in the section before B, and is implemented by pulling the point lever attached to B one way or the other. The person making the decision could be either the engine driver (although he would have to stop his train to switch the point) or a signalman who was specially in charge of the junction. Fortunately the distinction is not relevant to the model. All that matters is that a decision is made when the train is already on its journey, on the basis of the most recent information that can be obtained. Since the railway designer does not know in advance what the decision will be, he provides a set of points to make the choice possible.

The ability to choose one of several possible courses of action is a vital property of any computer system. It enables a machine to handle whole ranges of problems in a flexible way and raises it from the status of a robot blindly following a set of fixed instructions to a level such that it can be made to deal 'intelligently' with various unexpected happenings.

In Algol 68 the choice of a particular path of action relies on the use of boolean-expressions, which provide a way of allowing the computer to ask questions with 'yes-or-no' answers about the current values of the variables in its working store. In its simplest form a boolean-expression consists of two quantities — usually a variable and a constant — linked by a *relational symbol.* Consider a program that begins with the declarations

int z; **char** *next*;

Suppose that it also includes the boolean-expression

$z > 5$

(pronounced '*z* greater than *5*'). This expression would have the value 'yes' or **true** if the current value of *z* was 6 or more. Otherwise (if *z* was 5 or less) its value would be 'no' or **false**. Similarly, the program might include the boolean-expression

next = "T"

which would only be **true** if the cell labelled *next* actually contained the character " *T*". If *next* contained any other character the value of the expression would be **false**.

There are six relational symbols

=	(equals)
≠	(not equal to)
>	(greater than)
<	(less than)
> =	(greater than or equal to)
< =	(less than or equal to)

It is worth remembering that an integer cannot be compared directly with a character; it is illegal to try.

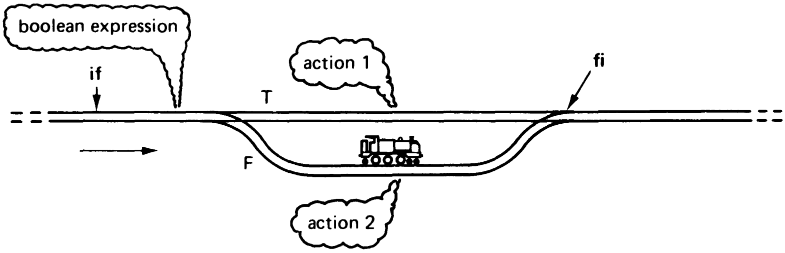


Figure 4.3

The actual selection of a course of action is made by the **if** construct. In general terms, this can be written **if** boolean-expression **then** action 1 **else** action 2 **fi**

Here **if, then, else** and **fi** are all system words, and **if** and **fi** are matched brackets. The machine takes action 1 if the expression is **true**, or otherwise action 2.

The railway equivalent of the **if** construction is shown in figure 4.3. Note that the boolean-expression is attached to the section of track immediately before the first point. It is always preceded by a warning **if**, which is a signal that a decision must be made. If it has the value **true** when the train passes, the point is set to let the train go straight on to the section marked T. If the expression has the value **false** the train is turned out instead to the section marked F. At the second point, marked **fi**, the two sections join to form one track again.

The following is a simple program that incorporates the **if** construction. It might be used to prepare accounts by a company that made and sold teapots. The price of the teapots is £3 each, or £2 if bought in quantities of 100 or more. The program reads in the number of teapots in any one order and prints out the total cost.

```
begin int qty, cost;
  read (qty);
  if qty < 100 then cost:= 3 * qty else cost:= 2 * qty fi;
  print("YOU HAVE BOUGHT");
  print (qty);
  print ("TEAPOTS.  YOU OWE US  £");
  print (cost)
end
```

The railway model for this program appears in figure 4.4. The **if** construction has a number of useful variants. It would be tedious to explain each one in turn, at great length, but the discussion can be given a much shorter and neater form by defining the terms *unit* (which was used loosely in chapter 3) and *serial-clause*. These are powerful generalisations, by means of which all the versions of the **if** construction can be set out using only two definitions.

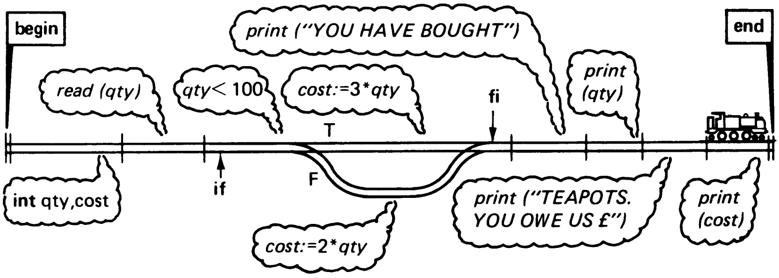


Figure 4.4

In chapter 3 the term *phrase* was used to include both a declaration and a unit, and the examples given there suggested that a unit could be a simple command like *read* (*a*) or *c*: = *a*+*b*. At this point it is worth introducing two other types of unit.

(1) A boolean-expression (or for that matter any expression) is a unit.

(2) Any piece of program enclosed in brackets (round or **if**–**fi** or **begin**–**end** or **case**–**esac**) is also a unit. Thus the entire **if** construction in the previous program, starting with **if** and ending with **fi**, is just a unit. It follows the rules in being separated from the neighbouring unit by semicolons. Furthermore, the entire program (and indeed every program) is a unit!

On the railway system it is convenient to regard a unit as a section with one way in and one way out. It may be just a simple piece of track with a command attached to it, or it may include some points provided that all the branches join up before the exit point is reached. It may even be an extremely complex system; but, provided that it conforms to the rules and has one entrance and one exit, it may still be regarded as a single section.

A serial-clause turns out to be a sequence of declarations and units separated as necessary by semicolons. The declarations are optional but there must always be at least one unit. An example of a serial-clause is

```
int a.b; a: = 1; b: = a+2; print (b)
```

One unit is sufficient in itself to constitute a serial-clause, so that another example is

```
a+3
```

In the railway model a serial-clause is represented by a number of sections connected end-to-end. The train can only pass them in the order that they are built.

In some cases it is sensible to think of the value of a serial-clause. The value is the same as that of the last (or only) unit in it, and if that clause is a boolean-expression the value must be either **true** or **false**. Such a clause is called a serial-boolean clause.

The proper definition of the **if** construction may now be given. It is written

```
if serial-boolean clause
    then serial-clause 1
    else serial-clause 2
fi
```

This definition has several useful implications. First note that each of the two possible actions is a serial-clause. This means that whole sequences of units may be introduced between **then** and **else** and between **else** and **fi**; in other words, the actions can be as complicated as desired. For example, the program to determine the best age for your prospective wife could be modified somewhat as follows

```
begin int mansage, girlsage;
  read(mansage);
  if mansage < 16
    then print("AT"); print(mansage);
         print("YOU ARE TOO YOUNG TO MARRY")
    else girlsage: = 7 + mansage ÷ 2;
         print("A MAN OF"); print(mansage);
         print("SHOULD LOOK FOR A WIFE OF");
         print(girlsage)
  fi
end
```

Next, remember that an **if** – **fi** construct is itself a unit, and therefore a simple type of serial-clause. If the decisions to be made are complex one condition can be nested inside another. The following example is taken (very properly) from a British Rail brochure in which various concessionary fares for the 'Intercity' network are advertised. It appears that the discount on a return ticket depends on the length of stay, as shown in Table 4.1. Furthermore, children aged 3 to 13 travel at half-fare, and children under 3 go free.

Length of stay (days)	Discount on full return fare	
1	50 per cent	Day return
2 or 3	30 per cent	Week-end return
4 to 17	25 per cent	17-day return
18 or more	0 per cent	Period return

Table 4.1

The program that follows could be used to work out the actual fare for any Intercity return journey. So as to avoid dealing with decimals, a topic not yet considered, all calculations will be in pence, which are alwayş whole numbers. Fractions of pennies are simply ignored.

The program will read the following data

(a) the full return fare, in pence
(b) the length of stay, in days
(c) the age of the traveller, in years.

```
begin int frf, stay, age, af, fare;
  comment frf is the full return fare;
          af is the adult fare for the journey;
          fare is the actual fare
  comment
 read(frf); read (stay); read (age);
```

```
    comment First the adult fare is calculated according to
                the length of stay
    comment
    if stay = 1
      then af: =frf*50÷100
      else if stay<4 then af: =frf*70÷100
                     else if stay<18 then af: =frf*75÷100
                                     else af: =frf
                     fi
         fi
    fi;
    comment In the next stage, the passenger's age is taken into account, so
               as to obtain the actual fare
    comment
    if age <3 then fare: = 0
              else if age < 13 then fare: = af÷2
                               else fare: = af
                   fi
    fi;
    comment the last step is to print the answer comment
    print("YOUR FARE IS");
    print(fare)
end
```

The railway model that corresponds to this program is shown in figure 4.5.

There is a shortened form of the **if**–**fi** construct, which omits the **else** and the serial-clause that follows it, thus

if serial-boolean clause **then** serial-clause **fi**

The effect is that if the serial-boolean clause is **false** nothing is done. The railway system still includes a turnout, but no actions are attached to it.

The fare-calculating program given above represents only one of many possible arrangements. Another form, which makes use of the shortened form of the **if**–**fi** construction, is as follows.

```
begin int stay, age, fare;
      read(fare); read(stay); read(age);
      if age <3 then fare:= 0
                else if age <13 then fare÷:= 2 fi
      fi;
      if stay <4 then
                if stay = 1 then fare: =fare*50÷100
                            else fare: =fare*70÷100
                fi
                else
                if stay <18 then fare: =fare*75÷100 fi
```

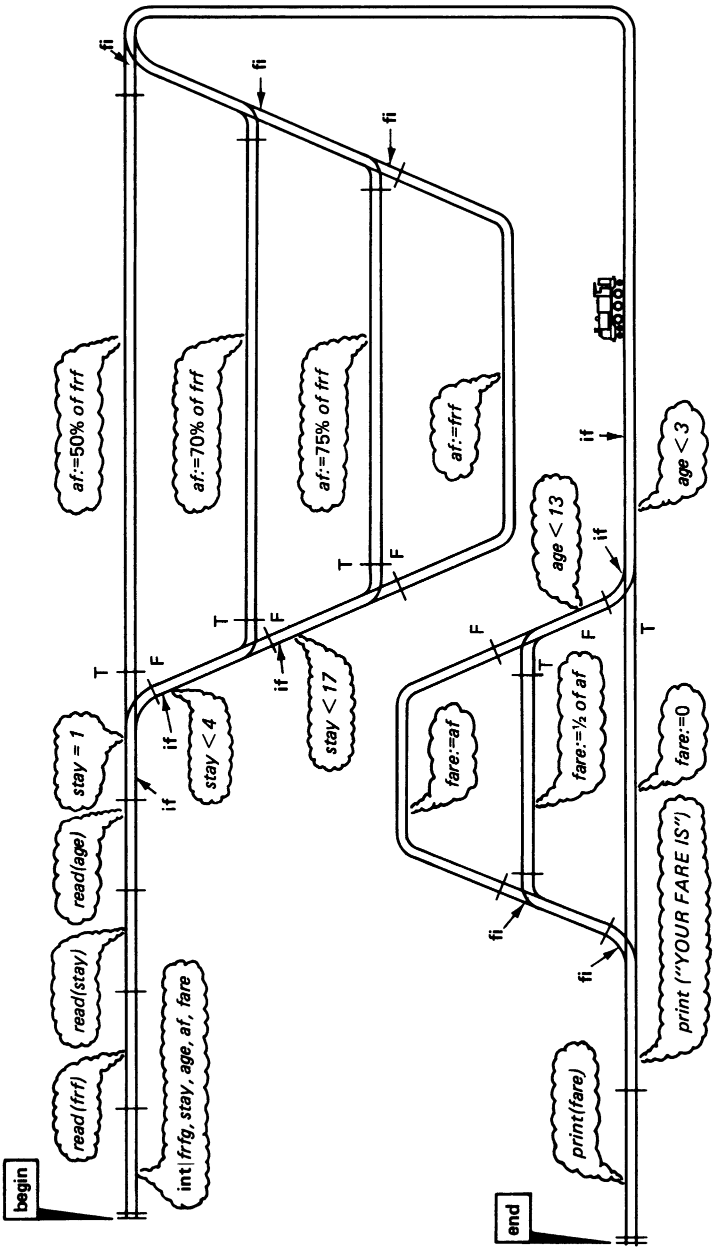

begin
read(frf)
read(stay)
read(age)
stay = 1
af:=50% of frf
int|frfg, stay, age, af, fare
if
T
F
fi
if
stay < 4
T
F
fi
af:=70% of frf
if
stay < 17
T
F
fi
af:=75% of frf
af:=frf
if
age < 3
T
fare:=0
if
age < 13
T
F
fare:=½ of af
F
fare:=af
fi
fi
print ("YOUR FARE IS")
print(fare)
end

Figure 4.5

```
        fi;
        print("YOUR FARE IS");
        print(fare)
end
```

Lastly, note that some units (like expressions) have *values*. It is not difficult to write an **if** – **fi** construction (which is a unit) that can be used instead of a value in an expression. All that is necessary is to ensure that both the actions are present, and that both of them yield a value of a suitable mode. For example, it is possible to replace

```
if qty < 100 then cost: = 3 * qty else cost: = 2 * qty fi
```

by the shorter but equivalent phrase

```
cost: = if qty < 100 then 3 else 2 fi * qty
```

but it is not acceptable to write

```
cost: = if qty < 100 then 3 fi * qty
```

This form is ambiguous; what would the computer use as a multiplier if the condition *qty* < *100* were **false**?

One feature that distinguishes a professional's program from that of an amateur is its *robustness*. This quality takes into account the evident fact that the program is going to be used by a human, with all the failings that this involves. For example, the programmer may specify that the data are to be in a certain format, or that they must obey certain conditions, but the person actually supplying the data may not follow the rules, perhaps because he misunderstands them or simply through carelessness. A number of errors of this kind are undetectable, because they could be valid data (for example, giving a person's age as 21 instead of 12). However, a great many more mistakes in the data can be trapped because they are logically wrong. A program that is robust should incorporate enough tests to ensure that the data represent a reasonable problem. If presented with invalid data, it should never simply give a wrong answer, but always print out a message describing the fault so that the user can repunch his data and try again.

A good example of lack of robustness is provided by the program on p. 48. The length of stay is supposed to be an integer of 1 or more; but what happens if the user actually supplies a zero? The fare will be calculated as 70 per cent of the full return fare, an obvious error.

A good plan in building robust programs is to check input values as they are read. If they should happen to be detectably wrong, then the program should print a message, and if necessary substitute a 'reasonable' value so that the program can carry on; this will allow further errors to be found.

With some degree of robustness engineered into it, the Intercity-fares program might now begin as follows.

```
begin int stay, age, fare;
    read(fare);
    if fare >10000 then print("FARE OVER £100 IS UNLIKELY") fi;
    read(stay);
    if stay <1 then print("STAY TIME INCORRECT"); stay: = 1 fi;
    read(age);
    if age >120 then print("AGE UNLIKELY") fi;
    - - - - - - -;
```

With a large program, it often pays to arrange that a section does not accept the result of an earlier section without first checking that it is correct or at least plausible. Consider a program that incorporates a variable *x*. The first part of the program is supposed to set *x* to one of the values 2, 7 or 33. The next part of the program uses this value to choose one of three courses of action.

If the writer of the second part were perfectly sure that the first part was correct and trustworthy, he could deduce that, whenever *x* did not hold 2 or 7, it must hold 33. He would write

```
if x = 2 then (action for x = 2)
          else if x = 7 then (action for x = 7)
                         else (action for x = 33)
               fi
fi
```

If the first part of the program was nevertheless wrong, and put into *x* an incorrect value like 29, the second part would do the action for *x* = 33, and the error would remain undetected. A more robust piece of work would make the second part of the program check the validity of the first, and print a warning message if an unexpected value was found. It would run

```
if x = 2 then (action for x = 2)
          else if x = 7 then (action for x = 7)
                         else if x = 33 then (action for x = 33)
                                        else print ("X = "); print (x);
                                        print ("THIS IS IMPOSSIBLE");
                                        print ("TELL A. J. T. COLIN")
                              fi
               fi
fi
```

In principle the section of code that prints the unexpected value of *x* could never be entered. *In practice* such traps frequently call the programmer's attention to unexpected mistakes and help to make programs more reliable. They are strongly recommended in all cases, but particularly when sections of the same program are being written by different people. Never trust any program, not even your own!

At this stage a brief discussion of the *costs* of computing would be appropriate. In practice, a program has a life history roughly as follows.

(1) Someone (usually a systems analyst) detects the need for the program and decides exactly what it is to do.
(2) A programmer actually writes the program and gets it working.
(3) The program is used, perhaps only once or perhaps thousands of times — possibly every day for several years.

All these stages have certain costs associated with them. About stage (1) little can be said at this point. The costs of stage (2) arise from two sources: the programmer's salary and the expense of using the computer. It is clear that, in general, the longer and more complex a program, the more it will cost to develop. Also a program on which a great deal of thought and care is lavished will usually cost more than one that is written with the minimum of effort.

The costs of stage (3) are incurred every time the program is run. The more instructions the computer has to do, the more the run will cost. If the program is to run a large number of times then over its entire life the bulk of the cost will lie here, but if it is used only a few times the cost of defining and writing it will constitute the greater part of the total.

In practice it is always possible to reduce the cost of running a program by taking careful thought and rewriting the program accordingly. Whether this is worth doing depends on how often the program is to be used.

EXERCISES

4.1 (1) Let *a*, *b* and *c* be declared as follows

int *b*, *c*; **char** *a*

Write boolean-expressions to test the following conditions

(a) *b* greater than 5
(b) *c* not less than 17
(c) *b* smaller than *c*
(d) 33 not equal to *b*
(e) *a* equal to "7".

***4.2** (1) Correct the mistakes in the following program

```
begin int x, y;
   read (x)
   if x < 100 then y = 4x else y: = 10x - 73;
   print (y)
end
```

4.3 (2) How many possible routes are there through the railway network shown in figure 4.5? Tabulate them, showing the conditions that correspond to each one.

4.4 (2) Calculate the length (in unitary clauses) of each of your routes in exercise 4.3.

***4.5** (3) Draw a railway diagram that corresponds to the second version of the Intercity-fares program. Calculate the length of each possible path and compare it with the corresponding length as calculated in exercise 4.4.

***4.6** (2) (P) Write a program that reads three numbers, interprets them as the lengths of three straight lines, and decides whether they can be assembled into a plane triangle. Your program should print the lengths and a clear statement of the result.

4.7 (2)

```
begin int a, b, c, p
    char x, why, zed
    read (a) read (b) c: = a+b
    if c = 25 then a: = 10 b: = 15 fi
    read (x) read (why) read (zed)
    if x ≠ zed
      then print (b)
      if x = why then read (zed) print (zed)
        read (a)
      fi
      print (a)
    fi
    p: = a+b+c
    print (p)
end
```

This program has all its semicolons missing.

(a) Draw rings round all declarations and units.
(b) Add semicolons as necessary

4.8 (5) The pension rules in a certain country state that a man receives 50 doubloons a week if he is over 65 and an extra 20 if he is over 70. A woman receives 45 doubloons a week if she is over 60 with an extra 25 if she is over 65.

Write a program that reads in a person's sex and age, and prints out the appropriate pension. The data are supposed to take the form of a character (M or F) followed by an integer.

Your program should be *robust*, and it should be made as *efficient* as possible (that is, costing the least when all runs are taken together) by using the following information.

	Male 65–70	Male over 70	Female 60–65	Female over 65
Percentage of all pensioners	10	25	15	50

5 *Loops and Program Efficiency*

'So
Round about
And *round* about
And *round* about and *round* about
And *round* about
And *round* about
I go.'

A. A. Milne, *Now We Are Six*

Every serious program must repeat some of its operations many times over. For example, a program that handles a file of employees and produces a set of payslips will apply the same set of rules to each employee record in turn.

The basic way of bringing repetition into an Algol-68 program is to enclose the part to be repeated (which is, as you might expect, a serial-clause) in the special brackets **do**–**od**. Consider, for example

```
begin
   do print(newline); print("FOR THE SNARK WAS A BOOJUM, YOU
      SEE")od
end
```

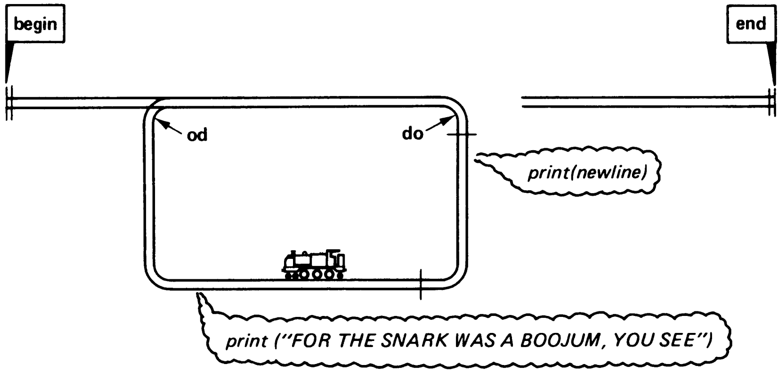


Figure 5.1

The snag is that the serial-clause in the **do** – **od** brackets would be repeated not once, nor twice, but indefinitely. It is like a railway with some missing track, as in figure 5.1.

As you can see (by tracing if necessary) the program will print

```
FOR THE SNARK WAS A BOOJUM, YOU SEE
FOR THE SNARK WAS A BOOJUM, YOU SEE
FOR THE SNARK WAS A BOOJUM, YOU SEE

. . . . . . . . . . .
```

and so on, without ceasing. In practice, the program will eventually be stopped by the operating system, which allows jobs just so many hundred lines of output before it suspends them with a dynamic fault. Programs of this kind are often called 'wallpaper' programs.

There are some applications in which an endless loop is actually needed. Two examples are a control program for traffic lights, and a computer operating system, which is intended to handle an endless stream of jobs. In most cases, however, the loop must eventually terminate; there must be a way for the train to arrive at **end**.

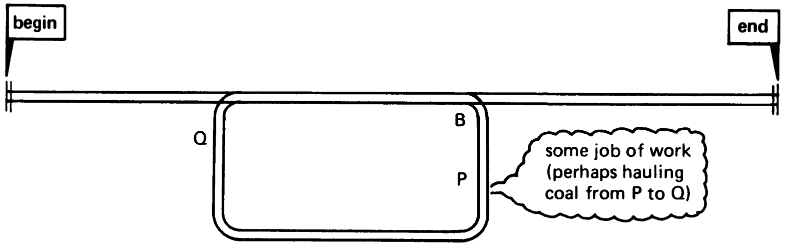


Figure 5.2

A railway layout that makes this possible is given in figure 5.2. Any train that starts from **begin** must eventually pass over the set of points B in a forward direction. A decision on whether to stay in the loop or not can be made just before the point is reached.

In Algol 68, a boolean-expression can be attached to this point by writing

while boolean-expression **do** serial-clause **od**

The serial-clause continues to be repeated so long as the boolean-expression remains **true**. The construct corresponds to that shown in figure 5.3. The whole structure has one entrance at x and one exit at y. It is therefore a unit, and should be treated as such.

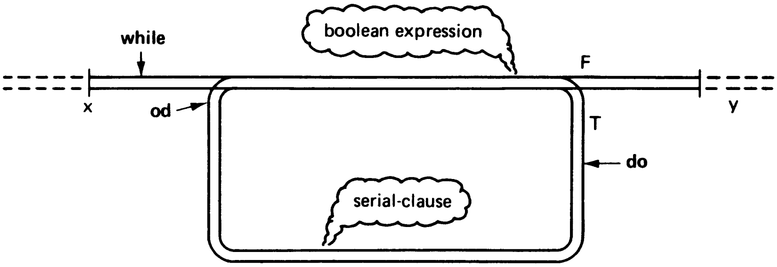


Figure 5.3

Consider a practical example. The program that follows is supposed to print the squares of the numbers 1 to 5; repetition is clearly involved.

```
begin int n, s; n: = 1;
   print("NUMBER SQUARE");
   while n < = 5 do print(newline); s: = n * n; print(n);
                      print(s); n + : = 1
                 od
end
```

In tracing the program, note that there are no read commands, and therefore no data. The program contains all the information it needs within itself.

Since the expression n < = 5 is in a key position, it will be given a column in the trace, even though there is no actual variable with this value. The trace has the following form.

(Input stream)

```
```

(Output stream)

```
NUMBER        SQUARE

  +1            +1

  +2            +4

  +3            +9

  +4           +16

  +5           +25
```

n	s	$(n <= 5)$
~~1~~	~~1~~	**true**
~~2~~	~~4~~	**true**
~~3~~	~~9~~	**true**
~~4~~	~~16~~	**true**
~~5~~	25	**true**
6		**false**

You are advised to work through this trace yourself, since the complete picture cannot show the order in which the various tests and assignments take place.

This program illustrates an important point: every serial-clause controlled by the **while – do – od** construct must do something that will eventually switch the boolean-expression from **true** to **false**, or the loop will never end. In the example, the statement $n +:= 1$, repeated over and over, ensures that n will eventually exceed 5 and make the program break out of the loop.

Loops that have to be repeated a fixed number of times are easy to program. Use an integer variable and initialise it to a suitable starting value (usually 1). Then, increment it by 1 in the body of the loop, and use the control expression to test whether it is still within the required range.

Sometimes the repetitions depend on some relationship between variables, and the number of circuits around the loop cannot be predicted in advance. A good example is Euclid's algorithm, which finds the highest common factor of two numbers. The algorithm specifies that given two numbers, the smaller be subtracted from the larger, repeatedly, until the two are equal; this is then the HCF.

The following is a program for finding the HCF of a pair of numbers, which are supplied as data.

```
begin int a, b;
   read(a); read(b);
   print("THE HCF OF"); print(a); print("AND"); print(b); print("IS");
   while a ≠ b do
               if a > b then a −:= b else b −:= a fi
               od;
   print(a) comment it would be equally correct to write print(b) comment
end
```

The following is the trace of this program for the data values 6 and 15

(Input stream) ~~6~~ ~~15~~

(Output stream)

```
THE HCF OF+6 AND+15 IS+3
```

a	*b*
~~6~~	~~15~~
3	~~6~~
	3

Note that the output states the *problem* as well as the answer. A program that merely printed

+3

or even

HCF = +3

would be almost useless; the casual reader would ask, 'The HCF of what?', and even the programmer himself could not be sure that his data had been punched and read correctly.

The decision as to whether to stay in the loop is often determined by some external condition. It is customary, for example, to arrange the data for a problem as a series of items (numbers or characters) ended by a special marker that is chosen so that it cannot possibly be an ordinary data item, and serves only to terminate the sequence. Thus a series of numbers all known to be less than 100 might use 999999 as a terminator, or a text consisting solely of letters and punctuation marks might be terminated by a slash (/).

Since the number of items is usually unknown to the programmer (and may even vary from one run of the program to the next) a program that reads such a sequence must be constantly on the look-out for the special terminator lest it run off the end of the data stream.

Consider the problem of adding up a series of numbers. The program would be expected to handle input data such as the following

```
1  3  7  12  15  21  999999
```

(the sum is 59) or

```
999999
```

(the sum is zero).

The first step in discussing this problem is to describe the basic mechanism for doing the job. This must then be elaborated to give a final version that uses a proper layout for its results and has the necessary properties of robustness.

The appropriate railway for the problem is shown in figure 5.4. It is not difficult to imagine the engine going round the loop, reading numbers and adding them to the sum *s*. Eventually it reads a 999999 and branches out to the terminal sequence.

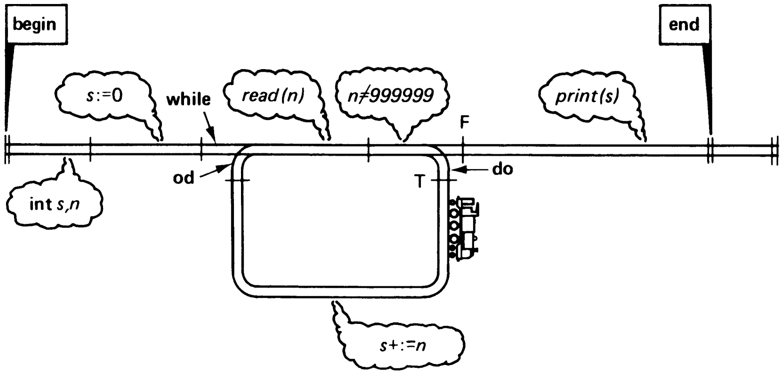


Figure 5.4

There is one essential difference between this system and the other programs that have been discussed in this chapter. Every time it goes round the loop the train must do something—namely, read the next data item—*before* it can decide whether to continue in the loop or not. In fact the *read*(*n*) part of the loop is traversed one more time than the rest of it.

Here once again the generality of Algol 68 proves useful. The full definition of the **while**–**do**–**od** facility says that **while** and **do** enclose a serial-boolean clause. Now any serial clause at·all becomes a serial-boolean clause if it has a boolean-expression as its last item, and so it is possible to write the following version of the program

```
begin int s, n; s: = 0;
   while read(n); n ≠ 999999 do s+: = n od
   print(s)
end
```

In practice, such programs are only useful if they deal with very many numbers indeed; if you only want to have 100 numbers added up it is quicker to borrow a pencil and do it yourself.

Large data streams are nearly always punched and checked by professional data-preparation staff. It is not practical for the program to print out all the numbers, since checking their accuracy by hand would be worse than adding them up in the first place. In any case, an occasional small error makes very little difference to the grand total, and in many cases such small inaccuracies do not matter.

There are, however, some types of error in the data that could make the results completely wrong. In spite of being checked, a card might be missing a space between two adjacent numbers, so that what should be

45 34

is actually punched as 4534 and will be read as a single number. Clearly, a few mistakes of this type could make a massive difference to the final total.

Another common blunder is to miss out part of the data altogether. If an operator drops your cards, and in picking them up manages to shuffle the 999999 terminator card into the middle of the pack, the program will ignore the second half of the data completely.

These considerations can suggest what safety features should be put into the program. Two simple checks will be used.

(a) Each item of data will be checked as it is read. If it is outside the permitted range of 0 to 100, the computer will print a warning message and refrain from adding it to the total.

(b) A tally will be kept of how many numbers are actually added into the total. Since the user of the program usually knows—at least roughly—how many numbers there should be, the tally will tell him if any of the data have got lost in processing.

With these amendments, and with a full set of layout instructions, the program is as follows.

```
begin int n, s, t; s: = 0; t: = 0;
   while read(n);
     n ≠ 999999 do if n > 100 then print("VALUE OUT OF RANGE");
                                   print(n); print(newline)
                              else if n < 0 then
                                   print ("VALUE OUT OF RANGE");
                                   print(n); print(newline)
                                            else
                                   s+: = n; t+: = 1
                                   fi
                   fi
               od;
   print("TOTAL = "); print(s);
   print("NUMBER OF NUMBERS USED IN FORMING TOTAL =");
      print(t)
end
```

In some loops, *all* the action has to be done before the test. Consider the problem of reading and copying a short message (in characters). The terminator, which is to be copied as well, is a slash. A suitable train system is shown in Figure 5.5.

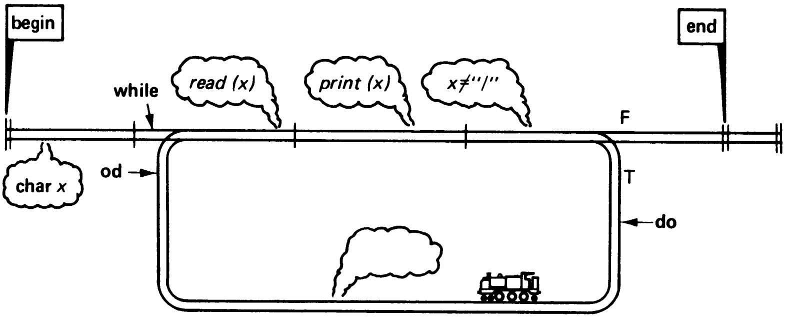


Figure 5.5

In coding the example you will find that there is nothing to put between **do** and **od**. Under these circumstances you can use the special system word **skip**, which is a unit meaning 'do nothing'. The result is as follows.

```
begin char x;
   while read(x); print(x); x ≠ "/" do skip od
end
```

Table 5.1 gives a summary of the three main variants of the **while**–**do**–**od** construction.

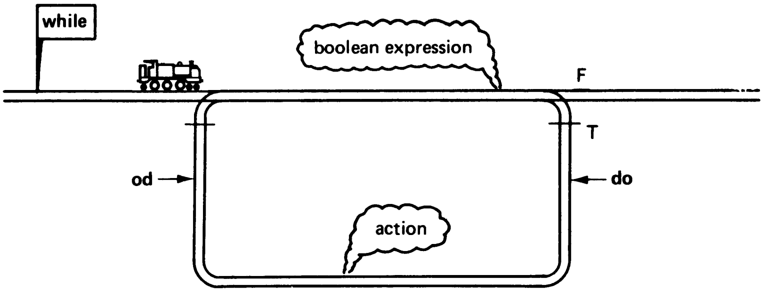
A common feature of many programs is their use of nested loops. In many jobs done by a computer, *each* passage through some major loop involves going round a minor loop several times. For example, consider a program that calculates the HCFs of each pair of numbers in a list. The input could be

```
  3   17
 27   18
 26   12
 31   29
 39   13
783    7
  0
```

(the zero can be used as a terminator since all genuine data values are supposed to be positive) and the corresponding output would read

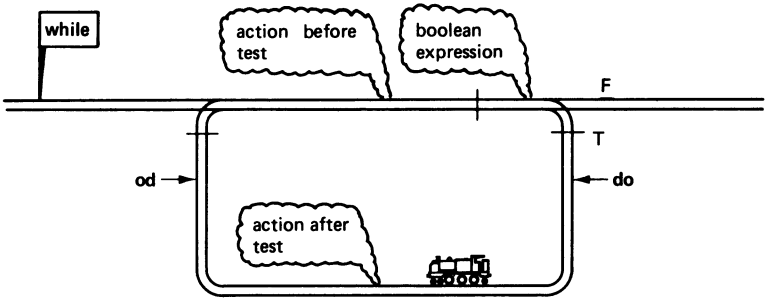
```
THE HCF OF   +3 AND+17 IS  +1

THE HCF OF  +27 AND+18 IS  +9

THE HCF OF  +26 AND+12 IS  +2
```

1. Test before any action



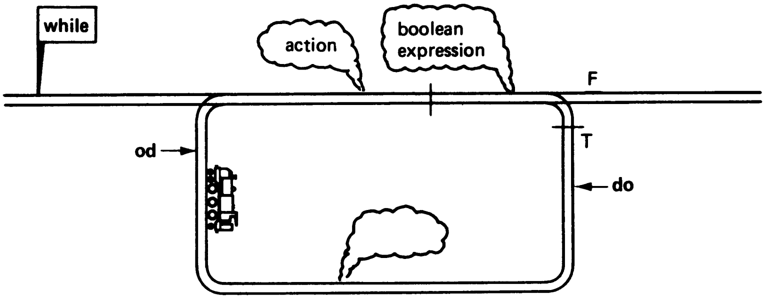
while boolean-expression **do** action **od**

2. Some action before test, some after



while action before test; boolean-expression **do** action after test **od**

3. All action before test



while action; boolean-expression **do skip od**

Table 5.1

```
THE HCF OF  +31 AND+29 IS  +1
THE HCF OF  +39 AND+13 IS +13
THE HCF OF+783 AND   +7 IS  +1
```

In Algol 68 the whole of a **while** – **do** – **od** construction is a unit, so that there is no difficulty in putting one loop inside another. To see how this is done, consider the first HCF program on p. 58, which handles only a single pair of numbers. Its general shape is

```
begin int a, b;
    read (a);  read (b);

              ┌──────────────────────────────┐
              │ instructions for working out and │
              │ printing the HCF of a and b      │
              └──────────────────────────────┘
end
```

What is required is to modify this program so that it loops round over and over again, each time reading new values of *a* and *b* and calculating their HCF. The loop must stop when $a = 0$. This can be done by using a **while** – **do** – **od** construct, as follows

```
begin int a, b;
  while read(a); a ≠ 0 do read(b);

                ┌──────────────────────────────────────┐
                │ instructions for working out and printing the │
                │ HCF of a and b (exactly as in the box above)  │
                └──────────────────────────────────────┘

                od
end
```

Substituting the section of program in the box gives the complete program

```
begin int a, b;
  while read(a); a ≠ 0 do read(b);
                ┌─────────────────────────────────────────┐
                │print("THE HCF OF"); print(a); print("AND");│
                │print(b); print("IS");                       │
                │while a ≠ b do                               │
                │             if a > b then a –:= b           │
                │                     else b –:= a            │
                │             fi                              │
                │        od;                                  │
                │print(a)                                     │
                └─────────────────────────────────────────┘
                od
end
```

The corresponding train track is shown in figure 5.6

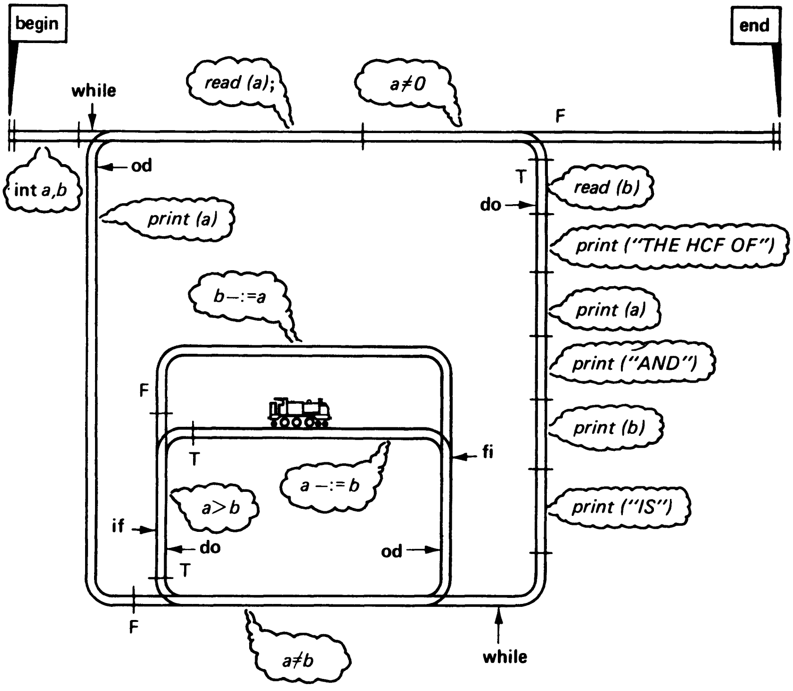


Figure 5.6

While on the subject of repetition, it is of interest to ask how many times each phrase in a program will be repeated. Consider a very simple program: one that just calculates and prints the multiplication tables up to 12 * 12.

```
 1 = 1          begin int x, y, z;
 1 = 1            x:= 1;
13 = 13           while x < = 12 do
12*1 = 12                        print (newline); print(x);
12*1 = 12                        print("TIMES TABLE");
12*1 = 12                        y:= 1;
12*13 = 156                      while y < = 12
12*12 = 144                      do print(newline); print(y);
12*12 = 144                         print(" * "); print(x); print(" = ");
12*12 = 144                         z:= x * y; print(z);
12*12 = 144                         y+: = 1
                                 od;
12*1 = 12                        x+: = 1
–                              od
–               end
```

Since it is known how many times each loop is obeyed, it is possible to calculate the number of times each phrase is actually carried out. The declarations and the assignation $x := 1$ are obeyed once. The **while** – **do** – **od** construct that follows is obeyed 12 times, once for each of the twelve tables. (This means that the expression $x < = 12$ is actually evaluated 13 times, 12 times to give the value **true** and once to give the value **false**). Next comes the serial clause enclosed in **do** – **od**, which starts with *print*(*newline*) and ends with $x +: = 1$. If the clause is considered on its own, these instructions are obeyed just once; but since the whole clause is actually executed 12 times it is necessary to multiply by 12 in order to calculate the over-all frequency of any phrase that it contains.

The inner loop, which corresponds to each line in the set of tables, is obeyed 12 times for each repetition of the outer loop, or 144 times in all. The control expression $y < = 12$ is actually executed 12 ∗ 13 or 156 times.

The columns on the left of the program listing show the frequency of each instruction, and the way that this frequency is calculated. The program itself is arranged so that all the phrases on any line are obeyed the same number of times.

An analysis of this program shows that

two phrases are obeyed once each
six phrases are obeyed 12 times each
one phrase is obeyed 13 times
eight phrases are obeyed 144 times each
one phrase is obeyed 156 times.

The total number of phrases obeyed is

$$2 * 1 + 6 * 12 + 1 * 13 + 8 * 144 + 1 * 156 = 1395$$

Of these, 1308—about 94 per cent—are in the inner loop!

This analysis points to a fundamental truth about all serious programs—the computer always spends the larger part of its time obeying the phrases in the inner loop of the program it is executing. The corollary is very simple: if your program takes enough computer time, and is to be run often enough, to be worth speeding up at all, then your efforts should be directed at the inner loop, since this is where nearly all the time goes. The outer sections can look after themselves.

In the example, about a quarter of the phrases could be saved by simply omitting the printing of the '∗' and ' = ' in the inner loop, and arranging to print a suitable heading instead. On the other hand, no amount of work on any part of the program outside the inner loop could possibly save more than 6 per cent of the phrases obeyed.

In many cases the number of repetitions of some part of a program depends on the data, and cannot be predicted in advance. A clear example is given by the second version of the HCF program on p. 64, where the frequency of the outer loop depends on the number of data pairs supplied, and the number of times around the inner loop depends on the values of the numbers in a rather complex way. The fundamental point remains unchanged—even if it is not known exactly how many times the inner loop is to be executed, it is certain that the computer will spend much of its time here, and that the improvements to the inner loop will do the most to increase the efficiency of the program as a whole.

There is obviously not much to gain by shortening a section of program if it is already short. The HCF example traced in full, where the two numbers were 15 and 6, took only three passes through the loop before the result appeared. This hardly seems worth worrying about. Perhaps this example is deceptive; what happens if the two numbers are very different? A few trials show that the smaller is subtracted from the larger many times over. For instance, in finding the HCF of 1001 and 2, the loop would be repeated 501 times. At the limit, if you set one of the numbers to zero the computer will loop round indefinitely doing nothing except using up your computing allowance. Eventually the operating system will stop your job with a dynamic fault, and print a message like "TIME ALLOWANCE EXCEEDED", but this will not happen until the machine has done several million completely fruitless cycles. Of course the numbers supplied are not supposed to be zero; the user is told that the program only works correctly on positive numbers. This does not alter the fact that in practice, perhaps through a punching error, a zero might still creep in.

The repeated subtraction of one number from another, until the residue is smaller than the number being subtracted, is equivalent to finding the remainder after division. Thus subtracting 2 from 1001 500 times leaves a remainder of 1, and the same result can be obtained after only one operation by specifying 'the remainder when 1001 is divided by 2'. In Algol 68 there is an operator with exactly this function—**mod**, which is an infix operator with a priority of 7. Thus the value of 1001 **mod** 2 turns out to be 1. *x* **mod** 0 (where *x* is any number) is undefined and produces a dynamic fault.

Substituting this operator into the above algorithm gives

```
. . . . . . . . . . .

while a ≠ b do if a>b then a: = a mod b else b: = b mod a fi
          od;
print (a)

. . . . . . . . . . .
```

This looks promising; but it is advisable, before submitting the new program to the computer, to trace one or two simple cases, just as a check. The trace for the pair of numbers 35 and 14 is as follows.

a	b
~~35~~	~~14~~
7	0

On the first time round, a > b and so the machine would do *a*: = *a* **mod** *b*. The new value in *a* is 7.

On the next time round, a < b, and the machine would do *b*: = *b* **mod** *a*. This gives *b* and value of 0, because 7 divides 14 exactly.

On the third time round, b is now again less than a, and so the machine would try to carry out *a*:= *a* **mod** *b*. As *b* is zero, this would give a dynamic fault! Furthermore, a few more checks with different pairs of numbers shows that this would happen every time.

Before giving up in despair and returning to the reliable but slow algorithm, it is as well to investigate this failure a little more deeply. The value zero can only arise if the larger number is an exact multiple of the smaller.

To prevent the dynamic fault, the loop can be rearranged so that it is never executed with either of the two variables set to zero. The section of program becomes

```
while a * b ≠ 0 do if a > b then a:= a mod b
                             else b:= b mod a

                   fi
                od;
if a = 0 then print (b) else print (a) fi
```

Some tracing experiments show that this version is working correctly. It seems to find the HCFs of most pairs of numbers (even those that are very different) in just a few cycles.

EXERCISES

***5.1** (2) Trace the following program, with its data. Say what you think it does, in general terms.

```
begin int x, y;
   read (x); while read (y); y ≠ 9999 do
   if y > x then x:= y fi
                                      od;
   print (x)
end
```

45 77 32 123 44 84 39 2 9999 (data)

***5.2** (2) Sketch the railway that corresponds to the program in exercise 5.1

5.3 (3) Take the HCF program on p. 58 and insert any changes you may think are needed to make it more robust. (Hint: Both data numbers are supposed to be *positive*.)

***5.4** (2) Trace the following program with its data.

```
begin int a, b, s, t;
  read (t); a: = 1;
    while a < = t do s: = 0; b: = 1;
                     while b < = a do s+: = b; b+: = 1 od;
                     print(a); print (s); print (newline);
                     a+: = 1
                 od
end
```

5 (data)

***5.5** (3) Using the program in exercise 5.4, show how the number of times the instructions $s+:=b$ is executed depends on the number read as the datum. (Hint: Trace the program for data values of 1, 2, 3, 4, . . . and hence derive a general expression.) How many times is the instruction executed when the datum is 100? How many instructions are executed in all when the datum is 100?

5.6 (3) Modify the program in exercise 5.4 so as to produce the same results more quickly.

***5.7** (4) (P) If x is the highest common factor (HCF) of two numbers a and b, their lowest common multiple (LCM) is ab/x.

(a) Using a section of code given on p. 68, write a program that reads two numbers p and q and prints out their LCM.

(b) Adapt your program to calculate the LCMs of a given data number q and all 10 numbers between 100 and 109. The results should be printed in the following form.

```
LCM OF 75 AND
100  101  102  103  104  105  106  107  108  109
300 7575 2550 7725 7800  525 7950 8025 2700 8175
```

(c) Adapt your program again to make it tabulate the LCM of p and q for all combinations of p from 1000 to 1099 and for q from 100 to 109 (1000 values in all).

6 *Program Structure*

'Beauty is truth, and truth, beauty'

John Keats, 'Ode on a Grecian Urn'

Algol 68 is designed on the principle of *orthogonality*. This implies that the language is governed by a small number of general rules, which can be applied uniformly, without exceptions and without interference with one another.

One of these general rules is that the value yielded by an expression of a certain type can always be stored in a variable of the same type. Thus an integer-expression can be assigned to an integer variable, and a character-expression can be assigned to a character variable. In the topics discussed so far, however, there is an obvious gap arising from boolean-expressions — to what, if anything, can their values be assigned?

In order to store boolean-values, variables of a new mode are necessary. Boolean-variables are declared in a similar way to **int** and **char** variables, but using the system word **bool** to start the declaration. As you would expect, a boolean-variable can have any identifer, provided that it does not clash with that of another variable.

The value of any boolean-expression is always either **true** or **false**. It follows that a boolean-variable is also limited to holding these two values. The symbols **true** and **false** are actually Algol-68 system words that specify all (that is, both) possible boolean literals.

Boolean-variables are often useful in remembering truth values that may be needed some time after they are calculated. The program in the following example is supposed to read a set of positive numbers terminated by 0, and simply to state whether any of the numbers is greater than 50. A boolean-variable called *yes* is used to remember the current situation as the program runs. Initially *yes* is set to **false**; then any number greater than 50 sets it to **true**, so that at the end of the data it only remains **false** if no such number was found.

```
begin int x; bool yes; yes: = false;
   while read (x); x > 0 do if x > 50 then yes: = true fi od;
   if yes then print ("AT LEAST ONE NUMBER GREATER THAN 50")
       else print ("NO NUMBERS GREATER THAN 50")
   fi
end
```

Note the use of a boolean-*variable* instead of an expression after **if**. This is quite legitimate; the value in the variable is used to make the choice.

Algol 68 includes a set of special operators for handling boolean-values. They are often called *logical* operators because they have been borrowed from the study of symbolic logic. There are three of them

not is a monadic operator, with the usual priority of 10. It takes one boolean-operand, and delivers a result according to the following simple rule

not true = **false**
not false = **true**

and is a dyadic operator of priority 3. It takes two boolean-operands and gives the value **true** only if both operands are **true**.

or is another dyadic operator, of priority 2. It also takes two boolean-operands, but returns the value **true** if either or both of the operands are **true**.

These rules are summarised in truth tables 6.1 and 6.2, where A and B stand for boolean-values.

A	**not** *A*
true	**false**
false	**true**

Table 6.1

A	*B*	*A* **and** *B*	*A* **or** *B*
true	**true**	**true**	**true**
true	**false**	**false**	**true**
false	**true**	**false**	**true**
false	**false**	**false**	**false**

Table 6.2

When writing boolean-expressions, you should remember that **and** has a higher priority than **or**, and **not** has the highest priority of all. If you want to change this order, or if you are not sure of it, use brackets.

Suppose that the boolean-variables *a*, *b* and *c* have the values **true**, **true** and **false**, respectively, and consider the expression

not (*a* **or** *b* **and** *c*) **and** (*a* **and** *b* **or not** *c*)

Substituting the values of the variables into this expression gives

not (true or true and false) and (true and true or not false)

It is possible to simplify the parts inside the brackets—first the **nots**

not (true or true and false) and (true and true or true)

then the **ands**

not (true or false) and (true or true)

and finally the **ors**. The brackets can now be removed because there are no operators left inside them

not true and true

The removal of the brackets allows the expression as a whole to be simplified. Applying the **not**: gives

false and true

while applying the **and** produces the result

false

This is the value of the expression.

Boolean-operators are often useful in sorting out complex categories. Imagine a program that handles records of people, and produces statistics about them. Each person is represented by the values in four variables, as follows

age: the **int** variable *age*
sex: the **char** variable *sex*, which takes one of the values "*M*", "*F*"
marital status: the **char** variable *ms*, which takes one of the four values "*S*" (single), "*M*" (married), "*W*" (widowed) and "*D*" (divorced)
education: the **char** variable *education*, which takes one of the three values "*U*" (university degree or equivalent), "*O*" (O level) or "*N*" (no qualification).

Initially, the particulars of each person could be punched on to a card as a number and three characters. Part of the data would appear as

```
39MMU
34FMU
73FWU
13MSN
11FSN
9MSN
3FSN
```

The values for each person could be read in by a sequence of *read* instructions like

read (*age*); *read* (*sex*); *read* (*ms*); *read* (*education*)

Suppose that it is required to know

(a) the total number of people in the set of records
(b) the total number of graduates under 25
(c) the total number of widowed or divorced women over 40.

It is possible to define boolean-variables that indicate whether a person is under 25, or widowed, or has any other attributes that can be deduced from the data. These can then be combined into expressions that describe the categories of interest.

The following is the complete program. It is assumed that the data are terminated by a dummy record with an 'age' of 999, and that the accuracy of the records has been checked by another program, commonly called a *data-validation run.*

```
begin int age, total, yg, owd;
   char sex, ms, education;
   bool under25, male, graduate, over40, widowed, divorced;
   total: = 0; yg: = 0; owd: = 0;
   while read (age); age ≠ 999 do read (sex); read (ms); read (education);
      under25: = age < 25; over 40: = age > 40;
      male: = sex = "M";
      graduate: = education = "U";
      widowed: = ms = "W"; divorced: = ms = "D";
      total+: = 1;
      if graduate and under25 then yg+: = 1 fi;
      if not male and (widowed or divorced)
         and over40 then owd+: = 1
      fi
                                    od;
   print (newline); print ("TOTAL NUMBER OF PEOPLE IS");
      print (total);
   print (newline); print ("NUMBER OF GRADUATES UNDER 25 IS");
      print (yg);
   print (newline);
   print ("NUMBER OF DIVORCED OR WIDOWED WOMEN OVER
      40 IS);
   print (owd)
end
```

The boolean-operators are also useful with relations. Once a relation like < has been worked out it must be either true or false, and so it takes a boolean-value. For example, it is possible to write instructions such as the following

```
int x, y;
. . . . . .
x < = 5 or y > 3 (true if either x < = 5 or y > 3)
x > = 10 and x < = 15 (true if x lies between 10 and 15)
```

Note that it is not possible to write something like *10* < = *x* < = *15* to indicate a range of values of *x*, because the machine would have to evaluate the first relation *10* < = *x*, and then it would be left with a boolean-value to compare with a number, which is illegal.

To bring order into the whole matter of expressions, it only remains to point out that the relations are actually *dyadic operators*; <, >, < = and > = have priority 5, and = and ≠ have priority 4. There is, however, one important difference between these operators and the others so far encountered. The pure arithmetical operators like + and * each take integers as their operands and yield an integer result. Likewise the boolean-operators **not**, **and** and **or** each take boolean-operands and deliver a boolean result. The relations, on the other hand, each take integer operands and yield a boolean result!

In order of priority, the relations come below the arithmetical operators but above the boolean-operators. The higher the priority, the more strongly will an operator 'bind' the items on either side. This means that the operands of the relations need not be only variables or literals but can also be arithmetical expressions of arbitrary complexity. In every case (except where brackets intervene) they are evaluated *before* the relational operators are applied.

In Algol 68, there are various ways of transferring values from one mode to another. Some of them are shown in figure 6.1

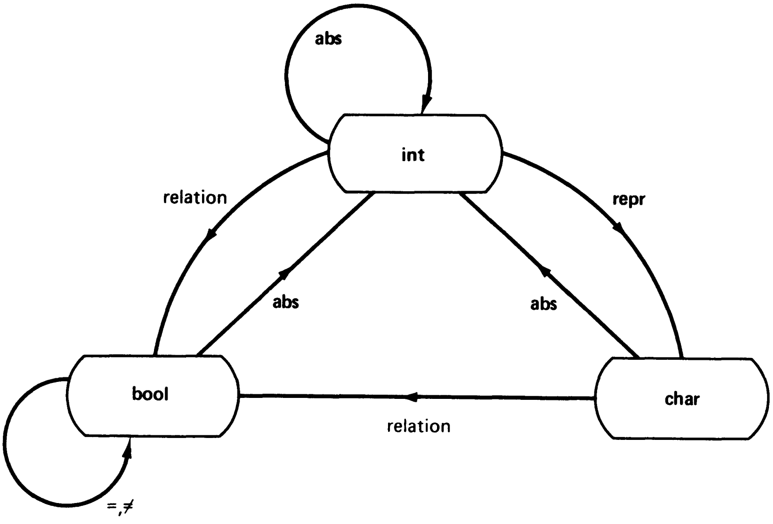


Figure 6.1

First, as already noted, integers and character expressions can be used to derive boolean-values by means of the relational operators. This is shown on figure 6.1 by the two lines marked 'relation'.

Second, the operator **abs** can be applied to objects of any of the three modes. This operator always produces something of mode **int**, but it works in different ways depending on the mode of the operand.

(1) If **abs** is applied to an integer, it simply produces the absolute value, or modulus. There is no change of mode.

(2) If **abs** is applied to a boolean, it translates **true** into 1 and **false** into 0.

(3) If **abs** is applied to a character, it generates an integer according to an internal code. This code will vary depending on the machine that is used, but it will always have three basic properties
   (a) there will be a unique numerical code for each member of the character set (that is, different characters always give different codes)
   (b) the letters "A" to "Z" give codes consisting of 26 different integers in the correct alphabetical order. On certain machines (particularly the ICL 1900) these integers will be consecutive but on others (such as the IBM 370 range) they may not be.
   (c) similarly, the digits "0" to "9" give 10 integers in the right order; they will be consecutive on most implementations.

Finally the operator **repr** transforms an integer that is the internal code of some character into that character; the effect of **repr** is exactly the opposite to that obtained when **abs** is applied to a character. If the operand of **repr** is not a legitimate character code (this is possible because the size of the character set is far smaller than the number of different integers) a dynamic fault is produced.

The operator **abs** when applied to boolean-values is often useful in conditional arithmetical expressions. Consider the case of a man's income-tax allowance, which might be something like

10 000 doubloons basic
+ 8000 doubloons if he is married
+ 4000 doubloons for each child he supports.

Suppose that it is required to write a program to calculate the tax allowance, and that a boolean-variable called *married* and an integer variable *cn* that gives the number of children have already been set up. There are various possibilities, such as

```
taxallowance: = 10000;
if married then taxallowance +: = 8000 fi
if nc ≠ 0 then taxallowance +: = (nc * 4000) fi
```

but perhaps the neatest is

```
taxallowance: = 10000 + abs married * 8000 + nc * 4000
```

It is convenient to use **abs** as applied to characters and **repr** in various alphabetical and lexical manipulations. For the most part the relational operators can also be applied to modes other than integers. Thus = and ≠ can be used with characters, with obvious meaning. They can also be used with booleans: = gives **true** if the two operands are the same, and ≠ gives **true** if they are different.

The operators <, >, < = and > = can also be used with characters. The expression j < k (where j and k are character variables or literals) has the meaning of **abs** j < **abs** k. In other words (because of the way **abs** is defined) the two operands are tested for being in alphabetical or numerical order. For example

“*P*” < “*Q*”

is true, but

“*8*” < “*5*”

is false.

The other relations work in a similar way. For example, a character x is certainly a digit if the following expression is true

x > = “*0*” **and** *x* < = “*9*”

As a further illustration, consider the following program, which reads a series of numbers ended by a / and converts it into code according to the simple rule 0→9, 1→8, 2→7, and so on up to 9→0. Spaces and other non-digit characters are left unchanged. The coded message ends with a / so that the same program can later be used to translate it back again.

```
begin char next, code;
   while read (next);
      next ≠ "/" do if next > = "0" and next < = "9"
                       then code: = repr (abs "9" + abs "0" - abs next);
                       print (code)
                       else print (next)
                    fi
                 od;
   print ("/")
end
```

There are some useful minor extensions of the constructions already considered.

Testing for a number of possibilities, one after the other, often produces several nested **if** statements. Consider the job of translating exam marks into grades, according to the following table.

Mark	More than 75	75–61	60–49	48–36	35 or less
Grade	A	B	C	D	E

A program for this could well include a section like

```
if exam > 75 then grade: = "A"
                else if exam > 60 then grade: = "B"
                                  else if exam > 48
                                          then grade: = "C"
                                          else if exam > 35
                                                  then grade: = "D"
                                                  else grade: = "E"
                                               fi
                                  fi
                fi
fi
```

This cumbrous array can be shortened (a little) by using the new system word **elif**. This replaces **else if**, but does not need a closing **fi**. Hence.

```
if exam > 75 then grade: = "A"
  elif exam > 60 then grade: = "B"
  elif exam > 48 then grade: = "C"
  elif exam > 35 then grade: = "D"
                 else grade: = "E"
fi
```

This version of the construction is easier to lay out and there is less likelihood that its **if**–**fi** brackets will be wrongly matched.

Another contraction is the rule by which the system words **if**, **then**, **else**, **fi** and **elif** can be replaced by (, | , | ,) and |:. **then** and **else** use the same symbol, but this never causes ambiguity. The shortened symbols can be confusing, but they are useful in conditional expressions that are long and uniform. For instance

```
y:= 59*if x < 4 then x+2 else 2-x fi
```

may be replaced by

```
y:= 59*(x<4|x+2|2-x)
```

and 'factoring out' the assignation in the instruction that assigns the grade leads to the form

```
grade:= (exam > 75|"A"|:exam > 60|"B"|:exam > 48|"C"|:exam
   > 35|"D"|"E")
```

So far, the *read* and *print* statements in this book have always used one operand each. Both these instructions can accept lists of operands, provided they are separated by commas and enclosed in an extra pair of brackets. Thus

read ((*sex*, *ms*, *education*))

is equivalent to

read (*sex*); *read* (*ms*); *read* (*education*)

and

print ((*newline*, "*POSTAGE WILL BE*", *p*, "*PENCE*"))

is a shorter form of

print (*newline*); *print* ("*POSTAGE WILL BE*"); *print* (*p*); *print* ("*PENCE*")

In either case the items on the list may be of different types.

Essentially, the print statement is concerned with handling values. It is not therefore limited to handling variables and constants, but can deal with anything that generates a value suitable for printing. Expressions of any complexity may be used, including conditional expressions. For example, an instruction such as *print* ((*newline*, "*YOUR RESULT IS*", (*marks* > = *50*| "*PASS*"| "*FAIL*"))) would be perfectly acceptable.

The read statement, on the other hand, is concerned only with reading values from the input medium and storing them in named variables. It does not make sense to 'read' the value of an expression, and the only proper operands for the read command are variables.

Programs are often compared to onions. On the outside, there is a shell. Removing this shell reveals another shell inside, and so it continues to the heart of the onion itself. Building a complex program is like designing onion shells. Sometimes it is best to start from the inside and to work outwards but on occasion it may be more convenient to go the other way.

A good example of the outward approach is the second version of the HCF program in chapter 5, which was constructed by writing a 'kernel' that found the HCF of just two numbers, and then adding an outer shell that allowed the program to handle a whole stream of data pairs.

Consider now a different problem: writing a program to find a perfect number. A perfect number is defined as one that is equal to the sum of its factors; it is assumed that 1 is always a factor but that the number itself is not. It turns out that, for example, 6 is perfect because the factors of 6 are 1, 2 and 3, which add up to 6; but 18 is not because the factors of 18 are 1, 2, 3, 6 and 9, which add up to 21.

Since it is already known that 6 is perfect, consider finding the next perfect number after 6. This time, the procedure adopted is to work from the outside inwards, and therefore the first step is to design the outermost shell of the program. An obvious choice is

```
begin int n;

        set n to the next
        perfect number after 6

        print (("THE NEXT PERFECT NUMBER IS", n))
end
```

This does not seem to have advanced matters very far: the next perfect number has still to be found. However, the printing requirement has been separated off, and in that sense the problem has been made a little easier — the top skin of the onion has been removed. The outermost shell is always very thin!

The second step is to decide how to find the next perfect number. One way is to generate all the numbers from 7 upwards and to test them, until one is found to be perfect. This gives

```
begin int n;

    n:= 7;
    while  n is not perfect   do n+: = 1 od;

    print (("THE NEXT PERFECT NUMBER IS", n));
end
```

Another layer removed! The inside still remaining is a serial-boolean clause that gives the value **true** if *n* is not perfect.

It is now necessary to establish a rule for determining whether a given number is perfect. In view of the definition of a perfect number, one method would be to add up all the factors of the given number and to compare the sum with the number itself. A new variable *s* is introduced for the sum

```
begin int n, s;

  n: = 7;

  while  set s to the sum of the factors of n ; s ≠ n  do n+: = 1 od;

  print (("THE NEXT PERFECT NUMBER IS", n))
end
```

The onion is now visibly decreasing in size, and the problem it contains is not so many steps away from solution.

The easiest way of finding which numbers divide a number n is to try all the possible factors from 1 upwards. The largest factor cannot be greater than $\frac{1}{2}$n. Another variable, say *x*, is needed to represent each trial factor

```
begin int n, s, x;
        n: = 7;
        while
            s: = 0; x: = 1;
               while x < = n÷2
                  do
                     if  x divides n exactly
                        then s+: = x
                     fi; x+: = 1
                  od;
                  s≠n
            do n+: = 1 od;
        print (("THE NEXT PERFECT NUMBER IS", n))
end
```

The last shell is easy to expand. If *x* divides *n* exactly then the remainder after division must be zero. The innermost shell may be replaced by the boolean-expression *n* **mod** *x* = *0*. The final version of the program will be

```
begin int n, s, z;
   n: = 7;
   while s: = 0; x: = 1;
   while x < = n÷2
      do if n mod x = 0 then s+: = x fi
         x+: = 1
      od;
      s ≠ n
      do n+ := 1 od;
   print (("THE NEXT PERFECT NUMBER IS", n))
end
```

This method of starting with the outside of the onion is sometimes called *stepwise refinement.*

EXERCISES

***6.1** (1) What is wrong with the following version of the program on p. 70?

```
begin int x; bool yes; yes: = false;
  while read (x); x > 0 do yes: = x > 50 od;
    if yes then print ("AT LEAST ONE NUMBER GREATER
       THAN 50")
          else print ("NO NUMBERS GREATER THAN 50")
    fi
end
```

***6.2** (0) Students often write something like

```
bool j;
. . . . .
if x = 17 then j: = true else j: = false fi
```

Suggest a shorter form.

***6.3** (2) Fill in the last column of this truth table

a	b	c	not (a **or** b **and** c) **and** (a **and** b **or not** c)
true	**true**	**true**	
true	**true**	**false**	**false**
true	**false**	**true**	
true	**false**	**false**	
false	**true**	**true**	
false	**true**	**false**	
false	**false**	**true**	
false	**false**	**false**	

***6.4** (3) Deduce a simpler boolean-expression with the same value as the one in exercise 6.3.

6.5 (2) Write a version of the program on p. 73 that tells you

(a) the number of married people between 20 and 25

(b) the number of divorced men who do not have university degrees

(c) the number of people over 65 who have no educational qualifications.

6.6 (4) (P) In most implementations of Algol 68, the internal representations of the decimal digits are consecutive integers

abs "1" = **abs** "0" + 1
abs "2" = **abs** "1" + 1

etc.

Use this fact to write a program which reads a series of numbers in the *octal* notation (scale of 8) and prints the corresponding decimal value of each number. The octal value should be printed first, for reference, and the output should be suitably headed.

You may assume that the octal numbers are separated by spaces, and that the last one is zero.

A sample output might be

```
OCTAL             DECIMAL

    7                +7
   10                +8
  144                +100
 1812          NOT OCTAL NUMBER
 1000                +512
    0                +0
```

7 Declarations and Reach

'"The name of the song is called 'Haddocks' Eyes'."
"Oh, that's the name of the song, is it?" Alice said, trying to feel interested.
"No, you don't understand," the Knight said, looking a little vexed. "That's what the name is *called*. The name really is 'The Aged Aged Man'."
"Then I ought to have said 'That's what the *song* is called'?" Alice corrected herself.
"No, you oughtn't: that's quite another thing! The *song* is called 'Ways And Means': but that's only what it's *called*, you know!"
"Well, what *is* the song, then?" said Alice, who was by this time completely bewildered.
"I was coming to that," the Knight said. "The song really *is* 'A-sitting on a Gate': and the tune's my own invention."'

Lewis Carroll, *Alice through the Looking Glass*

This chapter is about declarations. As you have already seen, declarations are used in every Algol-68 program to set up space for working variables and to attach names to the various storage cells that may be needed.

In most circumstances the designers of Algol 68 compel the programmer to say exactly what he means. In the case of declarations, however, they seem to make an exception. Most declarations can be written in two different ways: an *extended* form, which actually means what it says; and a *shortened* form, which is convenient for normal use but quite obscure if you want to discuss its precise meaning.

All the declarations used so far have been in the shortened form. Since the aim at this point is to explain what declarations really mean, it is best to set aside the shortened form and to start afresh with the extended notation. When the subject has been thoroughly discussed, the shortened form will be reintroduced and used for the remainder of the book.

Fundamentally a declaration serves to link an arbitrary identifier with a value of a certain mode. Once made, a declaration lasts for a finite time (for instance, until the end of the program) and, throughout its life, the value linked with the identifier remains *unchanged*.

The basic way of writing a declaration is

    mode identifier = value

Here 'mode' is the mode of the identifier; the identifier itself is freely chosen, and the 'value' must be compatible with the mode specified. The = sign is quite different from the relational operator =. Here it does not test for equality; it causes association.

A simple example of a declaration (which is written, remember, in extended, not shortened form) is

int *dozen = 12*

This declares the identifier *dozen*, and associates it with the mode **int** and the constant value 12. The identifier accesses this value throughout its existence, so that it is possible to write[1] '*dozen*' wherever the integer 12 is meant. The word dozen now *stands for* an integer of fixed value; it is an identified constant.

Identifiers of mode **bool** and **char** may be declared in a similar way. It is often convenient to put

bool *f* = **false**; **bool** *t* = **true**

because the single letters *f* and *t* may then be written instead of the full symbols **false** and **true**.

Since identifiers of mode **int**, **bool** and **char** all stand for constants, it is not sensible to try to assign any new values to them; it is as absurd to say, for example

dozen: = *19*

as it is to put *12*: = *19*.

By now, you may feel that all the supposedly solid ground you were standing on is somehow dissolving beneath you. Remember that neither of the words *variable* and *store* has yet been mentioned. Take courage and read on.

The conventional idea of a variable is that of a named cell in the workspace, which can hold a value of a certain type. As the calculation proceeds this value is used from time to time and may be changed by a read or an assignation.

The standard declaration of such a variable in Algol 68 would be written

$$\underbrace{\textbf{ref}\ \text{mode}}_{\text{mode}}\ \text{identifier} = \underbrace{\textbf{loc}\ \text{mode}}_{\text{value}}$$

for instance

ref int *kate* = **loc int**

where the mode of the identifier *kate* is **ref int** and the value is **loc int**.

The rules demand that the value associated with the identifier must be a constant: it may not change during the life of the declaration. This seems to lead to a contradiction—*kate* is intended to be a variable! There is, however, one thing about a variable that is invariant, and that is the position or address of the corresponding cell in the store. No matter how often its contents are changed, the place occupied by a cell in the working store always remains the same. The value ascribed to the identifier in a declaration of this type does not

[1] Note that it is not permissible to write '*dozendozen*' to mean 'one thousand two hundred and twelve' because *dozen* is an **int**, and not a sequence of two digits.

represent the (changing) *value* of the variable in some cell; it represents the (constant) *position* of that cell. This value has a new mode (**ref int**) but, since it has already been accepted that values can represent numbers, characters and truth values, the idea of a value representing a position should not prove too difficult to grasp.

As a rule, modes that start with the qualifier **ref** all represent positions. Thus a **ref int** is the position of a cell that can hold an **int** value.

The action of the declaration can now be described precisely. When a program starts, the computer has a supply of workspace divided into numbered cells, much like the long thin blackboard described in chapter 4. This workspace is usually called the *stack.*

As declarations are executed, the cells in the workspace are allocated to variables, working from one end of the stack. (In the diagrams in this book, cells will be allocated from left to right.) At any time, therefore, all the cells up to a certain point will have been allocated, and those beyond that point will still be free. A special secret memory cell, called the *stack pointer*, will remember where the free cells start.

When the machine obeys a declaration like the one above the following events happen.

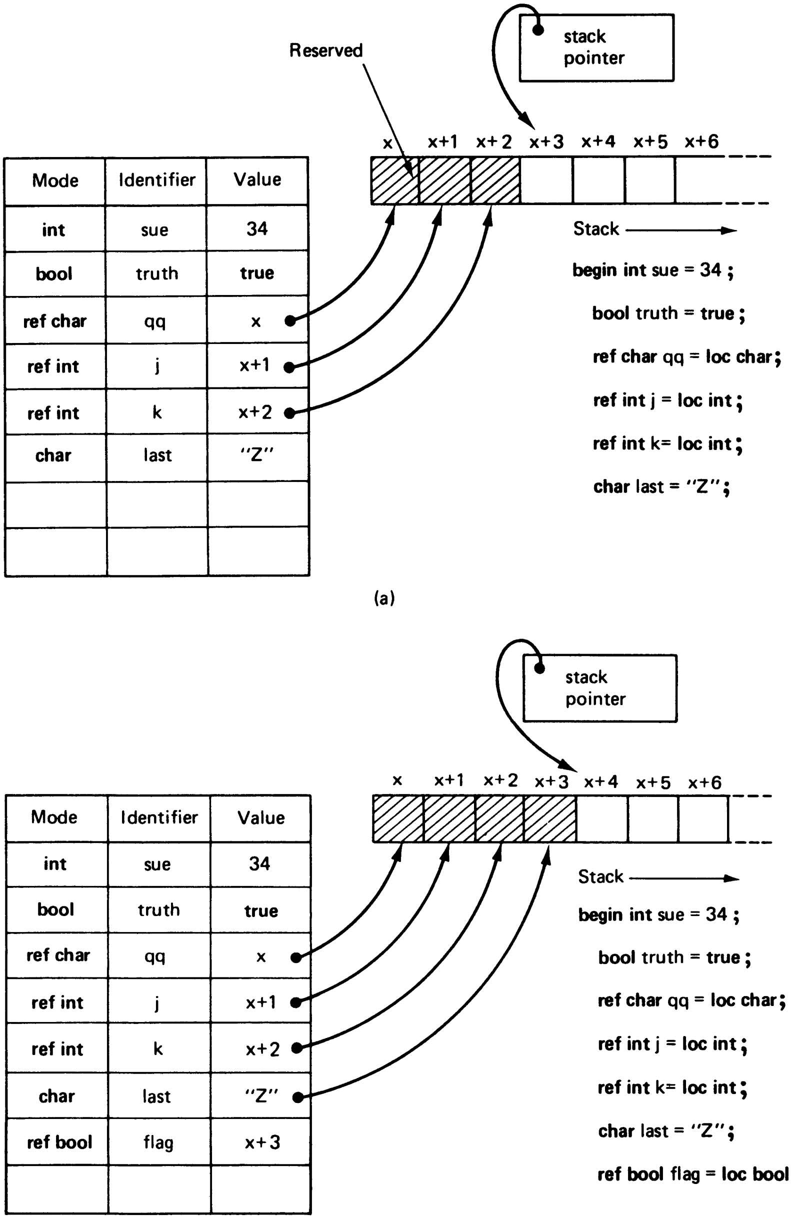
(1) The machine makes a note of the identifier in a special table together with the mode that precedes it; in the given example, *kate* would be associated with **ref int**.

(2) The machine works out the expression **loc int**. **loc** is a special monadic-operator that takes the name of a mode as its operand. It 'reserves' a storage cell suitable for an object of that mode, and delivers the position of the newly reserved cell. As a side effect of reserving the cell, it moves the stack pointer up by the width of one cell so that the same cell cannot be allocated to another variable as well. The value of the expression **loc int** is therefore the *position* of a new integer cell.

(3) The value so produced is ascribed to the identifier, which continues to access the value for the whole of its existence.

Figure 7.1 shows the effect of a series of declarations. The positions of the cells in the working store are deliberately indicated as x, x + 1, x + 2 and so on to show that the programmer does not know, and does not need to know, exactly where they are. It is sufficient for the computer to know and to record the appropriate values in the table of identifiers. Thereafter the identifiers are said to 'point to' the cells concerned.

Unfortunately this does not end the difficulties. The objects that you may have thought of as variables of mode **int** turn out to be constants of mode **ref int**. What effect does this have on expressions and assignations that involve objects of this type?



Mode	Identifier	Value
int	sue	34
bool	truth	**true**
ref char	qq	x
ref int	j	x+1
ref int	k	x+2
char	last	"Z"

```
begin int sue = 34 ;
   bool truth = true ;
   ref char qq = loc char;
   ref int j = loc int;
   ref int k= loc int;
   char last = "Z";
```

(a)

Mode	Identifier	Value
int	sue	34
bool	truth	**true**
ref char	qq	x
ref int	j	x+1
ref int	k	x+2
char	last	"Z"
ref bool	flag	x+3

```
begin int sue = 34 ;
   bool truth = true ;
   ref char qq = loc char;
   ref int j = loc int;
   ref int k= loc int;
   char last = "Z";
   ref bool flag = loc bool
```

(b)

Figure 7.1(a) Table of names and their associated modes and values; (b) effect of adding a new declaration

First consider an assignation

kate: = *0*

The value on the right is an integer literal, which can be stored in an integer cell, while on the left there is an identifier of mode **ref int**, whose associated value is the position of an integer cell. Clearly there is no real problem; the computer can easily store the integer values in the integer cell to which the identifier points.

Next, consider

```
kate: = 17 ÷ kate
```

The ÷ operator needs two integers as its operands, but in this case one of them is a **ref int**. It makes no sense to divide a number by a position and, even if it did, this is not what the computer is being asked to do. When a situation of this kind arises, the machine does not use the (constant) value of the identifier directly, but looks up and uses the contents of the cell to which the value points. This process is called *dereferencing* and is one example of *coercion*, or change in mode directed by circumstances.

The law of orthogonality ensures that all the remarks made regarding the declaration of integers apply equally to the declaration of characters, booleans and other modes still to be discussed.

In the declarations studied so far, an identifier can access the position of a cell, but the contents of the cell itself are left undefined. To ensure that the cell does have an initial value, the declaration can be extended by adding an assignment operator and an expression of a suitable mode. Thus it is possible to write

```
ref char x = loc char: = "Q"
```

The machine will put the character value "*Q*" into the cell whose position is ascribed to the identifier *x*, as if it had been presented with

```
ref char x: = loc char; x: = "Q"
```

At this point the shortened form of the declaration can safely be reintroduced. It turns out that

```
int j
```

is just a shorthand way of writing

```
ref int j = loc int
```

so that in this case *j* is actually an identifier of mode **ref int**.

Identifiers declared in the shortened form can also be initialised. For instance

```
int n: = 7
```

is equivalent to

```
int n; n: = 7
```

and therefore to

```
ref int n = loc int; n: = 7
```

It is important to distinguish three forms of declaration, which look similar but are somewhat different

(a) **int** *q* (a variable-declaration) is short for **ref int** *q* = **loc int** and declares an identifier of mode **ref int**, but does not initialise the corresponding cell in the workspace.

(b) **int** r = 19 (an identity-declaration) declares an identifier of mode **int**(not **ref int**) and ascribes to it a constant integer value. This is *not* a contraction!

(c) **int** s: = 19 (also a variable-declaration) is short for **ref int** s = **loc int**: = *19*. It declares an identifier of mode **ref int**, and initialises the corresponding cell in the workspace.

Contracted declarations, where several identifiers are declared in one group, also have their full and shortened forms. For example

```
bool red, blue, green
```

is short for

```
ref bool red = loc bool, blue = loc bool, green = loc bool
```

So long as they have the same mode, the identifiers in a declaration of variables need not all have initial values associated with them. Hence it is possible to write sequences like

```
char high: = "W", middle, low: = "H"
```

which defines three identifiers of mode **ref char**, and assigns initial values to two of them. On the other hand it is not legal to mix different modes, even by implication. A 'declaration' like

```
int j, k = 23
```

would be rejected because *j* is of mode **ref int**, but *k* is of mode **int**.

Remember that a serial-clause consists of a sequence of one or more items — units and declarations, including at least one unit. The items may follow one another in any order provided that every identifier is declared before it is used.

For technical reasons, some versions of Algol 68 — in particular Algol 68R — impose an additional rule: if a serial-clause has any declarations at all, it must begin with a declaration. Although the rule is not part of the proper definition of Algol 68, it will be used in this book.

The practice followed so far in this book has been to put all declarations immediately after the opening **begin** of the program, but this rule can be varied as required. Declarations can use expressions as well as simple numbers or truth values, provided that the computer can work them out when the declarations are obeyed. For example, a program could start with

```
begin int x, y; read (x); read (y);
      int size = (x+1)*(y+1)
```

Throughout this program, size would stand for a constant integer with a value determined by the first two data items supplied.

As already noted, any serial-clause enclosed in brackets becomes a unit and can be embedded in another serial-clause. The same applies to serial-clauses that are delimited by special pairs of system words like **then** – **else** or **do** – **od**. Most of the programs in this book have used nesting of this type but so far none of these inner clauses has had its own declarations.

Declarations are acceptable in inner clauses, and are often useful there. There is an important practical difference between identifiers declared in nested clauses and those declared at the outermost level. As mentioned earlier, every declaration has a finite life. In fact, an identifier only continues to exist as long as the serial-clause in which the declaration lies is actually being executed. When that clause ends, the identifier is taken away from the table of identifiers held in the machine, and the space occupied in the working store (if any) is freed and made available to other declarations.

Consider an identifier declared in the outermost serial-clause (just after **begin**). The end of this clause coincides with the end of the program so that the name exists for the entire duration of the program, and can be referred to anywhere. If, on the other hand, an identifier is declared in an *inner* serial-clause it will be deleted before the end of the program is reached, and so there will be parts of the program where the identifier is not available and where its use would be an error.

As an illustration, consider the following short program with nested declarations

```
┌──────────────  1  begin int a, b, sum: = 0, ap: = 1;
│                2    read (a); read (b);
│                3    while ap <= a do
│          ┌───  4      int bp: = 1;
│          │     5      while bp <= b do
reach of   reach 6        sum +: = (ap + bp)↑2;
a, b, sum, of bp 7        bp +: = 1
ap         │     8                 od;
│          │     9      ap +: = 1
│          └─── 10               od;
│               11    print ((newline, "A = ", a, "B = ", b, "F IS",
│                       sum))
└────────────── 12  end
```

The program reads in two numbers *a* and *b*, and works out the sum of the squares of all combinations of numbers up to $a + b$. For example, if $a = 3$ and $b = 4$, the machine calculates

$$(1+1)^2+(1+2)^2+(1+3)^2+(1+4)^2$$

$$+(2+1)^2+(2+2)^2+(2+3)^2+(2+4)^2$$

$$+(3+1)^2+(3+2)^2+(3+3)^2+(3+4)^2$$

or 266. (As far as I know, this particular function of two variables is not at all an interesting one, and the only purpose of the program is to demonstrate the ideas of nested declarations.)

In this program, the identifiers declared in the outermost serial-clause are *a*, *b*, *sum* and *ap*. They exist for the entire program and can be referred to anywhere.

One of the inner clauses of the program is delimited by the **do** in line 3 and the **od** in line 10. The identifier *bp*, which is declared in this serial clause, is said to be *local* to that clause. Its reach extends from line 4 to line 10 only; *bp* can be used anywhere within this serial-clause but outside—either before line 4 or after line 10—it does not exist and any reference to it would be illegal and meaningless.

To discover what actually happens during the running of the program, consider a series of 'snapshots'. First of all, figure 7.2a shows the state of affairs just after the declarations in line 1 have been obeyed. Four cells of the workspace have been allocated, and two of them have been given initial values. The contents of the cells that belong to *a* and *b* are as yet undefined.

When the values of a and b have been read, the program moves on to the declaration in line 4. This results in figure 7.2b.

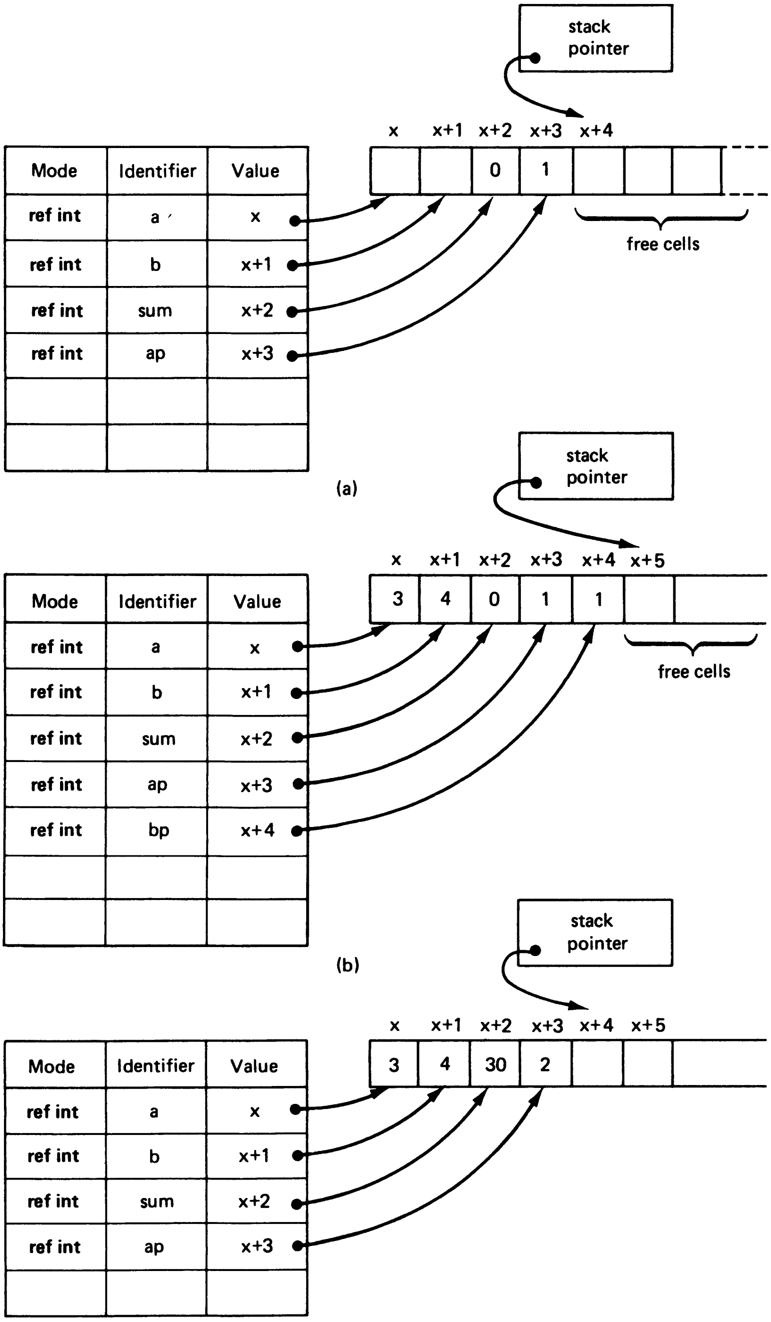
The identifier *bp* has now been entered into the identifier table and a cell (x + 4) has been initialised and reserved.

Now the rest of the loop which runs from line 3 to line 10 is obeyed. As *b* = *4* the statements on lines 6 and 7 will be obeyed 4 times. On reaching the **od** in line 10 *bp*, *ab* and *sum* will hold the values 5, 2 and 30 respectively, as you can easily confirm by tracing.

The **od** in line 10 closes the serial-clause in which *bp* was declared. As the **od** is passed, the program liquidates the identifier *bp*; it removes it from the table of identifiers, and returns the cell associated with it to the set of free cells by moving the stack pointer *back* by one position. The cell that once belonged to *bp* can be used to satisfy the next declaration that needs a cell, no matter what identifier or mode it may have. This gives figure 7.2c.

As the clause with the declaration is part of a loop it is very soon entered again. Each time around, the declaration for *bp* is obeyed; an entry is made in the identifier table and a new cell is allocated. In this example the cell will be the same one (x+4) every time but this is more by chance than design—the system, when it wants more space, will always take the first free cell available.

Several times in this book it has been noted that identifiers can be chosen freely 'so long as they do not clash with other identifiers'. This certainly means that all the identifiers declared in any one serial clause must be distinct, but identifiers declared in different serial-clauses may be the same and still retain their separate identities. There are two cases to consider.



Mode	Identifier	Value
ref int	a	x
ref int	b	x+1
ref int	sum	x+2
ref int	ap	x+3

x	x+1	x+2	x+3	x+4		
		0	1			

(a)

Mode	Identifier	Value
ref int	a	x
ref int	b	x+1
ref int	sum	x+2
ref int	ap	x+3
ref int	bp	x+4

x	x+1	x+2	x+3	x+4	x+5
3	4	0	1	1	

(b)

Mode	Identifier	Value
ref int	a	x
ref int	b	x+1
ref int	sum	x+2
ref int	ap	x+3

x	x+1	x+2	x+3	x+4	x+5
3	4	30	2		

(c)

Figure 7.2

First, imagine two quite disjoint serial clauses as follows

```
                          17  if x < 3 then
        ----------------- 18    int joe, charlie, pat
reach of joe
charlie, pat              ..    .........
        ----------------- 25       else
        ----------------- 26    int sue, jane, anne; bool pat = true
reach of sue,
jane, anne, pat           ..    .........
        ----------------- 33  fi
```

Both of these serial-clauses have declarations for the identifier *pat*. However, if *pat* is mentioned in the first, it must access the **ref int** declared in line 18. Similarly a *pat* in the second serial-clause means the **bool** declared in line 26. Outside the two clauses the identifier does not exist at all, and so there is no ambiguity about the use of the identifier anywhere.

The second situation arises when one clause is embedded in another and both have declarations involving the same identifier, as follows

```
   |---------------------------  1  begin int red, green, purple: = 0;
   |  reach of                   2    read (red); green: = red - 2;
   |  green (outer)              3    while purple < = green do
   |               |-----------  4      char green;
   |               reach of      5      read (green);
   |               green         6      print (green);
reach of           (inner)       7      purple +: = 1
red,     |         |-----------  8                            od;
purple   |                       9    print ((green, "CHARACTERS IN ALL"))
   |-----|---------------------  10 end
```

The rule here is that the inner declaration, where it exists, *masks* the outer one. When green is mentioned in the inner clause between lines 4 and 6, it accesses the **ref char** declared in line 4. In the outer clause, when the **ref char** no longer exists, the identifier green now accesses the **ref int** declared in line 1. Thus the first value for the print statement in line 9 is a *number.*

The diagram shows the reach for the outermost declaration of green. It has a hole in it.

Although this variable is inaccessible while the inner clause is being obeyed, it does not cease to exist. When the inner clause ends the masked variable returns with the same value as it had when the inner clause started.

If you have read this far and followed all the argument, you have understood one of the most difficult aspects of Algol 68. Discuss the reasoning behind the rules with your tutor, and try to answer exercise 7.6.

One facility of major importance, which involves an *automatic* declaration, is an extension of the **while** – **do** – **od** construct. Often the serial clause in the **do** – **od** brackets has to be executed a fixed number of times, or once for each value of a variable that is increased in equal steps. Up to now such loops have always been coded as follows

```
int j: = a;
  while j < = c do serial-clause; j +: = b od
```

where *a*, *b* and *c* are the starting value, increment and final value of the variable *j*. The actual number of times round the loop is *1* + (*c* − *a*) ÷ *b*.

Algol 68 offers a convenient shorthand for this type of construct, as follows

```
for identifier
  from integer-unit
    by integer-unit
      to integer-unit
        while boolean-serial-clause
          do serial-clause od
```

Here **for**, **from**, **by** and **to** are new system words, introduced for the first time. The various parts of the construction are used as follows.

(a) **do** serial clause **od**: This clause is obeyed a number of times determined by the preceding parts of the construction.

(b) **for** identifier: The identifier introduces a *control counter* that, as will be seen, is increased in equal steps for each execution of the serial-clause. The **for** constitutes a declaration, and the identifier of the control counter need not be declared elsewhere. The reach of the control counter extends only to the end of the controlled serial-clause. The identifier can be selected quite arbitrarily; it will mask any other identifier of the same spelling and therefore cannot clash. Although the control counter takes different values, it is of mode **int**, so that no assignments can be made to it. It can only have integer values.

(c) **from** integer unit: This part defines the starting value of the control counter.

(d) **by** integer unit: This component yields the increment — the amount by which the control counter is to be increased each time round the loop. If the value here is negative, then the control counter is *decreased.*

(e) **to** integer unit: This part defines the final value of the control counter.

(f) **while** boolean-serial-clause: This part defines an additional stopping condition. The clause is evaluated every time round the loop before the main serial-clause is obeyed, and the loop ends if the boolean clause is found to be **false**, even if the last value of the control counter has not been reached.

The full form of this construction is unwieldy, and its considerable power is hardly ever needed. It is therefore convenient that any part of it except **do** – **od** is optional and can be omitted.

If any section is left out, a *default option* is assumed if necessary. The defaults are as shown in table 7.1.

Omitted part	*Default*
from integer unit	**from** 1
by integer unit	**by** 1
to integer unit	**to** "infinity"
while serial-boolean-clause	**while true**

Table 7.1

To give some examples

```
for q to 3 do print (q) od
```

is equivalent to

```
for q from 1 by 1 to 3 while true do print (q) od
```

and will print

```
+1      +2      +3
```

If the control counter is not needed in the serial-clause, the **for** and the identifier can be left out. For example, the instruction

```
to 20 do print ("!") od
```

will give

```
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! !
```

This can replace the longer sequence

```
int n; n: = 1;
while n < =20 do print ("!"); n: = n+1 od
```

Omitting **for, from, by** and **to** leaves the **while** – **do** – **od** form used in the first part of this book. It is, as is now apparent, a special case of the more general construction.

Omitting all of the optional parts gives

```
do serial-clause od
```

This implies idefinite repetition and is hardly ever useful.

When the increment (defined by **by**) is less than zero, the loop counts *down* to the final value rather than up. For instance, the statement

```
for j from 10 by -2 to 2 do print (j) od
```

would print

```
+10   +8   +6   +4   +2
```

It is a convenient and easy to nest loops one inside the other. The following is a condensed version of the program first given on p. 89

```
begin int a, b, sum: = 0;
   read (a); read (b);
   for ap to a do
      for bp to b do
      sum +: = (ap + bp)↑2
                od
             od;
   print ((newline, "A = ", a, "B = ", b, "F IS", sum))
end
```

EXERCISES

***7.1** (1) State whether the following declarations are in shortened or extended form. If they are in shortened form, expand them.

(a)	**int** *k*	(b)	**int** *bob* = 87	(c)	**bool** *x*:= **true**
(d)	**char** *q*:= "*X*"	(e)	**char** *x* = "*Q*"	(f)	**bool** *seven*

***7.2** (2) Using line numbers, tabulate the modes and the reaches of all the variables used in the following program.

```
 1  begin
 2     for n to 1000
 3        do int p = n ÷ 100;
 4           int q = (n − 100 ∗ p) ÷ 10;
 5           int r = (n − 100 ∗ p − 10 ∗ q);
 6           if p↑3 + q↑3 + r↑3 = n
 7              then print (newline);
 8              to 20 do print ("!") od;
 9              print ("ANOTHER MAGIC NUMBER IS");
10              print (repr (p + abs "0")); print (repr (q + abs "0"));
11              print (repr (r + abs "0")); print ("   ");
12              to 20 do print ("!") od
13           fi
14        od
15  end
```

7.3 (3) Take the perfect-number program on p. 80 and rewrite it so that the reaches of the variables are as small as possible. Do you see an improvement in the structure?

***7.4** (3) What do you think the program in exercise 7.2 does? Hint: the character literals in lines 10 and 11 represent zeros (not 'oh's). Assume that the digits 0 to 9 have internal representations which are consecutive integers (see exercise 6.6).

7.5 (3) Explain the difference between the following two forms, and discuss the advantages of each.

```
(a)  begin int x, y; read (x); read (y)
        int size = (x+1)*(y+1)
(b)  begin int x, y; read (x); read (y);
        int size: = (x+1)*(y+1)
```

7.6 (7) Write five short notes to justify the following statement. 'The complicated rules about declarations in Algol 68 are intended to solve five different but related problems.

(1) they give complete freedom to choose meaningful identifiers
(2) they give good protection against spelling mistakes
(3) They allow storage space to be used efficiently
(4) they permit the system to check that the modes of objects agree with the operations that are to be done with them
(5) they permit sections of programs to be written independently without the risk of identifier clashes.'

***7.7** (2) What sequence is printed in each of the following cases?

```
(a)  for j from 3 by 2 to 15 while j < 11 do print (j) od
(b)  to 4 do print (1) od
(c)  for j while j < 8 do print (2 *j-3) od
(d)  for j by 7 while j < = 30 do print (j) od
(e)  for j to 3 do for k to 2 do print (j↑2+k↑2)od od
(f)  int j: = 45;
        for j to 5 do print (j) od; print (j)
```

***7.8** (P) The hexadecimal notation for numbers uses the base 16. The 16 digits are written 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A (ten), B (eleven), C (twelve), D (thirteen), E (fourteen) and F (fifteen). The hexadecimal number xyz (where x, y and z are digits) is equal to $x * 16^2 + y * 16 + z$. For example, 1FF = 511 (decimal) and 064 = 100 (decimal).

Write a program to print a table of hexadecimal numbers from 1 to 1024, placing 16 numbers on each line of output.

8 *Simple Arrays and Methods of Search*

'And the earth was without form, and void'

Genesis 1:2

In the beginning there was chaos; and God brought it into order. Since the Creation orderliness has been accounted a major virtue, and no-one can do a complex job, or fulfil responsibilities for other people, unless he practise it with care and diligence.

Order has a great deal to do with classification. If you are an orderly person you will keep your papers in separate files, one for each subject. All the items in any one file are objects of the same category — letters from the tax inspector, or the school reports of your children or entries in your catalogue of gramophone records. Perhaps you even ensure that within each file the objects are sorted into order — alphabetical or by date of arrival. All this takes trouble to set up and maintain, but it gives two advantages that you would not have if you kept all your papers together in an untidy heap. First, you can find any item you may need almost instantly; and second, you can now handle groups of related items as single entities — such as a tax file.

In programming there is a similar situation. Many problems involve groups of similar items, which somehow have to be handled at the same time. If a computer is used to keep records, the program will often have to search a complete list of entries in order to discover whether a specific item is present. Again, programs often use tables of various kinds. Each table consists of a number of items, perhaps of different value but of the same generic type.

In Algol 68, groups of items of the same type are called *arrays*. An array may be pictured as a row of objects arranged in order. The entire group has a single common identifier (a *family name*) and the individual objects themselves can be selected by a numeric label or subscript. Consider a very simple cypher, in which the code lies merely in transmitting the message backwards. If he were limited to the features of Algol 68 already discussed, a programmer would be unable to write a coding program unless he actually knew how many characters the message contained. Even then the program would be painfully long, and would look like a school punishment exercise, as there would be no opportunity to use loops of any kind (see exercise 3.10).

With arrays, the problem is greatly simplified. The program given below will invert a message of up to 80 characters ended with a slash (80 was chosen because this number will fit on to a single punched card).

```
begin int p:= 1; [1:80] char buffer;
  while read(buffer[p]); buffer[p] ≠ "/" do p+: = 1 od;
  while p−: = 1; p > 0 do print(buffer[p]) od
end
```

This program has several unfamiliar points. Consider first of all the declaration

[1 : 80] **char** *buffer*

which is actually short for

ref[ ] **char** *buffer* = **loc**[*1*:*80*] **char**

This declares an *array* variable of 80 characters, all with the collective identifier *buffer* (which is an arbitrary choice). When the declaration is obeyed, a block of 80 consecutive cells in the working space is reserved. The entry in the identifier table associates the identifier *buffer* with the mode **ref** [ ] **char** (pronounced 'reference to row of character') and with the position of the first of the 80 cells.

When the array has been declared, and provided it is still in reach, the program can refer to any of the individual characters in it by using the common array identifier followed by a subscript in square brackets. For instance *buffer*[*1*] is the first character in the array, and *buffer*[*80*] is the last. The subscript need not be a simple number: it may be *any* unit that delivers an **int** value.

Next comes the first **while** of the program. At this stage, *p* holds the value of 1, because it was so initialised. The command *read* (*buffer*[*p*]) is therefore interpreted as *read*(*buffer*[*1*]), and the first character in the data is read and stored in *buffer*[*1*].

The boolean-expression tests whether the character in *buffer*[*1*] is a slash. If not, then *p* is increased by 1 (making it 2) and the cycle is repeated, with the next character from the data going into *buffer*[*2*].

Eventually, a slash is encountered. By then, all the characters of the message are stored, in their correct forward order, in the first few cells of the array. At this stage *p* is equal to the total number of characters read, including the slash. A trace as far as this point is shown in figure 8.1.

The second **while** prints out the characters in reverse order. The variable p is used to 'point to' the next character to be printed, and is *decreased* by 1 every time round the loop. The instructions are arranged so that the first character to be printed is the one immediately before the slash and the last is taken from *buffer*[*1*]. Figure 8.2 shows the complete trace of the whole program.

This simple example illustrates all of the really important properties of arrays. A number of other features will be described in the next chapter, but it is worth remembering that the following two aspects overshadow all others.

(a) the possibility of declaring a row or ordered group of items all of the same mode, and of giving them a common identifier

(b) the ability to refer to any item within a group by using a subscript with a value that may vary as the program is executed.

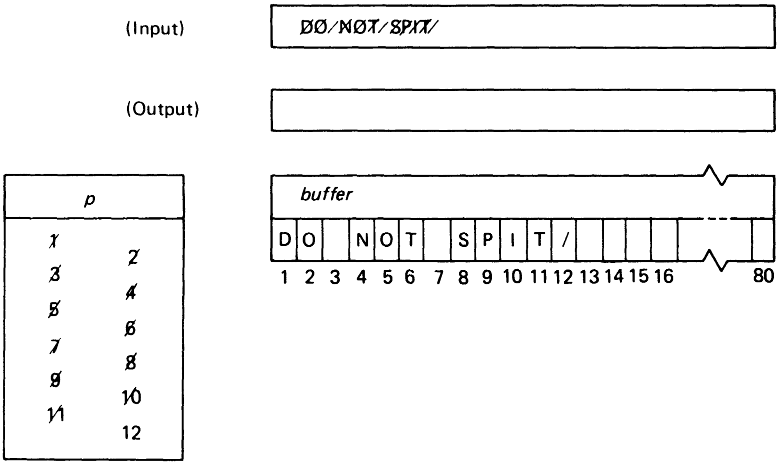


Figure 8.1

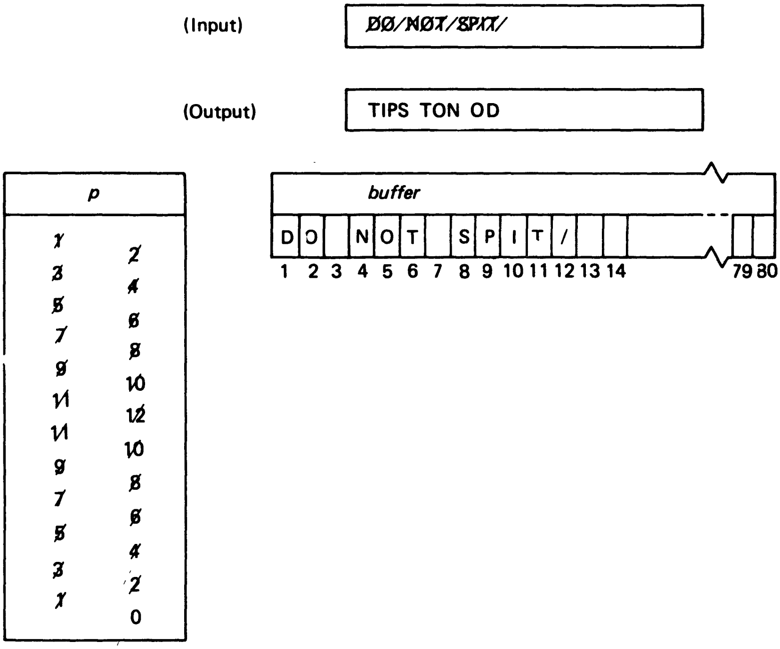


Figure 8.2

The facilities for declaring arrays apply equally to arrays of integers, booleans and objects of any other mode.

The power and flexibility that arrays provide are best illustrated by a number of examples. The first is a problem that could be of interest to the owner of a Chinese restaurant. On the menu the items are usually listed by name and number, as in figure 8.3, and most customers order by number, saying, for example

'Three 34s, two 57s, a 49, a 67, and four 18s'

MENU

1	Egg Foo Yung	75p
2	Beef Chow Mein	85p
3	Beef Chop Suey	80p
4	Pork Chow Mein	70p
5	Pork Chop Suey	67p
6	Sweet and Sour Pork	92p
7	Rice (per portion)	20p
.	....	...
67	Soy Sauce	2p
68	Chow-chow	20p

VAT = 8 %

Figure 8.3

The program that follows is designed to help the *restaurateur* by working out his customers' bills. The data for each run of the program will be in two parts: first the actual price list and second a list of the quantities and menu numbers of the items consumed. Each block of data is terminated with a 0, which cannot occur naturally as a data item. A possible set of data is shown in figure 8.4.

```
75  85  80  70  67  92  20 ┐
.......................... ├ (Price list)
2  20  0                   ┘
3  34   ┐
2  57   │
1  49   ├ (Details of actual dinner eaten)
1  67   │
4  18   ┘
0
```

Figure 8.4

The program itself uses an integer array called *pricelist*. It is arranged that the price of an item of a given number is stored in the cell with the corresponding subscript, so that, for example, the cost of a portion of rice is to be found in *pricelist*[*7*]. It is assumed that there will be not more than 100 items on the menu.

Other variables in the program are

nd	(the actual number of different menu items)
price	(the price of a single item)
qty	(the amount of any single item in the customer's dinner)
menunumber	(the identity of an item)
sum	(the total that has to be paid).

The program is as follows.

```
begin int nd: = 0, price; [1:100] int pricelist;
  while read (price); price > 0 do nd +:= 1; pricelist [nd] := price od;
  print ((newline, "CUSTOMER'S ACCOUNT", newline,
            "DISH NUMBER UNIT PRICE TOTAL", newline
       ));
  int qty, menunumber, sum: = 0;
  while read (qty);
    qty ≠ 0 do read (menunumber);
              print ((newline, menunumber, qty));
              if menunumber < = 0 or menunumber > nd
                then print ("WRONG ITEM NUMBER")
                else price: = pricelist [menunumber];
                     print ((price, qty * price));
                     sum +: = qty * price
              fi
           od;
 print (("TOTAL", sum, newline,
 "      VAT AT 8%", sum * 8 ÷ 100, newline,
 "      TOTAL TO PAY", sum * 108 ÷ 100,
        "P."
      ))
 end
```

A typical set of results from this program is shown in figure 8.5.

This example shows how an index (in this case an item number) may be used to select an entry from a table. Each entry is effectively identified by its position. It is convenient that the items are numbered 1, 2, 3, upwards, since their prices can then occupy adjacent cells in the array.

The selection of an item by means of an index (*direct indexing*) is fast and efficient. Most other methods of table look-up rely on searching, which is always very much slower and often means looking at every entry in a table.

CUSTOMERS' ACCOUNT

DISH	NUMBER	UNIT PRICE	TOTAL
34	3	73	219
57	2	49	98
49	1	115	115
67	1	2	2
18	4	32	128
		TOTAL	562
		VAT AT 8%	44
		TOTAL TO PAY =	606P

Figure 8.5

Fortunately, direct indexing is rather more flexible than has so far been apparent. The item numbers do not necessarily have to start at 1 and go up in steps of 1: it is enough if they form an arithmetical progression because it is then possible to use a *linear transformation*, a formula that converts the first item number into 1, the second into 2, and so on.

To illustrate this idea, suppose that in a seed catalogue the various orchids are listed as follows

Orchid Number	Price
1700	84p
1715	39p
1730	47p
1745	9p
1760	81p
....	...
2435	4p

Here the orchid numbers form an arithmetic progression. This implies that direct indexing can be used to look up the prices. A possible method would be to set up an integer array called *orchidprice* and to arrange for *orchidprice* [*1*] to hold 84, *orchidprice* [*2*] to hold 39 and so on. Then if *t* were a variable giving an orchid number, the price of that orchid could be written as

orchidprice $[(t-1700) \div 15+1]$

where the expression $(t-1700) \div 15+1$ is the appropriate linear transformation.

Sometimes the item numbers of entries in a table are not so conveniently arranged in an arithmetic progression. If you were to go shopping for a computer, you might find the following item numbers in the catalogue

360, 370, 650, 704, 709, 803, 1401, 1493, 1410, 1440, 1620, 1800, 1900, 2100, 2903, 2980, 3000, 4004, 4020, 4030, 4040, 5500, 6600, 6700, 7040, 7044, 7090, 7600, 8008, 8080

(This is not supposed to be an exhaustive list of all the computers ever built. It is just an example of what a second-hand computer broker might have in stock at any given time.)

This list is *sparse*: many numbers are simply not represented. Those that are present are scattered almost at random. One possible way of setting up a price list would be to declare an array with 8080 elements, in which element [2980], for instance, would be used to hold the price of a 2980, and element [2981] would contain a code like (– 1) to show that there was no computer in stock with that item number. In all, only 30 out of 8080 cells would be occupied. The efficiency of storage utilisation (30/8080 = 0.37 per cent) would be unacceptably low.

For the moment, it will be convenient to set aside the question of price and to consider a simpler problem: given a list of computer numbers, is a particular number (such as 1710) among them? In the following discussion the number that is the object of the search is called the *target* and, if it is found, a *hit* is said to have occurred.

As the list of items does not form an arithmetical progression, the only approach is to set up a table of item numbers and to search it, comparing the target with one entry after another, until either a hit occurs or it is certain that the target is not in the table at all. The table now only needs to have as many cells as there are different computer numbers. The following program reads a list of 30 computer numbers that are currently in stock, and then reports whether a given number is among them.

```
1  begin int q; [1:30] int stocklist;
2     for j to 30 do read (stocklist[j]) od;
3     int target; read (target); q:= 1;
4     while target ≠ stocklist [q] and q < 30 do q +: = 1 od;
5     if target = stocklist [q] then print ((target, "IS IN STOCK"))
6                                   else print ((target, "ISN'T IN STOCK.
                                        SORRY!"))
7     fi
8  end
```

```
 360   370   650   704   ⎫
 709   803  1401  1403   ⎬ List of computer numbers (30 in
  –     –     –     –    ⎪ all)
7090  7600  8008  8080   ⎭
1234                       Number to be looked up
```

The heart of the search is in line 4. The boolean-expression gives the condition that the search should go on to the next entry in the table by incrementing the pointer *q*. The test is twofold—firstly the current element *stocklist* [*q*] must be different from the target (if it is not, there is no point in searching any further), and secondly there must still be some entries left to search. The search stops if either of these conditions is **false**.

When the program reaches line 5 after the search, there are two possibilities. The search may have stopped either because there was a hit, or because the end of the table was reached without a hit. The conditional statement in line 5 is used to decide this question. If the item was found then *target* = *stocklist* [*q*] must still be true but, if the search stopped because the table was exhausted, it will be *false*.

Searching lists is a common and time-consuming activity on most computers. In practice, tables are usually set up once and for all, and then searched hundreds or thousands of times. Accordingly, if the above program is modified so that it searches not for just one model number, but for many of them one after the other, it will be much more realistic.

```
 1  begin int q; [1:30] int stocklist;
 2    for j to 30 do read (stocklist[j]) od;
 3    int target;
 4    while read (target);
 5    target ≠ 0 do
 6      q:= 1;
 7      while target ≠ stocklist [q] and q < 30 do  q+: = 1 od;
 8      if target = stocklist [q]
 9        then print ((newline, target, "IS IN STOCK"))
10        else print ((newline, target, "ISN'T IN STOCK. SORRY!"))
11      fi
12            od;
13  end
```

```
 360   370   650   704  ⎫
------ ------ ------ ------ ⎬  Data: list of computer numbers (30
7090  7600  8008  8080  ⎭      in all)

1234   803  4040  1900  ⎫  List of possible numbers to be
1910  3219  2903  1620  ⎭      looked up
0                          Terminated by a zero.
```

How quickly does this program do its job? Assume that the program looks up so many numbers that the time taken to set up the table in the first place is negligible; the loop in line 2 may then be ignored.

The total time needed depends on the data. As an example, consider the following 'reasonable' assumptions about the data

(1) that, in all, there are 600 items to be looked up

(2) that, of the 600 items, half are eventually found to be in stock and the other half are not
(3) that, for those items that are in stock, each item is equally likely.

Thus 300 items are not in stock and, of the remainder, about 10 will refer to each computer model in the list. What are the consequences of these assumptions?

As noted in chapter 5, most of the time in any program is spent in the inner loop. In the above search program the inner loop is in line 7: it is

while *target* ≠ *stocklist* [*q*] **and** *q* < *30* **do** *q*+: = *1* **od**;

On each circuit of the loop the machine calculates two relations and one assignment

target ≠ *stocklist* [*q*]
q < *30*
q+: = *1*

say, three operations in all. It also does an **and**, but as this takes very little time it can be ignored.

It is now possible to work out the total number of operations needed.

First, each data item causes the instructions in lines 4, 5, 6, 8 and (9 or 10) to be executed once. This is five operations per item, 5 ∗ 600 or 3000 in all.

Next consider those items that are not found in the list. The program must go through the entire list of 30 items in order to be sure that a given item does not appear, and so the loop in line 7 is executed 30 times for each such item. This is 3 ∗ 30 operations per item, or 3 ∗ 30 ∗ 300 = 27 000 in all.

Lastly, consider the items that score hits. Some will be near the beginning of the table, so that the loop in line 7 is only obeyed a few times, while others will be near the end. The average position, using the assumption of uniform distribution, will be in the middle of the table, giving 15 times round the loop. The total number of instructions here will be 3 ∗ 15 ∗ 300 = 13 500.

Adding up these three figures gives 3000 + 27 000 + 13 500 = 43 500 operations. This is an average of 72 operations per data item.

If the initial assumptions are changed and it is assumed that nearly every data item is found on the list, the average falls to 55; if hardly any are found, then it rises to 95. This is not an important difference: in both cases it is much higher than the number of operations needed if direct indexing could have been used.

The search time is long, and it is easy to see that, as the length of the list grows, the search time goes up roughly in proportion. Such simple methods are much too expensive to use if the table is longer than several hundred items, and a great deal of effort has been expended in devising quicker ways of searching.

One very simple method of improving the search time is to observe which entries are referred to the most often, and to rearrange the order of the table so that they appear near the beginning. This will greatly reduce the average number of times the inner loop is executed for items that are found, and if most of the items are hits there will be a dramatic reduction in search time. If the majority of the items are misses, the method makes very little difference.

An alternative method, which is particularly useful if there is a high proportion of misses, relies on keeping the entries in increasing order of size. When this is done the search can be stopped as soon as an entry in the table is found to be greater than the target: all subsequent entries must be greater still and the target number itself cannot be represented further down.

The modification to the program necessary to implement this method is less than laborious

```
    7  while target < stocklist [q] and q < 30 do q+: = 1 od;
```

It leads to a halving (roughly) of the time needed to be certain of a miss. However, the method does nothing to speed up hits, and cannot be combined with a method that puts common items first.

Inspection of the inner loop shows that it consists largely of tests. Each time around, the computer has to evaluate two conditions. To reduce these to one, if it were possible, would shorten the inner loop, and so save some time in the right place.

Suppose that a table with one spare element at the end is used, and that before beginning the search a copy of the target is inserted in this spare cell. The search is bound to succeed, and there is no need to test whether the list is exhausted. When the inner loop is finished, it remains to decide whether the hit is genuine or simply corresponds to the dummy element at the end of the list. A program with this modification would be

```
 1  begin int q;  [1:31] int stocklist;
 2    for j to 30 do read(stocklist[j] od;
 3    int target;
 4    while read(target);
 5    target ≠ 0 do
 6                 q:= 1; stocklist[31]:= target;
 7                 while target ≠ stocklist[q] do q+: = 1 od;
 8                 if q ≠ 31
 9                   then print((newline, target, "IS IN STOCK"))
10                   else print((newline, target, "ISN'T IN STOCK.
                       SORRY!"))
11                 fi
12               od
13  end
```

Here the inner loop has to be executed one extra time if the target is not on the list, and there is an extra operation in line 6. On the other hand, the inner loop now has only two operations. Using the same assumptions as before, the total operation count is $6 * 600 + 2 * 31 * 300 + 2 * 15 * 300 = 31\,200$. This is an average of only 52 operations per data item, an improvement over the first method of about 28 per cent. As the list size grows, and the inner-loop time dominates the program more and more, the improvement will creep up to about 33 per cent. Furthermore, the method can be used together with a list that has the common items near the top, thereby giving a further worthwhile improvement.

An interesting and effective way of searching a list, which can be much faster than any of the methods so far described (except for direct indexing), is called the *logarithmic search*, or sometimes the *binary chop*. It can be used with advantage whenever the items in the list are not in arithmetical progression but can nevertheless be placed in some well-defined order.

The logarithmic search is a formalisation of an everyday procedure. Imagine that you wish to look up a word in a dictionary. To find the right page, you would probably begin by opening the book in the middle, and the word that you found there would tell you whether to go forward or backward. Next, you would turn to the half-way point (as near as you could judge) of the part of the book in which the wanted word must be—that is, you would open it either a quarter or three-quarters of the way through. The word at the head of this page would tell you which quarter of the dictionary to concentrate on. Each further look would again halve the number of possible pages, and it would need only a few applications of this rule to land you on the page you wanted.

In its computer version, the method depends on keeping two 'pointers' to the list of items being searched. In the program given below they are called *high* and *low*, and they indicate the limits within which the target must lie. For example if *high* = 17 and *low* = 10 the target, if it is present in the list at all, must lie somewhere between element 10 and element 17.

Initially *high* and *low* are set to the extreme limits of the list. This is reasonable, since the target, if present, must be somewhere within the list; it will certainly not be outside, any more than a word you were looking up in a dictionary could ever be found outside the covers.

Each time round the cycle the list subscript that corresponds to the mid-point of the current range of search is calculated, dropping the odd half if need be. Since the mid-point does not change during any one execution of the cycle, and need not be remembered between one execution and the next, its value is ascribed to a *constant*, *mid*, of which the reach is restricted to the inner loop (lines 7–13).

The target is compared with the item at the current mid-point. There are three possible results.

(1) A direct hit occurs. If this happens the search can stop, and the fact can be recorded by setting both *high* and *low* to the current value of mid.

(2) The target is *less* than the item at the current mid-point. This means that it must be nearer the beginning of the list, and so the upper limit can be reset to mid – 1. There is no new information about the lower limit, and so it is left unchanged.

(3) The target is greater than the item at the mid-point. This implies that it

must be nearer the end of the list, and the lower limit is reset to mid + 1. The upper limit is unaltered.

As you can see, the effect of each cycle is either to find the item or to narrow the range of search by half. Eventually the limits will converge to the same value, possibly (but not necessarily) because a direct hit was scored. The possible area of search will then be one item, and a direct comparison will show whether the target is represented or not.

If the computer-stock program is modified to use the logarithmic search, it takes the following form.

```
begin [1:30] int stocklist;
   for j to 30 do read(stocklist[j]) od;
   int target;
   while read(target); target ≠ 0
      do int low:= 1, high:= 30;
         while low < high
            do int mid = (low + high) ÷ 2;
               if target = stocklist [mid]
                  then low: = mid; high: = mid
                  elif target < stocklist [mid]
                     then high:= mid − 1
                        else low: = mid + 1
               fi
            od;
         if target = stocklist [low]
            then print ((newline, target, "IS IN STOCK"))
            else print ((newline, target, "ISN'T IN STOCK. SORRY!"))
         fi
      od
end
```

In general, each stage in the process halves the area of uncertainty. Doubling the initial size of the list will only increase the number of stages needed by 1 so that, if n is the total size of the list, then the number of stages needed to search it is roughly log n (to the base 2). This is why the method has its name.

An analysis of the binary-chop algorithm shows that the number of operations per item is about $6 + 5\log_2(n)$, where n is the number of items in the list. In the table on the next page, the performance of the binary chop is compared with a simple search, in which the inner loop is deemed to use only two instructions.

The discipline of computing abounds in horror stories. A frequent theme concerns the unfortunate programmer who tests his program on a set of sample data and finds it works perfectly; but as soon as he puts it into serious use he discovers that it takes far more time than was expected. This can be very serious—consider a so-called 'daily run' that takes 25 hours.

Length of list	Number of operations per item	
	simple search	logarithmic search
(n)	$(5+3n/2)$	$(6+5\log_2(n))$
4	11	16
8	17	21
16	29	26
32	53	31
100	155	39
1000	1505	56
32 000	48 005	81
1 000 000	1 500 005	126

The usual cause of this type of disaster is a failure to analyse the algorithm at the heart of the program. When it happens, there are two frequent reactions. Some programmers identify the inner loop of their program and proceed to 'polish' it in various ways, for example, by removing one of the tests as in the simple search program above. This will nearly always give some improvement in the program's performance: some people claim to be able to double the speed of any program just by polishing the code, and they can usually prove their boast if challenged.

Other programmers might start by considering the algorithm itself. If they have enough experience, or if their reading is wide enough, they will know a few dozen good algorithms for handling various situations, and they will soon see if one of these can be adapted to solve the problem. A good programmer might replace a simple search by a logarithmic search, and improve the speed of his program by a factor of 50. This would reduce 25 hours to half an hour, so that further polishing to save another 15 minutes would simply not be worthwhile.

It is possible to formulate a general rule: that the choice of algorithm is much more important in determining the performance of a program than the slickness of the actual coding. Look after the algorithms, and the code will look after itself!

Now consider again the original problem of keeping a price list for items not in arithmetic progression. This is easily done with two arrays of the same size—one for the item numbers themselves and the other for their prices. It is arranged that, if an item number occupies a given cell in the first array, its price will be stored in the corresponding cell (the one with the same subscript) in the second array. Looking up the price of any item reduces to

(a) finding the item itself in the first list (by any method) and noting its subscript.

(b) fetching the price from the second array, by direct indexing using this subscript.

EXERCISES

***8.1** (1) Expand the declarations

```
[1:25] int quirk
[1:1000] bool able
```

***8.2** (2) Write a single boolean-expression that checks whether a purported orchid number (see p. 102) is valid. (Hint: Does your expression give correct results for, say, 1714?)

8.3 (2) Trace the logarithmic-search program on p. 108 for the data values 803, 76, 709, 10.

***8.4** (3) The second-hand-computer dealer observes that some machines are much more in demand than others. Queries are distributed as follows

370	40 per cent of all queries
8080	30 per cent of all queries
1900	20 per cent of all queries
360	8 per cent of all queries
All others	2 per cent of all queries

Using these figures, state a good method of deciding whether a given computer is in stock. Analyse it and calculate the average number of operations needed to look up an item.

8.5 (3) Trace the following program for a data value of 20. What do you think it does?

```
begin int n; read (n); [1:1000] bool sieve;
  for j to n do sieve [j]:= true od;
  for j to n
    do if sieve [j]
          then for k from j by j to n
            do sieve [k]:= false od;
            print ((newline, j))
       fi
    od
end
```

***8.6** (4) (P) Write a program that reads a number of integers (not exceeding 100), sorts them into ascending order and prints them out. The data consist of a stream of *positive* numbers terminated by 0 (which is not part of the set to be sorted).

9 *Some Applications of Arrays*

'Gentlemen — I have discovered the Fifth Dimension'

Dialogue from more than one SF film

In this chapter a few more rules about arrays are introduced, and some practical applications of these facilities are discussed.

The 'identifier table' used to illustrate various points in the chapter is a convenient mental model, but the reader is again warned against taking it too literally; in most practical cases the Algol 68 system will do something which is roughly equivalent, but more efficient, and usually more complicated.

The upper and lower limits of an array are called its *bounds*. The values in the array declaration may be specified much more flexibly than has so far been suggested. For instance, there is no need for the lower bound to be 1. Any other value will serve, although in practice 1 or 0 will be used most of the time, since there is rarely any reason to do otherwise.

In principle, the upper bound may have any value that is not less than the lower bound. However, the number of elements in the array is (upper − lower + 1), and there is always a practical limit to the amount of working store available.

In the declaration itself, the bounds need not be given as integers. Any unit that yields an **int** value will be accepted, provided that the computer can evaluate it at the time the declaration is obeyed, that the upper bound does not turn out to be beneath the lower bound and that the total number of elements is not too high. Arrays with bounds given as expressions are called *dynamic arrays*. They are useful in cases where the programmer cannot predict in advance the size of array that he needs. Consider, for example, a problem in which a stock list to be searched is not of fixed length but varies from day to day. The program that handles it must cope flexibly with the situation.

Basically, there are two ways of presenting the stock data to the computer. One method, which is already familiar, relies on following the actual data items with a distinguishable terminator. If the items are to be read into an array, the main snag is that the program does not know how many items there are until they have all been read in, and it is therefore impossible to declare an array of exactly the right size. The programmer has to guess an upper limit to the likely number of items and to use an array that, in his opinion, is big enough for any conceivable set of data. If he is a good programmer he will insert a trap that gives a warning message if his assumption is violated.

```
begin int count:= 0, item; [1:1000] int stock;
  while read (item); item ≠ 0
    do count+: = 1;
      if count > 1000
        then print ((newline, "STOCK LIST TOO LONG. ITEM",
                    item, "DISCARDED"
                  ))
        else stock [count]:= item
      fi
    od;
```

This simple method has the disadvantage that in most actual runs the number of items will be far less than the programmer's upper limit, and space will be wasted. This is sometimes a serious drawback.

The second method relies on counting the items beforehand and telling the computer how many there are in advance. Thus the data items themselves are often preceded by an integer that shows how many data items there will be. If there are six items, for example, the data would be punched as

```
6                         (item count)
49  128  19  77  14  813  (actual items)
```

Now exactly the right size of array can be declared

```
begin int count, item;
  read (count);
  [1:count] int stock;
  for p to count do read (stock[p]) od;
  . . . . . . . . .
```

This approach also has its drawbacks. It demands that the data items should be counted in advance, and if this is done by hand there is a high probability of error. A good programmer would use a 'belt-and-braces' philosophy: he would insist that the data had a recognisable terminator as well as an initial count, and his program would ensure that the two were in agreement and that the count itself did not overload the capacity of the machine.

Consider next the declaration of arrays. The rules governing array declarations are logical extensions of those discussed in chapter 7, and you will see that every aspect of declaring an ordinary variable or literal has its counterpart here.

The form

```
[a:b] int x
```

(where a and b are unitary clauses) is short for

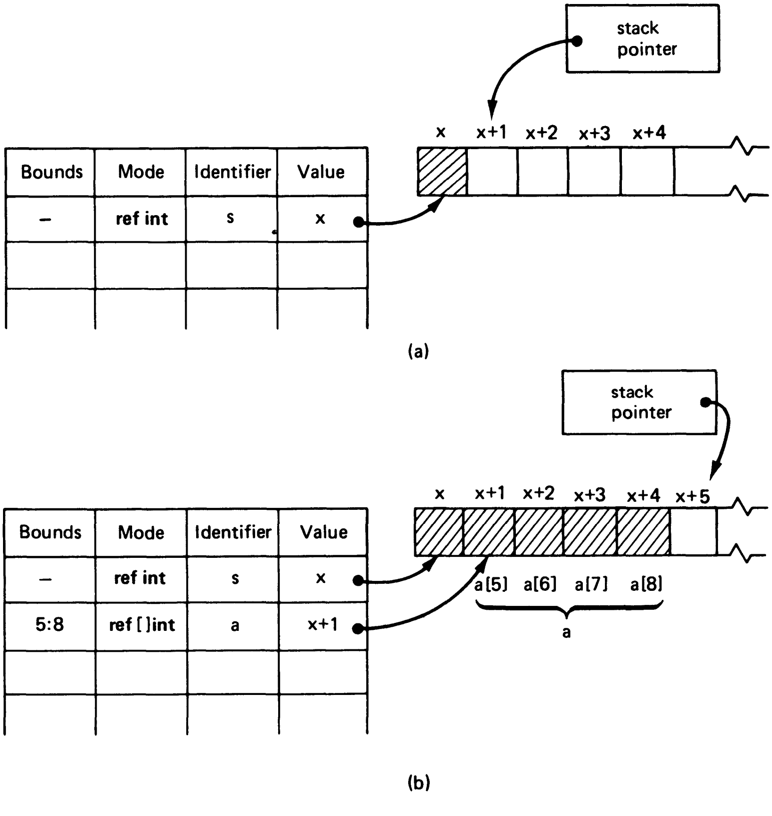
```
ref [ ] int x = loc [a:b] int
```

Here the left-hand side introduces an object called *x*, of mode **ref [ ] int**: this object may be called an *array variable*. The value of *x* is the position in the store of a row of (*b*−*a*+*1*) integer cells taken from the stack when the array is declared. You may suppose that the actual values of *a* and *b* at the time of the declaration are stored in the identifier table as well, so that references to array elements can be checked for accuracy of range and correctly handled. Consider the following sequence of code

```
begin int s;
    ref [ ] int a = loc [5:8] int;
    ...............
    print (a[s])
```

(Here the declaration of *a* is the full form of the contraction '[*5*:*8*] **int** a'.)

Figure 9.1a shows the situation near the beginning of the program when the declaration **int** *s* has been obeyed.



Bounds	Mode	Identifier	Value
–	ref int	s	x

Bounds	Mode	Identifier	Value
–	ref int	s	x
5:8	ref [ ]int	a	x+1

Figure 9.1 (a) Situation after obeying **int** *s; (b) situation after obeying* **ref** [ ] **int** *a* = **loc** [*5*:*8*] **int**

Figure 9.1b shows what happens when the declaration of the variable array a is executed. The bounds in the expression **loc** [5:8] **int** specify a row of four integers, and so this space is taken from the stack and reserved. The value delivered is the position of the first of these four cells. This value, together with the identifier *a*, its mode and bounds, are placed in the identifier table.

Later, when element *a* [*s*] is used (as in line 4), the position of the corresponding cell must be worked out by the following process.

1. The value of *s* is checked to see whether it lies within the bounds of *a* (5 to 8 in this case). If it does not, a dynamic fault occurs and the program stops.
2. If *s* is within bounds, the position of the actual cell is taken as (value of *a*) + (value of *s*) – (lower *bound*). For example, the address of *a* [7] would be (x + 1) + 7 – 5=x+3. A glance at figure 9.1 b shows that this is correct.

When a program is being executed, it may wish to remind itself of the bounds of one of its arrays. The operators **upb** and **lwb**, when applied to an array identifier, generate its upper and lower bounds, respectively. For example

[23:99] **bool** *qqq*; *print* ((**upb** *qqq*, **lwb** *qqq*))

would result in

```
+99      +23
```

Declarations also provide a way of setting up array constants—rows of items that never change from the time they are created.

An array constant is declared by the identity-declaration

[ ] **mode** name = row-display

Here the brackets on the left are always empty. The mode is any mode (**int**, **bool**, etc.) and the name is chosen freely. The row-display is a bracketed sequence of units separated by commas, which give the values for the elements of the array constant. The units must all yield values of the mode specified, but they may (as usual) be units of arbitrary complexity.

When the array constant is declared, the lower bound is automatically taken as 1, and the upper bound is always the same as the number of units in the row-display, so that each unit can be used to determine the value of one element. For example, the identity-declaration

[ ] **int** *primes* = (2, 3, 5, 7, 11, 13)

sets up a constant array of six elements, each of which is an **int**. The values of the elements are

primes [*1*] = *2 primes* [*2*] = *3 primes* [*3*] = *5*
primes [*4*] = *7 primes* [*5*] = *11 primes* [*6*] = *13*

and the mode of the constant itself is [ ]**int**.

An alternative way of declaring an array of mode [ ]**char** is to use a string-literal instead of a row-display. For example, consider

[ ] **char** *monarch* = "ELIZABETH"

Here *monarch* will have nine elements: *monarch* [1] is "E", *monarch* [2] is "L" and so on.

The declarations of array constants are not abbreviations or shortened forms. When an array constant is declared, no space is reserved on the stack; the values of the elements are recorded in the identifier table itself, where they cannot be changed during the life of the identifier.

Following the general pattern of chapter 7, it is now possible to return variable-array declarations and show how they can be initialised. This is simply done by following the ordinary declaration (which may be in full or shortened form) by the := sign and a suitable row-display that gives the starting values of the elements, for instance

[*1*:*3*] **bool** *h*:= (**true**, **false**, **true**)

which is short for

ref[ ]**bool** *h* = **loc**[*1*:*3*]**bool**:= (**true**, **false**, **true**)

Now the initial value of *h* [1] is **true**, but since *h* is a variable array, this can be changed by the program. The values of the elements of *h* reside in cells taken from the stack.

When array variables are initialised, the number of items in the row-display must agree with the number of elements as determined by the bounds of the array.

A string-literal may also be used to initialise a variable of mode **ref**[ ]**char**, but it is important to ensure (as always) that the number of characters in the string matches up with the bounds given in the brackets. Thus

[1:3] **char** *feline*: = "*CAT*"

is correct, but

[1:6] **char** *author*:= "*APULEIUS*"

is wrong.

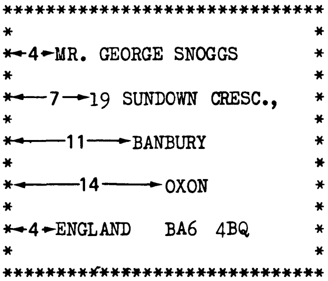
The declaration of array constants and the initialisation of array variables are again deceptively similar. The important differences are these.

(a) Array variables use space on the stack. Array constants do not.
(b) Array variables, if declared in the short form, must always include bounds in the square brackets on the left. Array constants never do.
(c) Initialised array variables use: =. Array constants use =.

It is now time for another example. Computers are often used to print labels for envelopes. Imagine a file of names and addresses, perhaps of people known to be generous towards the cause. The information is punched on cards, one card to each person. Each name-and-address takes five lines; but to save space the lines are pushed together as closely as possible and separated only by + signs. The file is terminated by a card with a slash in the first column.

```
W.A.MOZART,+7 SERAGLIO RD.,+WOKING,+SURREY,+UK+
I.STRAVINSKY,+99 RAKE ST.,+WIGAN,+LANCS,+UK+
C.DEBUSSY,+1 CLARE CRES.,+BANGOR,+CO. DOWN,+NI+
E.ELGAR,+222 POMP AVE.,+IRVINE,+AYR,+SCOTLAND+
/
```

Although this format is compact, and there are obvious advantages in having only one card for each name-and-address, it is not suitable for direct printing as a postal label. Something like figure 9.2 would be preferable.



The arrows show the number of spaces by which the various lines are indented from the left-hand border

Figure 9.2

A program that reads the file and generates a set of labels can be written using two new facilities, which have not yet been described

(1) *read* (*backspace*) moves the input stream back by one character so that the last character to have been read will be read again

(2) *print* (*qqq*), where *qqq* is an array variable or constant of which the elements are **chars** will simply print out all the characters of the array one after the other.

The program was designed in three stages, using the onion-skin approach.

At the outermost level the program reads a character from the beginning of a card. If it is a slash, it stops; otherwise it goes on to print a label for the name-and-address on that card and returns to read another card. Remember that, if the first character was not a slash, it was part of the name and must not be lost. The easiest way of keeping it is to push it back on to the input stream. This can be done by *read* (*backspace*).

At this stage, the program is

```
begin char x;
   while read (x); x ≠ "/"
     do read (backspace);
                  ┌─────────────────────────────┐
                  │ code to read a single card  │
                  │ and print out the address on│
                  │ it in the right format      │
                  └─────────────────────────────┘
     od
end
```

Next, consider the printing of the label itself. So that each label is separated from the next by a few blank lines, the program will begin by calling for some new lines. Then a line containing stars in positions 1 to 32 will be printed, followed by a line with stars in positions 1 and 32 only, the rest being filled with spaces.

The next step is to print the five lines of name-and-address given on the card. Each line is indented by a different amount, and is followed by another line that is blank except for the left- and right-hand margins.

Lastly, another full line of stars and a few more newlines will be printed.

The pattern that forms a line of stars is referred to in two places—at the top and bottom of the frame. To avoid writing the pattern out twice, it can be made into an array constant and referred to by its identifier. The same applies to the line that is blank except for the stars marking the side of the frame. In the program, these two patterns are called *top* and *side.*

The code to print a complete label now reads as follows.

```
. . . . . . . . . . . . . . . . .
   char top =  "********************************";
   char side = "*                              *";
   [ ] int margin = (5, 8, 12, 15, 5);

   print ((newline, newline, newline, top, newline, side));
   for count to 5
     do
                  ┌───────────────────────────────────────────┐
                  │ read and print one line of the name-      │
                  │ and-address, indenting it by (margin      │
                  │ [count]) spaces from the left.            │
                  └───────────────────────────────────────────┘
     od;
   print ((newline, top, newline, newline))
    . . . . . . . . . . . . . . . .
    . . . . . . . . . . . . . . . .
```

The right number of lines of address are obtained by using a counter called *count*. Each line must be printed with its own indentation, but there is no obvious connection between the line number and the amount it is to be indented. To take care of the indentation, a constant array of integers that gives the starting position of each of the five lines is declared. The array is called *margin*.

Lastly, consider the inner section of the program, which reads and prints one line of the name-and-address. The approach adopted in writing this section is to construct the complete line as a character array before it is printed. The section begins with a 'blank' called *line* that has stars at positions 1 and 32, and spaces everywhere else, and employs a pointer called *inset*, which indicates where the next character in the line is to be placed. The starting value of *inset* corresponds to the indentation needed for this line.

The code of the loop is quite simple: it reads characters from the card and plants them, one by one, into the array *line*. When it reaches the terminating +, it prints out the whole line followed by another blank, as required by the picture in figure 9.2. The code for the inner loop is

```
[1:32] int line:= "*                              *"
       co         <---------------30 spaces---------------> co
int inset: = margin [count];
char next;
while read (next); next ≠ " + "
  do line [inset]:= next; inset +:= 1 od;
print ((newline, line, newline, side))
```

Putting all the sections together, and moving *read* (*backspace*) down so as to conform with our convention that a declaration must come first, we obtain

```
begin char x;
 while read (x); x ≠ "/"
     do [ ] char top = "********************************";
        [ ] char side = "*                              *";
        [ ]int margin = (5, 8, 12, 15, 5);
       read (backspace);
       print ((newline, newline, newline, top, newline, side));
       for count to 5
         do
           [1:32] char line:= "*(←—— 30 spaces ——→)*";
           int inset:= margin [count];
           char next;
           while read (next); next ≠ " + "
             do line [inset]:= next; inset +:= 1 od;
           print ((newline, line, newline, side))
         od;
       print ((newline, top, newline, newline))
     od
end
```

This program is still deficient in one respect: what happens if a data card has more than, or fewer than, 5 '+'s on it? The case is left, literally, as an exercise for the reader.

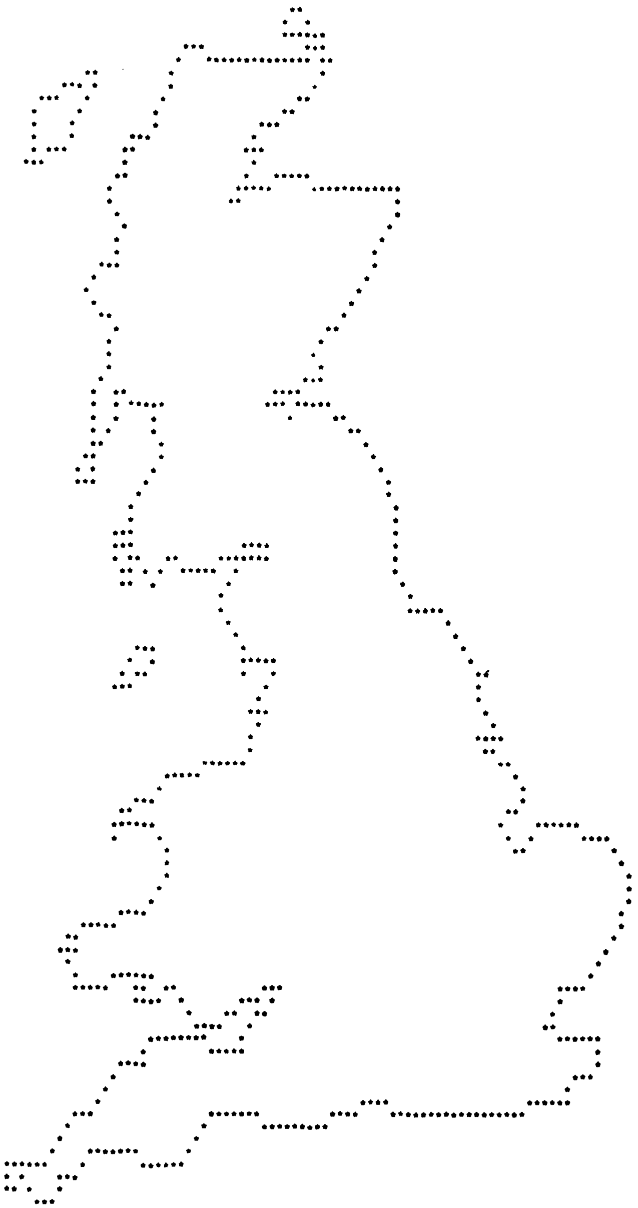


Figure 9.3

Finally in this chapter, consider another example. Computers are often used to collect and edit various types of geographical survey, such as investigations into the habitats of various plants and animals, or levels of pollution or the distribution of political opinion. The best way of presenting the results of these surveys is by maps with shaded regions. Good maps can be drawn directly by a computer using a graph plotter, but this is slow and expensive. If some degree of crudity can be accepted, serviceable maps can be made quickly and cheaply on a line printer. Figure 9.3 is an example of such a map. Great Britain and three of the remoter off-shore islands are shown, but the islands close to the mainland such as Skye or the Isle of Wight have been omitted since the resolution of the map (that is, the level of detail) is not fine enough to show narrow sea channels clearly.

The program that produced this map had to begin by reading a description of the coastlines of each of the four islands. The whole map itself is produced by printing stars at selected points in a 90 × 100 grid (9000 possible positions in all). One way of representing the necessary data would have been to define some arbitrary order for the 9000 points (perhaps from left to right within each row, and the rows one after the other starting at the top) and to state, for each point, whether it is on a coastline or not. This would have needed a large amount of data—9000 characters in all.

A somewhat better method would be to list the co-ordinates of each point on the coast as so many units East, and so many North of the bottom left-hand corner of the map. There are about 550 points and, since it would take four digits to specify the position of each one, 2200 characters in all (not counting spaces) would be required.

A further improvement can be obtained; as long as no sea crossings are made, each point on a coast is adjacent to another one, in one of eight possible directions. Once the co-ordinates of any one point on the coast of an island are known, the positions of all the other points can be specified by giving a chain of directions. Starting at the South-Western corner, the sequence

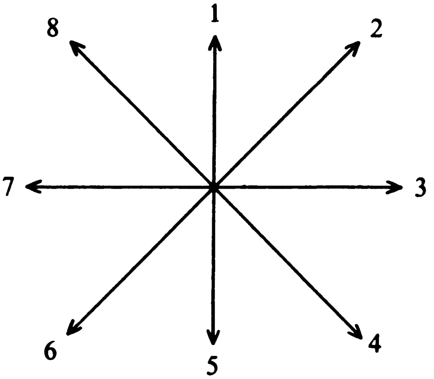
> NE, N, N, N, NE, E, E, NE, E, E, NE, E, S, SW, SW, SW, S, SW, W, W, SW, W, W

would produce an outline of the Isle of Lewis.

Using suitable (single-character) codes for the directions, this method could reduce the bulk of the data to about 560 characters.

Still further compression is possible. Since the points defining the coastline are often in straight lines, it may be useful to specify a code that represents a line as a single 'group', whatever its length. To this end, the first requirement is some means of indicating direction. It will be convenient to use a compass rose labelled as shown on the next page.

Note that the direction marked 2 is not exactly North-East because the grid of points that comes out of the line printer is not square: there are usually 10 points per inch horizontally, but only six vertically. Direction 2 is therefore $\tan^{-1}(0.6)$, or approximately N 35°E. Similar comments apply to directions 4, 6 and 8.



Next it is necessary to specify the rules for writing groups

(a) where the length of the group is only one unit, it is enough to give its direction; thus '1' (by itself) means 'go one step North'
(b) when the length is more than a single unit, the number of steps is given directly in front of the direction; '46' means 'go four steps South-West'.

Using this rule, the code for the shape of Lewis is the sequence

2 31 2 23 2 23 2 3 5 36 5 6 27 6 27

This more compact code brings the number of characters needed to represent the map down to 418, which is 20 times less than in the first method discussed. The reduction in the volume of data is important for two reasons: first it is cheaper to punch and store, and second it is easier to specify and check. For large problems—for example, where the detail of a map has to be fine enough to make a good large-scale drawing on a graph plotter—this saving can be extremely useful.

The compression only works because the map has certain regularities that a completely random collection of points would not have. Thus most of the grid points—about 94 per cent—are blank. The points on the coastlines form continuous closed loops, and often fall into straight lines several units long. All these features are used in the compression rules.

The data for the entire map are shown in figure 9.4. It was obtained in four steps.

(a) The outline was traced on to a sheet of 'layout planning paper'. This consists of a large chart printed with a grid having exactly the same spacing as the line printer, and is used for designing computer outputs.
(b) Using a ruler, the drawing was transferred to another sheet of squared paper, using only straight lines that join the centres of squares and run in one of the eight permitted directions. This was done so as to follow the actual coastline as closely as possible.

(c) A point on the coastline of one island was chosen at random, and its coordinates (with respect to the corner of the picture) were written down. Then, working continuously round the coast, each line was put down as a coded group. When the circuit was complete, a zero was put in as a terminator. The same process was repeated for the other islands. Finally another zero was added to the end of the entire data set.

```
78 15                                                      Coordinates of Dover
21 57 8 7 22 1 33 52 1 2 21 38 37 8 57 26 7 28
2 3 2 1 28 7 8 7 8 33 38 1 2 7 58 47 1 28 51
8 1 48 7 8 7 8 47 6 8 27 2 33 2 23 1 8
22 3 52 1 32 21 117 8 47 6 37 6 7 42 27 22 23
2 3 2 3 32 7 8 6 137 8 27 26 5 26 5 6 27
5 7 25 7 6 5 24 6 25 27 26 24 3 4 6 25
26 45 26 5 23 21 32 21 3 4 43 6 5 4 5 46 35 7 1 7            The coastline of
25 4 5 3 21 3 24 22 3 4 43 2 23 2 33 5 37 36                  Great Britain
5 34 25 2 33 5 36 23 26 5 6 57 6 47 26 27 6 7 6 5
2 43 24 25 36 37 6 47 6 7 6 23 6 4 5 43 2 53 6 7
5 33 12 3 34 33 2 3 2 23 2 23 26 7 26 5
47 8 77 6 5 37 46 27 46 57 25 3 2 24 23 2 1
23 22 63 4 53 42 63 4 83 2 33 2 33 4 173 2 53
1 2 23 2 0
16 46 32 23 5 6 7 6 27 0                                      The Isle of Man
4 87 2 31 2 23 2 23 2 3 5 36 5 6 27 6 27 0                    Lewis
38 97 1 2 3 4 5 23 5 27 8 27 0                                Orkney
0
```

Figure 9.4

The interesting point about using the computer to translate this mass of figures into a map is that the line printer can only move the paper in one direction — upwards. This implies that the sections of coast cannot be printed as they are read, but the whole map must somehow be built up inside the computer first. When it is completed, it can then be printed from the top downwards.

Fortunately a feature of Algol 68 makes the internal storage of the map a simple matter: it is possible to declare arrays not only of one dimension, but of any number — two, three or more.

One-dimensional arrays have already been discussed. They can be regarded as rows of elements of some mode like **int** or **char**. Each row has an upper and lower bound, and any element can be selected from the row by giving a single integer index.

A two-dimensional array is like a table: it has rows *and* columns, and to choose any element both the row and the column numbers must be given.

A typical declaration of a two-dimensional array might be

[1:3, 1:5]**int** *qcum*

(This is the shortened form of

ref[,]**int** *qcum* = **loc**[*1*:*3*, *1*:*5*]**int**

and the mode of *qcum* in this case is **ref**[,]**int**.)

There are two sets of bounds—[*1*:*3*] and [*1*:*5*]. Each set corresponds to one of the two dimensions. It is usual to think of the first set as defining the rows of the table, and the second set as specifying the columns. The array *qcum* may be pictured as having three rows numbered 1 to 3 and five columns numbered 1 to 5 as in table 9.1.

3	*qcum*[3, 1]	*qcum*[3, 2]	*qcum*[3, 3]	*qcum*[3, 4]	*qcum*[3, 5]
2	*qcum*[2, 1]	*qcum*[2, 2]	*qcum*[2, 3]	*qcum*[2, 4]	*qcum*[2, 5]
1	*qcum*[1, 1]	*qcum*[1, 2]	*qcum*[1, 3]	*qcum*[1, 4]	*qcum*[1, 5]
Row number	1	2	3	4	5
			Column number		

Table 9.1

The total number of elements is the product of the numbers of rows and columns. In this case it is 15.

Once the array has been declared, any element can be selected by using two subscripts. Table 9.1 makes this clear.

A three-dimensional array can be set up in the same way. It has three sets of bounds, and could be drawn as a solid 'brick'.

The rules of Algol 68 allow arrays with any number of sets of bounds, but in practice it is very rare to use three dimensions or more; not many problems seem to require it.

All of the facilities that apply to arrays of one dimension can also be used with 'higher' arrays. Thus any of the bounds may be declared as an expression, the elements may be of any one mode, and it is possible to declare higher array constants as well as variables.

The operators **upb** and **lwb** can be used on arrays with several dimensions, but it is necessary to specify clearly the bounds to which they refer. Accordingly, they are written in a different form: thus

x **upb** *y*

(where x is an integer and y an array) will give the upper bound of the x-th dimension of y. To give an example, following a declaration

[*23*:*67*, *99*:*1234*]**int** *s*

the command

```
print ((1 lwb s, 1 upb s, 2 lwb s, 2 upb s))
```

will produce

```
+23  +67  +99  +1234
```

The map printing program is as follows.

```
 1  begin [1:100, 1:90] char map;
 2    int x, y, xstart, ystart;
 3    [ ]int xinc = (0, 1, 1, 1, 0,−1,−1,−1);
 4    [ ]int yinc = (1, 1, 0,−1,−1,−1, 0, 1);
 5    int group;
 6    for j to 100 do for k to 90 do map[j,k]:= " " od od;
 7    while read(xstart); xstart ≠ 0
 8      do x: = xstart; read(ystart); y: = ystart;
 9        while read(group); group ≠ 0
10          do int direction = group mod 10,
11            length = (group > 10 | group ÷ 10 | 1);
12              to length
13              do map[y, x]:= "*";
14                x+ : = xinc[direction];
15                y+ : = yinc[direction]
16              od
17          od;
18        if x ≠ xstart or y ≠ ystart
19          then print((newline, "COASTLINE NOT PROPERLY
              CLOSED"))
20        fi
21      od;
22      for v from 100 by−1 to 1 do
23        print(newline);
24        for w to 90 do print(map[v,w])   od
25            od;
26      print((newline,newline,newline))
27  end
```

In this program the image of the map is constructed in the two-dimensional **char** array map. The variables *x* and *y* keep track of the current position; *xstart* and *ystart* are used to remember where each coastline starts, so that if it does not end in the same place an error message can be printed.

The variable *group* is used to read in the value of a group (like 1 or 46) as a pseudodecimal number. The inner loop, which is executed for each group, runs from line 12 to line 16. The direction is fixed throughout. Just before the loop is entered, the value of *group* is split up into a direction and a length. The *direction* and *length* are both declared as constants and given the correct values in line 11 and 12. Note the use of the short forms of **if – then – else – fi**.

The change in *x* and *y* on each step in the inner loop depends on the direction of the step. Using the compass rose, it would have been possible to write a section of code such as

```
if     direction = 1 then y+: = 1
   elif direction = 2 then x+ : = 1; y+ : = 1
   elif direction = 3 then x+ : = 1
   elif direction = 4 then x+ : = 1; y− : = 1
   elif direction = 5 then y− : = 1
   elif direction = 6 then x− : = 1; y− : = 1
   elif direction = 7 then x− : = 1
   else                    x−: = 1; y+: = 1
fi
```

but it is simpler to declare two tables, *xinc* and *yinc*, which give the correct increments for each of the possible directions. Thus, in lines 14 and 15, if *direction* = *3*, *x* is incremented by 1 and *y* is unchanged because *xinc* [*3*] = *1* and *yinc*[*3*] = 0.

Finally, in lines 22 to 26, the map is printed out with a double loop; *w* is used to scan the rows and *v* to scan the columns. The array is printed from the top downwards so as to get the map the right way up.

EXERCISES

9.1 (2) Consider the problem of reading in a stocklist that is preceded by an item count and terminated by a zero. Write a robust version of the input section on p. 112, which checks for as many errors as possible and prints warning messages as necessary. Assume that your computer cannot hold an integer array of more than 10 000 elements.

9.2 (1) Give the modes and the number of elements in the arrays identified in the following declarations. Also state the expanded forms of the declarations, where applicable.

(1) [*3*:*17*]**bool** *j*
(2) [ ] **char** *greeting* = "*HELLO*"
(3) [−*4*:*0*] **int** *q*:= (*34*, *97*, *143*, *77*, *12*)
(4) [*1*:*1*, *1*:*1*] **int** *z*
(5) [ ] **bool** *xx* = (**true**, **false**, **true**, **false**)
(6) [*3*:*5*,−*6*:*12*, *56*:*60*] **bool** *qq*

***9.3** (2) A 'chessboard' is declared as an 8 × 8 array of boolean elements, thus: [*1*:*8*, *1*:*8*] **bool** *chess*. Write a sequence of statements that will set each element **true** if it is a black square, and **false** if it is white; *chess*[*1*, *1*] should be black.

9.4 (2) Copy out the program on p. 118, showing the reach of each identifier. Make a list of the identifiers showing their modes.

9.5 (4) Consider the map-printing program on p. 124. Discuss how you would alter the program and the rules for preparing the data so as to allow the inclusion of political boundaries such as national or county borders.

9.6 (7) How would you adapt the map-printing program so as to distinguish land from sea by shading all the land inside the coastline? ('Shading' may be taken to mean printing a dot at every grid position.) If possible, the program should use exactly the same data as at present.

9.7 (3) A defect of the program on p. 118 is that it does not deal gracefully with the odd data cards that do not have the right '+'s on them. Explain how you would modify the data format and the program to overcome this difficulty.

***9.8** (6) (P) The game of 'life' is the recent invention of a Cambridge mathematician. It is supposed to simulate the life cycle of an idealised colony of bacteria. Initially, each germ lives in a cell somewhere in a square grid of cells. The adjacent cells may or may not be occupied, so that each cell has at most eight neighbours (diagonal neighbours count). Periodically the colony goes through a life cycle in which some new germs are born and others die. The rules are as follows.

(1) If a germ in a given cell has one neighbour or none it dies of loneliness.
(2) If a cell has four or more neighbours it dies of overcrowding.
(3) If an empty cell has exactly three adjoining cells that are occupied, a new germ is born there.
(4) The cells at the edge of the grid always remain empty.
(5) All the changes happen at the same instant.

An example illustrates the game. The starting position is

			+			
		+	+	+		
	+	+		+	+	
		+	+	+		
			+			

If the germs due to expire are ringed, and cells in which germs are to be born are marked with a dot, the following picture is obtained

		•	+	•		
	•	⊕	⊕	⊕	•	
	+	⊕		⊕	+	
	•	⊕	⊕	⊕	•	
		•	+	•		

In the next generation this leads to

		+	+	+		
	+				+	
	+				+	
	+				+	
		+	+	+		

The generation after this one is

```
      +
    + + +
  +   +   +
+ + +   + + +
  +   +   +
    + + +
      +
```

and so the colony continues to evolve for a number of generations, each producing a new pattern.

Write a program to simulate and print out 25 generations of the life game. Use 15 × 15 cells, and declare two arrays — one to store the current position, and one to store the changes due to happen at the next generation.

First, run the program with the starting pattern

```
*   *   *   *   *

*   *   *   *   *

*   *   *   *   *
```

Next, conduct some experiments and find another pattern that produces interesting results.

10 *Hierarchy, Procedures and Parameters*

> '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.'
>
> Robert Pirsig, *Zen and the Art of Motorcycle Maintenance*

Programming is a practical activity, dealing with complex situations in the real world. Programmers cannot sidestep this complexity; they must face it and accept its existence, handling it as well as they can.

Psychologists have made experiments to see what happens when people are confronted with problems of ever-increasing complexity. It appears that a person can cope with problems up to a certain level, after which he is suddenly overwhelmed and can make no further progress. The over-all difficulty of a problem is connected with the number of distinct ideas or concepts that have to be kept in mind at the same time. The simplicity or difficulty of each individual concept makes very little difference to the over-all effect.

Various estimates have been made of the number of concepts that an individual can retain at any one time. Although the number suggested varies from six to about twenty, all the experimenters agree that it is limited, and that if at any point, a problem involves more concepts than a person can handle, then for him that problem is insoluble.

In a computer program, a good measure of the number of concepts that the programmer must retain in his mind is given by the number of names in current use at the same time. The logarithmic search given in chapter 8, for example, involves the stocklist itself, the item being sought, the upper and lower limits of search, and the current position being examined. This gives five concepts immediately, and according to at least one estimate this is quite near the limit of the programmer's understanding.

Fortunately this does not mean that nothing more complicated than a logarithmic search can ever be programmed, because a variety of mental tools is available for controlling and reducing complexity. The most important of these is the idea of hierarchy.

A hierarchical structure may be thought of as a pyramid. The prime example is an army. In this case, one of the objects in the pyramid is a regiment. A general who is planning the strategy of a battle occupies a position at the top of the pyramid, and from that vantage point he can see each regiment in his command as a single unified entity. He can consider the relationships between the various regiments in the field, and he can issue orders that will be obeyed by an entire regiment as a whole. For the general, the regiment is one unit, so that planning a battle in which four regiments are engaged involves him with no more than about four concepts.

To the people inside it, the regiment is a much more complicated object. There are a number of different companies and various back-up services such as transport, catering and field hospitals. The commander of the regiment must understand the relationships between all these elements. He would describe the regiment in terms of several concepts, which interact in a complex way.

This type of structure is repeated all the way down the chain of command. An army is a far from simple object, but it can be controlled and used by one man because its hierarchical structure ensures that no-one in command of any part of it has to deal with more than a very few concepts. Hierarchy, then, is a means of imposing structure on a complex situation and of packaging up several distinct concepts into one 'higher' idea.

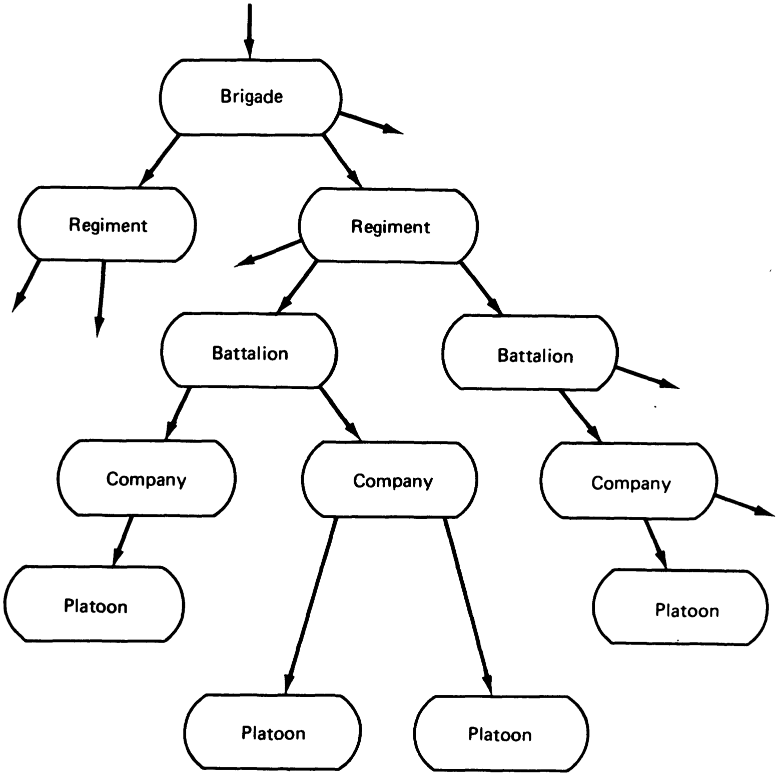


Figure 10.1

The most popular representation of a hierarchy is a tree.[1] Figure 10.1 shows the organisation tree of an army. In the military tree, the links downwards represent the chain of command, and the same lines upwards show the flow of responsibility.

The idea of hierarchy can be applied to the design of algorithms, and can be used to impose structure on a complex set of instructions. As a thought experiment, consider how the tree idea may be applied to an algorithm. The structure that emerges will be rather as follows.

At the top level, the entire algorithm consists of just a few self-contained steps with a simple and clear relationship to one another. A common top-level algorithm would be

```
collect data
deduce conclusions
write a report incorporating given conclusions.
```

Some of these steps might be very complex and so, in the next level down the tree, each step is further elaborated and subdivided into its own constituent steps. This structure is continued downwards until the really elementary parts of the algorithm are reached.

This sounds plausible, but so far it is the result only of introspection. The next step is to look at a practical case, and to find out how well it agrees with prediction.

The contents page of a do-it-yourself surgery book on my shelf is shown on the next page.

When the surgeon looks at the instructions for taking out the appendix, he finds that they are surprisingly short. They read

(1) put patient to sleep (see pages 16–17)
(2) cut patient open at tummy (see page 18)
(3) find the appendix (see pages 19–30)
(4) remove it with your knife (see page 1)
(5) put a stitch in the place from which you removed the appendix (see page 5)
(6) sew patient up (see pages 31–33)
(7) attempt to wake patient (see page 34)
(8) if successful, present bill for £150 (see pages 6–12); otherwise, see pages 13–15

If he now turns to the standard procedures, he finds that some of them are also short, because they refer to the 'basic elements' for the detailed descriptions needed. For example, the instructions for cutting a patient open at any given position read

(1) mark the line of the cut with a soft pencil
(2) hold the knife in your right hand (see page 1)
(3) cut firmly along your line

[1] In computer science, trees usually grow downwards.

(4) stop bleeding as necessary (see pages 2–3)
(5) repeat until you reach the inside of the patient.

The Compleat Chirurgeon

PART 1 The Elements of Surgery

How to hold a knife	1
How to prevent bleeding when you cut a patient	2–3
How to thread a needle	4
How to make a cross-stitch	5
How to present an account	6–12
How to fill in a Death Certificate	13–15

PART 2 Standard Procedures

How to put a patient to sleep	16–17
How to cut patient open at any stated place	18
How to find your way round inside	19–30
How to sew patient up	31–33
How to wake patient after the operation	34

PART 3 Complete Operations

To remove tonsils	35
To make a heart transplant	36
To remove appendix	36

Using these instructions, the tree of the appendix operation can be drawn. The result is shown in figure 10.2.

By and large, this structure is as expected. One important difference is that the structure is not really a tree at all, because several of the branches grow together. Some of the elementary instructions like holding the knife or making a stitch are used by more than one of the procedures above them in the hierarchy. This is unexpected, but may turn out to be useful.

The organisation of any complex algorithm into a hierarchy gives certain advantages, as the above example illustrates.

(1) A considerable compression in space is achieved. The complete set of instructions for removing the appendix, which would have taken 34 pages if written out in full, has been reduced to 9 lines. If the correctness of the

various procedures is assumed, then the accuracy of the instructions can be checked at a glance, and it is easier to ensure that the book as a whole is correct.

(2) Given the set of standard techniques and procedures, it is much easier to make up and describe new operations. A surgeon who is inventing a heart transplant operation no longer has to worry about details like making stitches: this is a standard technique, which he can assume and build on.

(3) As the procedures and techniques are isolated from one another, it is possible to alter and improve one of them without changing anything else in the book. Thus in the first edition of *The Compleat Chirugeon*, the section on putting a patient to sleep recommended the use of brandy and a large wooden mallet. By the time the next edition appeared, anaesthetics had been discovered, and the authors had rewritten this section completely. As far as I can tell by a close inspection of the two editions, nothing else was changed; most of the sets of instructions for complete operations still began

(1) put patient to sleep (see pages 16–17)

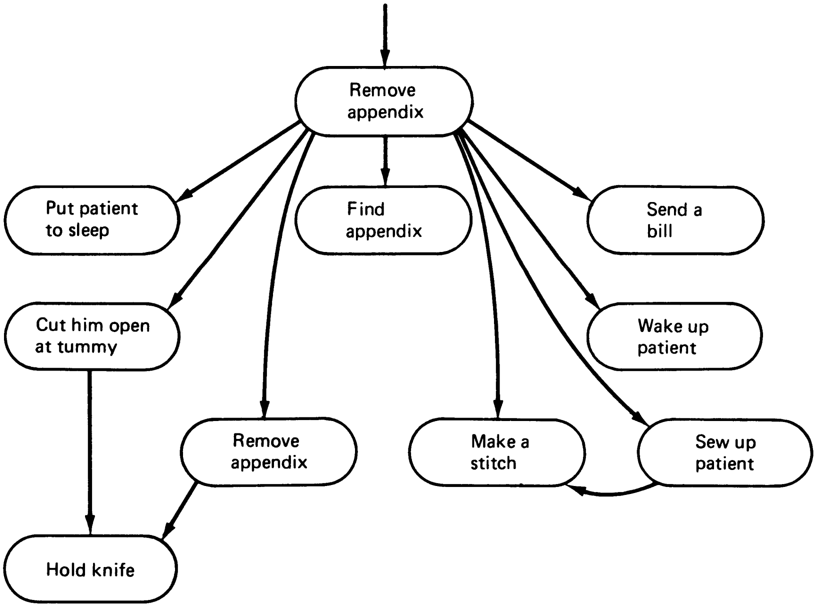


Figure 10.2

In Algol 68 the idea of program hierarchy is formalised in the system of *procedures.*

An Algol-68 procedure is a self-contained set of instructions, which the computer can be called upon to obey at any time. Like any other section of program, a procedure must be present in the computer when it is used, and for this reason it is convenient to regard it as an object sharing many of the attributes of other objects present in the computer such as variables or arrays. A procedure also has an identifier, a mode, a reach and a value. It is declared in a procedure declaration, which in its own way is similar to the declarations of other objects.

The simplest procedures of all are those designed to do a fixed job, which never changes no matter what the circumstances. The following is the declaration of such a procedure.

```
proc void doublearrow = void:
begin print((newline, newline,
        "  *                                   *  ", newline,
        " *                                     * ", newline,
        "*****************************************", newline,
        " *                                     * ", newline,
        "  *                                   *  ", newline,
        newline, newline
      ))
end
```

The procedure-declaration follows the normal pattern

mode identifier = value

This procedure is called *doublearrow*, and its mode is **proc void**.

The value of the procedure is written as the system word **void**, followed by a colon and a bracketed serial-clause; **void** implies that the procedure does not produce a result to hand back to the program that called it, even though it may print something. The full significance of the word will become clear later. In this case, the serial-clause does nothing except print a shape like this: <———>. This shape can be used to separate the different batches of output from some program.

To illustrate the declaration, consider the following program.

```
 1  begin int j: = 1;
 2    proc void doublearrow = void:
 3      begin . . . . . . . . . . .   ⎫
 |            . . . . . . . . . . .   ⎬  The serial clause of procedure
 |            . . . . . . . . . . .   ⎭  doublearrow, exactly as above
11      end;
12      while j < 4 do print ((newline, j)); doublearrow; j+: = 1 od;
13      while j > 1 do print ((newline, j)); doublearrow; j-: = 1 od
14  end
```

It is instructive to trace the actions of the computer through this program. The computer begins by obeying the declaration of *j*, and setting up an entry in the identifier table. The identifier is *j*, the mode is **ref int** and the value is the position of an integer cell in the store. The initial content of this cell is 1.

Next the computer comes to the declaration of *doublearrow*. As always, it sets up an entry in the identifier table. The identifier is *doublearrow*, the mode is **proc void**, and the value is the sequence

```
void:
   begin print ((. . . . . . . . . . . .
                 . . . . . . . . . . . .
               ))
   end
```

exactly as given in the program. The state of affairs is now as follows.

Pointer to stack

Mode	Identifier	Value
ref **int**	j	x
proc **void**	*double* *arrow*	**void**: **begin** *print*((*newline*, *newline*, “ * * ”, *newline*, “ * * ”, *newline*, “*******************************”, *newline*, “ .* * ”, *newline*, “ * * ”, *newline*, *newline*, *newline*)) **end**

Note that so far nothing has been printed. The body of the procedure has been stored as a constant of mode **proc void**; it has *not* been executed.

Next, the machine comes to the first **while** – **do** – **od** loop. Initially, *j* = 1 and so the condition is satisfied. The machine prints the value of *j* on a new line, and then comes to *doublearrow*, the identifier of the procedure. In this position the identifier constitutes an instruction to obey the procedure, by executing the code that constitutes its value. At this point, therefore, the computer carries out (for the first time) the instructions that it finds stored in its identifier table, and prints the doublearrow symbol.

After the procedure has been obeyed the program returns to the place whence it was called. The value of *j* is incremented by 1 and the loop is repeated. It is executed three times in all, and so is the second loop, which starts on line 13. A sketch of the over-all result is as follows.

```
       +1
←―――――――――――→
       +2
←―――――――――――→
       +3
←―――――――――――→
       +4
←―――――――――――→
       +3
←―――――――――――→
       +2
←―――――――――――→
```

This example shows that procedures can be declared once, but used in several places. The reach rules for procedure identifiers are exactly the same as for all other identifiers, and a procedure, once declared, can be called wherever its identifier is in reach.

Procedures of mode **proc void** which always do exactly the same thing, are rarely used. Normally, a procedure is required to do a job that is related to the current circumstances.

A cookery book might provide the following set of instructions for making jam.

Fruit Jam
(1) wash *fruit* and discard any that is rotten
(2) add an equal weight of sugar
(3) boil hard until *fruit* is soft
(4) continue to boil gently until a jelly-like consistency sets in
(5) bottle and seal quickly
(6) attach label saying '*Fruit* Jam'.

This recipe is intended to apply to all kinds of fruit. In reading it, the cook would replace the word 'fruit', wherever it occurred, by 'plums', or 'guavas' or the name of whatever particular fruit was at hand. The word 'fruit' does not mean itself—you cannot buy a jar labelled 'fruit jam'—but stands for any kind of fruit that the cook wants to use. In linguistic terms it is a kind of pronoun; in computing, it is called a *formal-parameter*. It is understood that a formal-parameter is never intended for use in its own right; it merely represents something that is unknown at the time the instructions are written, but will be determined when they are obeyed.

In Algol 68 a procedure can take one or more objects of any kind as formal-parameters. In the program given below, the procedure (which is declared between lines 2 and 9) takes an **int** as its parameter and decides whether its value is prime. The formal identifier of the parameter is *n*. This is indicated by the (**int** *n*) in line 2.

```
 1  begin
 2  proc (int) void primetest = (int n) void:
 3     begin int q: = 1;
 4        while q↑2 < = n and n mod q ≠ 0 do q+: = 1 od;
 5        if n mod q ≠ 0
 6           then print ((newline, n, "IS PRIME"))
 7           else print ((newline, n, "IS NOT PRIME. A FACTOR IS", q))
 8        fi
 9     end;
10  int z: = 2;
11  primetest (527);
12  while z < = 10 do primetest (z); z+: = 1 od
13  end
```

In this program the declarations continue as far as line 10. At this point, the identifier table contains two objects—a **ref int** called *z* and a procedure of mode **proc (int) void** called *primetest*. No code has yet been executed.

The execution of the program starts in line 11, and the first statement to be obeyed is

primetest (527)

This is taken as a call to the procedure. The actual-parameter, which replaces the formal-parameter when the procedure is run, is given in the brackets that follow the procedure identifier.

Obeying the call, the machine begins to execute the code that is the value of the procedure. The first item is the parameter specification **int** *n*. The computer obeys it as if it were a declaration of an **int** called n, with a value equal to the value of the actual-parameter supplied. In this case the equivalent declaration is

int *n* = 527

(Note that this is the declaration of an **int**, not a **ref int**.)

The next item in the code is the bracketed serial clause that forms the body of the procedure. This is executed in the normal way, but in the reach of an **int** called *n*, of value 527. Since 527 is 17 ∗ 31, the procedure ends by printing

```
+527  IS NOT PRIME.  A FACTOR IS+17
```

When the machine reaches the end of the serial-clause, it erases from the identifier table not only the local variable *q* but the parameter *n* as well. Then it returns to the next statement in the main sequence.

The next group of commands is the **while**–**do**–**od** in line 12. Initially $z = 2$ so the condition is **true**. The procedure is called with *z* as parameter. Now *z* is a **ref int**, but the value needed for the formal parameter is an **int**. Fortunately, a **ref int** can be coerced to an **int** by dereferencing (see chapter 7) and this is what is done. The procedure is effectively called with a parameter of 2. It prints

```
+2 IS PRIME
```

The loop is repeated until $z = 10$. The over-all output of the program is

```
+527  IS NOT PRIME.  A FACTOR IS  +17
  +2  IS PRIME
  +3  IS PRIME
  +4  IS NOT PRIME.  A FACTOR IS   +2
  +5  IS PRIME
  +6  IS NOT ORIME.  A FACTOR IS   +2
  +7  IS PRIME
  +8  IS NOT PRIME.  A FACTOR IS   +2
  +9  IS NOT PRIME.  A FACTOR IS   +3
 +10  IS NOT PRIME.  A FACTOR IS   +2
```

The mechanism for supplying parameters to procedures, seen at work in this example, is actually very powerful. The following is a summary of its main features.

(1) The procedure declaration must give the exact mode of any parameter.
(2) When the procedure is obeyed, each parameter description is taken as the declaration of a constant, with a value equal to the corresponding actual parameter, after any necessary coercions to get the modes to match.
(3) When the code of the procedure ends, the parameter names and any identifier declared locally in the procedure are deleted from the identifier table.

These rules always apply, and they can be used to decide what happens in any circumstance, no matter how complex.

Procedures often need more than one parameter. The various parameters can be listed one after the other and separated by commas, but it is necessary to remember that the mode of a procedure depends on its parameters. For instance, a procedure that took four parameters of modes **int**, **char**, **char** and **bool** might begin

```
proc (int, char, char, bool) void qbix = (int j, char x, char y, bool a) void:
```

The mode of *qbix* is

```
proc (int, char, char, bool) void
```

because these are the parameters it takes.

The form given above is quite clumsy; in particular it is evident that all the details of the mode of *qbix* could be deduced from the list of parameters in the brackets to the right of the = sign. Algol 68 permits a shortened form

```
proc qbix = (int j, char x, char y, bool a) void:
```

Another possible contraction is that, where two or more of the parameters are of the same mode, the mode system word need only be written once. This would give

```
proc qbix = (int j, char x, y, bool a) void:
```

When a procedure takes more than one parameter, the actual parameters that follow the procedure call must be given in the same order as the corresponding list of formal parameters. For instance, a call of *qbix* would be followed by brackets containing an **int**, two **chars** and a **bool** (or units that could be evaluated to these items) separated by commas, as follows

```
qbix(32, "A", "X", true)
```

(For an example of a procedure that takes these parameters, see exercise 10.4.)

When deciding what mode to give a parameter, it is usually quite clear whether it should be an **int**, a **bool** or a **char** (for instance). It can be less obvious whether to include a **ref** as well.

Remember that an object of mode **int**, **char** or **bool** has a value that is fixed throughout its existence. Now the lifetime of an actual parameter is exactly the same as one execution of the procedure, and so an object that has one of these modes can be used but not changed by the procedure (although on different calls various different values can be supplied). On the other hand, an identifier of mode **ref int**, although it also is fixed throughout its existence, represents the position of a cell in the working store. Given the value of the identifier, the procedure can alter the contents of the cell to which it points.

This leads to a simple rule. If a procedure is expected to alter the value of one of its parameters, the mode of that parameter should include a **ref** so that the procedure, when it is being obeyed, can have access to the position of the cell being altered. If the value is not to be altered, the **ref** is unnecessary and can result in a lot of redundant dereferencing.

In the following example the procedure is supposed to interchange the values of two integer variables.

```
begin proc swap = (ref int a,b) void:
   begin int q = a;
      a: = b; b: = q
   end;
   int j: = 17, k: = 39;
   print((newline, j, k)); swap(j, k); print((newline, j, k,))
end
```

If run, this program will print

```
+17          +39
+39          +17
```

Next consider the matter of procedures that actually return values. So far, all the procedures considered have done specific jobs, and most of them have used values that were supplied as parameters. Often a procedure is required to hand back its result to the program that called it. One example is a procedure that checks whether a given character is a decimal digit. Its parameter will be a **char**, and its result will be a **bool**—**true** if the **char** is a digit and **false** otherwise.

To write such a procedure only two new rules are needed.

(1) The mode of the required result must be written just before the colon in the procedure declaration, replacing the **void**, thus

```
proc digit = (char x) bool:
```

(note that the mode of digit is **proc(char)bool**).

(2) The value yielded by the body of the procedure must be the required result (and of course it must be of the right mode). Remember that the value of a serial clause is the value of the last (or only) unit in it. In all probability, therefore, the body of a procedure that is supposed to return a boolean value will end with a boolean-expression, and one that is expected to produce an **int** value will have an integer-expression as its last item.

Using these rules, the full declaration of the procedure digit might be

```
proc digit = (char x)bool: begin x > = "0" and x < = "9" end
```

In the second example, which will be given without comment, the procedure takes a number *n* as its parameter and delivers the nearest integral power of 2 which is equal to or greater than *n*.

```
proc power of two = (int n) int:
  begin int t: = 1;
  while t < n do t *:= s od;
  t
end
```

The next example uses the principle of procedures to obtain a hierarchical structure. It is required to build a program that reads in Roman numbers and prints out their arabic (decimal) equivalents, up to and including a terminator of CMXCIX (999).

Roman numbers are built up of letters ('Roman digits') with the following values

I	V	X	L	C	D	M
1	5	10	50	100	500	1000

Normally the letters that make up a number are simply added together; but when (reading from left to right) a letter is followed by one of larger value, the smaller value is subtracted instead. Thus VII = 5 + 1 + 1 = 7, but XIX = 10 − 1 + 10 = 19.

It is assumed that the Roman numbers in the data are separated by any characters whatsoever except Roman digits. This ensures that, apart from a missing terminator, there can be no detectable errors; the program is to extract Roman numbers from any arbitrary stream of characters. For example, given the input stream

ELIZABETH, VICTORIA, GEORGE VI

the program would be expected to produce the numbers 51, 104, 1 and 6 before running out of data.

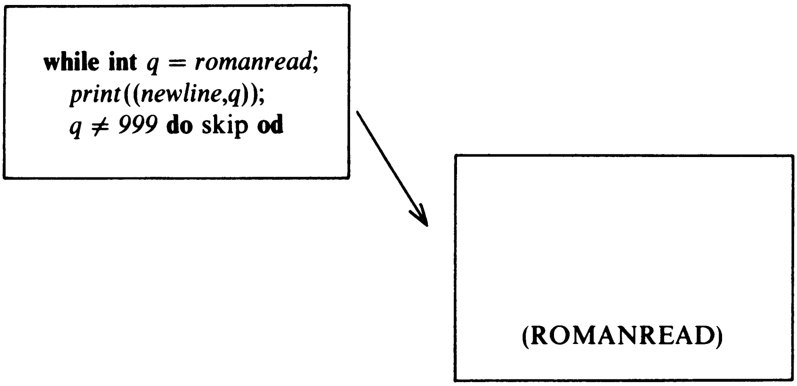
The program will be designed from the top down, using the onion-skin approach. The 'main' section, which is at the top of the hierarchy, has to read Roman numbers and print them, stopping after 999. A simple **while**–**do**–**od** loop will suffice.

```
while int q = romanread; print((newline, q)); q ≠ 999 do skip od
```

All is familiar except *romanread*, which determines the value of *q* each time round the loop. In fact, *romanread* is a procedure, which has now to be written. It will take no parameters, but will deliver an **int** result — the value of the next Roman number in the data.

This information about the procedure's identifier, parameters, result and function is called the *interface.* The user of a procedure needs to know everything about its interface but nothing about how it actually works — the knowledge is not relevant at this point and might even prove a hindrance, by bringing in unnecessary complexity.

At this point, the hierarchical form of the program begins to emerge.



```
while int q = romanread;
  print((newline,q));
  q ≠ 999 do skip od
```

The next step is consciously to move one level down the hierarchy and to begin to consider the details of the procedure *romanread* itself.

A good way to design an algorithm of any kind is first of all to do the job yourself, keeping to any limitations that the computer might have, and watching what you do very carefully. Put another way, you act out what the computer might do.

One of the chief limitations on a computer is that it can only read one character at a time. A trained eye can tell the entire value of a Roman number at a glance, but this is not how a computer does it. A more faithful imitation of a computer consists of getting a friend to reveal the characters in the data one by one. Here is an example, in which Bob is acting the computer, and Pam the input device.

Bob So far, the value of my Roman number (which I shall call *rn*) is zero. Can I have a character please, Pam.

Pam Space.

Bob That isn't part of a Roman number. Next please.

Pam Comma.

Bob That isn't part of a Roman number. Next please.

Pam C.

Bob That's a Roman digit worth 100. It must be the first digit of the Roman number; so I'll set *rn* to 100. Next character please.

Pam C.

Bob That's another Roman digit worth 100. It's not bigger than the previous one, so I add it to *rn*, giving 200. Next character please.

Pam X.

Bob That's another Roman digit, worth 10. It's not bigger than the previous one, so I add it to *rn*, giving 210. Next character please.

Pam L.

Bob That's another Roman digit, worth 50. It *is* bigger than the previous one, so I add it to *rn*, and subtract the previous one from *rn*, twice. This cancels out the previous addition, which was incorrect, and adds in *minus* the previous digit. The new value of *rn* is 240. Next character please.

Pam I.

Bob That's another Roman digit, worth 1. It's not bigger than the previous one, and so I add it to *rn*, giving 241. Next character please.

Pam Point.

Bob That isn't a Roman digit. The number must have ended, and its value is 241.

The process can now be formalised into the following steps.

(1) Read characters and discard them until a Roman digit is encountered. Set its value as the current Roman number *rn*.

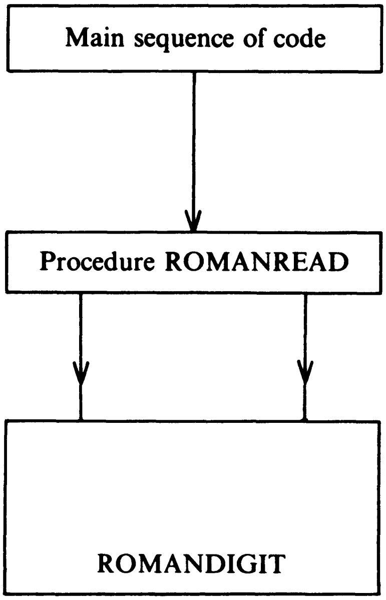
(2) Remember the current digit as the 'previous digit', and get the value of another character as the current digit. If it is not a Roman digit, then take *rn* as the value of the Roman number and end the procedure. Otherwise add the current digit to *rn* and, if the current digit is greater than the previous digit, subtract the previous digit from *rn*, twice. Repeat step (2).

The code for the procedure is a good deal shorter than the description. It includes a comment that describes the interface, which is standard practice in writing procedures. The procedure heading is the normal one for a procedure that takes no parameters but delivers a result.

```
proc romanread = int:
  comment This procedure reads the next Roman number from the data
          and delivers its value as an int. It takes no parameters
  comment
  begin int rn: = 0, cd: = 0, pd;
    while rn: = romandigit; rn < 0 do skip od;
    while pd: = cd; cd: = romandigit; cd > 0
           do rn+: = cd;
             if cd > pd then rn−: = (2 * pd) fi
           od;
  rn
  end
```

This is again quite simple except in one respect—once again a non-existent procedure has been used. This procedure called *romandigit*, is supposed to read the next character and decide whether it is a roman digit at all; if so, its value is delivered as a positive integer. Otherwise, *romandigit* produces the special marker value − 1.

This additional procedure introduces another step in the hierarchy, which is now as follows.



There are two links from *romanread to romandigit* because romandigit is called twice, in different contexts.

The code for romandigit is

```
proc romandigit = int;
  comment This procedure reads the next character from the data stream.
          If it is a roman digit (I, V, X, L, C, D or M) the procedure
          delivers the corresponding value. Otherwise it gives − 1.
  comment
  begin char x; read (x);
    if x = "I" then 1
     elif x = "V" then 5
     elif x = "X" then 10
     elif x = "L" then 50
     elif x = "C" then 100
     elif x = "D" then 500
     elif x = "M" then 1000
     else − 1
    fi
  end
```

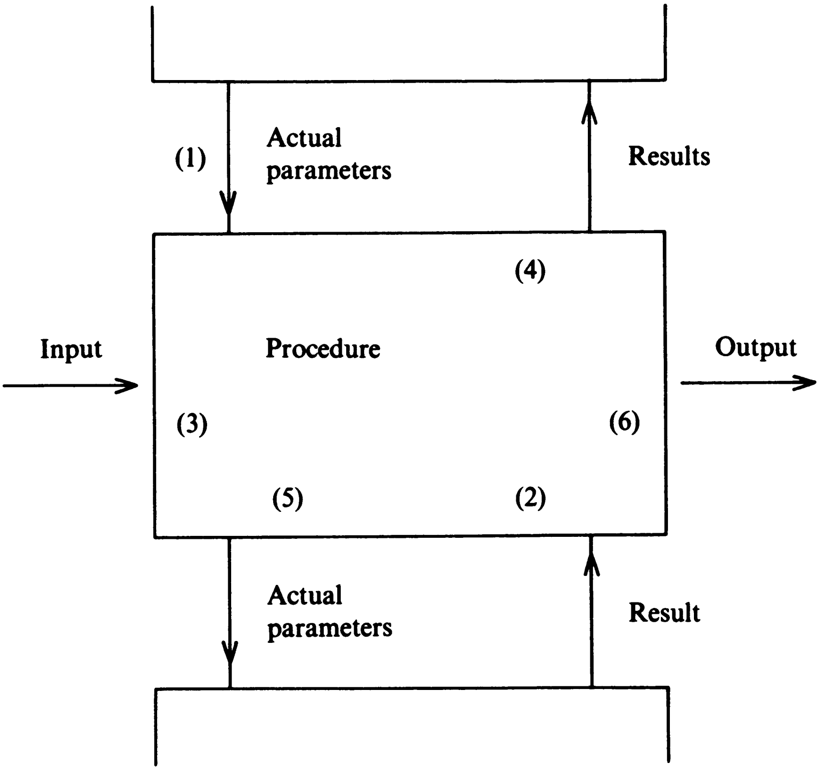
The entire program (omitting comments) is given below, using all the normal contractions including ( ) for both **if**–**fi** and **begin**–**end**.

```
begin
  proc romandigit = int:
    (char x; read(x);
      (x = "I" | 1 | : x = "V" | 5 | : x = "X" | 10 | : x = "L" | 50
        | : x = "C" | 100 | : x = "D" | 500 | : x = "M" | 1000 | − 1
      )
    );
  proc romanread = int:
    (int rn: = 0, cd: = 0, pd;
     while rn: = romandigit; rn < 0 do skip od;
     while pd: = cd; cd: = romandigit; cd > 0
       do rn +: = cd; (cd > pd | rn −: = (2 * pd)) od;
     rn
    );
  while int q = romanread; print ((newline, q)); q ≠ 999 do skip od
end
```

The procedures should be declared in the right order, for otherwise romandigit would be used before it was declared. This would be unacceptable to some implementations. Note that the declarations are separated by semicolons as usual.

In any hierarchy, there are formalised channels of communication between the various objects and also between some of the objects and the outside world.

A procedure in a program has a maximum of six different kinds of channel, as shown in the following diagram.



Three of the channels pass information inwards. They are

(1) actual parameters coming from a procedure above this one in the hierarchy
(2) results being returned by a procedure below this one
(3) new data being read from the input stream.

Three more of the channels pass information outwards. They are

(4) results returned to the calling procedure
(5) parameters to a procedure being called
(6) results being sent to the output stream.

It is very unusual for any procedure to use all six types of channel. Most procedures are designed to do a job that is internal to the program. They get their data as parameters and return their answers as values, and make no use at all either of input or of output.

To say that a channel of communication is formalised means that decisions have been made in advance as to exactly what kinds of things will be said and exactly *how* they will be said. The link between a pilot and the air-traffic control is a formalised channel, because the language is always English (irrespective of the nationality of the pilot or controller) and the only matters that may be discussed are those relating to flying.

The communications channels between procedures are very much of this type—the subject of discourse and the manner of expression must both be fixed in advance and included as part of the specification of any new procedures.

The rules about the reach of identifiers, which also apply to procedure declarations, make it possible to declare a procedure inside any serial-clause. For instance, it is possible to declare a procedure inside another procedure. Just as with other types of object, this facility offers a useful degree of protection whenever a procedure is designed to be called from only a limited part of the entire program.

At this stage, another type of error must be added to the list given in chapter 3. In using procedures, it is common to make mistakes in the number or mode of the parameters used. This can happen either in the declaration or in the call, but if the declaration is wrong the error may not be detectable at that point; instead a spurious 'error' will be reported at every correct call of the procedure.

EXERCISES

10.1 (1) Look at the address-printing program on p. 118. How many different concepts are involved in the inner loop, and what are they?

10.2 (2) Take any organisation you know (except the army) and draw a tree that shows the chains of command and responsibility.

10.3 (1) Describe the action of the following program (a sketch will do).

```
begin
  proc square = void:
  begin
    to 18 do print((newline,
                    "++++++++++++++++++++++++++++++++++"
                    ))
          od;
    print((newline,newline))
  end;
  proc triangle = void:
  begin
    for j to 18 do print(newline);
                    for k to j do print("*") od
                  od;
```

```
      print((newline,newline))
    end;
    square;triangle;square
  end
```

10.4 (3) Describe (with a diagram) the action of the following program.

```
  begin
    proc shape  = (int j, char x, char y, bool a) void:
    begin [1:j] char top, middle;
      for p to j do top[p]: = x; middle [p]: = y od;
      middle [1]: = x; middle [j]: = x;
      [1: 120−j] char marg; for p to 120−j do marg[p]:= “   ” od;
      print((newline, newline,
        (a|marg|“   ”), top));
      for q to j do print((newline, (a|marg|“   ”),
                             middle))
               od;
      print((newline, newline, (a|marg|“   ”), top))
      end;
    shape (10, “+”, “   ”, false);
    shape (20, “*”, “.”, true);
    shape (30, “   ”, “-”, false)
  end
```

10.5 (3) Trace the following program, decide what the procedure does and write down a description of the interface.

```
begin proc ps = (int x,y) int:
  begin int s: = 0;
    for t to x do s+ : = t↑y od;
    s
  end;
  int a,b;
  while read(a); read(b); a ≠ 0 or b ≠ 0
    do print((newline, a, b, “PS = ”, ps(a, b))) od
end
```

```
   3   2 ⎫
   7   1 ⎪
   0   3 ⎬   Data
   2   4 ⎪
   0   0 ⎭
```

***10.6** (3) The program below is supposed to read in two numbers and print their highest common factor. What is wrong with it? Give a corrected version.

```
begin proc hcf = (int a,b) int:
  begin while a ≠ b do (a > b|a − : = b|b − : = a) od;
   a
  end;
  read(a); read(b);
  print(("HCF = ", hcf(a,b)))
end
```

***10.7** (3) A student was asked to write a procedure which took a single character as its parameter, and determined whether it was a letter. The result was to be of mode **bool**. The student was given the procedure heading, and wrote

```
proc letter = (char x)bool:
  begin read(x); char r: = "F";
    if x < = "A" and x > = "Z" then r:= "T" fi
    r
  end
```

What are the mistakes? Why do you think the student made them? Give a correct version of the procedure.

***10.8** (5) (P) Write a procedure called *fineprint*, which takes the following three parameters

(1) (**int**) the value of a number to be printed
(2) (**int**) the total number of spaces to be used across the page
(3) (**bool**) **true** if a plus sign is required for positive numbers, and **false** if a space is to be used instead.

Your procedure should also

(a) always precede negative numbers with a minus sign
(b) insert extra spaces in front of a number, if they are necessary to make up the total required
(c) replace the number with a row of stars if it is too big to fit into the space available
(d) print 0 correctly.

Include the procedure in a program that calls it enough times, and with enough variety of parameters, to verify that it works under all conditions. (Hint: Start by writing a simple procedure that prints a positive number without any layout. Then embed the procedure in another that carries out all the necessary tests and outputs the preceding spaces, if necessary.)

11 *Recursion*

'This is the cock that crowed in the morn
That woke the priest all shaven and shorn
That married the man all tattered and torn
That kissed the maiden all forlorn
That milked the cow with the crumpled horn
That tossed the dog
That worried the cat
That killed the rat
That ate the malt
That lay in the house that Jack built.'

Nursery Rhyme

The last chapter gave all the rules that govern the use of procedures. The rules are simple, but their consequences are far-reaching, particularly since the principle of orthogonality ensures that they can be applied in combination with the other rules of Algol 68 and without mutual interference. This chapter will explore some of the more interesting and useful consequences of the basic rules.

One rule states that a parameter to a procedure can be specified as an object of any mode whatever. This immediately opens the possibility of two new types of parameters—arrays and procedures.

Procedures with array parameters are extremely common. They are used whenever a program needs to carry out a self-contained operation on an array. The following procedure picks out and returns the largest item in a one-dimensional array of integers.

```
proc max = (ref [ ] int q) int:
  begin int largest: = q [lwb q];
    for p: = lwb q+1 to upb q
       do if q[p] > largest then largest: = q[p] fi
       od;
    largest
  end
```

Once this procedure has been declared, it can be called in some sequence such as the following.

```
    [1:50] int pricelist;
    for a to 50 do read(pricelist[a]);
    print((newline, "THE MOST EXPENSIVE ITEM COSTS",
            max(pricelist), "PENCE"
        ))
```

The procedure *max* raises several interesting points. First, *q* is the formal parameter that stands for the array identifier. The mode of *q* is correctly stated as **ref** [ ] **int**. This tells the procedure that *q* represents a reference to a one-dimensional row of **int**s, but it does not indicate what the bounds of the row are. This is just as well, because the procedure is required to work on an array of any bounds. In any case, a form like **ref** [*1*: 50] **int** is not a legitimate mode for a formal-parameter.

When the procedure is actually used, it must of course take account of the bounds of the array that it is supposed to be searching. Since they are unknown when the procedure is being written, the operators **lwb** and **upb** are used to indicate them. Thus **lwb** *q* is always the lower bound (whatever it may be on various occasions) and *q*[**lwb** *q*] is always the 'first' element. Similarly, **upb** *q* is the upper bound.

You may have noticed that the mode of the parameter was given as **ref**[ ] **int** rather than [ ] **int**, even though the procedure makes no attempt to change any element in the array. The form [ ] **int** would also have worked correctly, but it would have been less efficient and would have taken more computer time. The reason lies in the amount of work necessary to set up the identifier table. If an array constant is used, the value of every element must be entered in the identifier table, and this means that each element must be copied in from elsewhere every time that the procedure is obeyed. If the parameter is given as an array variable, then only a position (which is a single quantity) is copied. The general rule, therefore, is to use the **ref** form unless the actual parameter is always going to be a row-display or a string-literal.

The next example is the binary-search program of chapter 8 rewritten with the actual search cast as a self-contained procedure. The advantages are clear.

(1) The 'main program' is simpler.
(2) The binary-search procedure can easily be extracted and used with other programs.
(3) If a better method than the binary search is developed, the search procedure can be replaced without altering anything else.

Compare the version below with that given on p. 108.

```
begin proc search = (int t, ref[ ]int w) bool:
   comment This procedure searches for an occurrence of t in
           the row of ints w. It assumes that the entries
           in the row are in increasing order of size. If
           successful, the procedure returns true
```

```
    comment
    begin int low: = lwb w, high: = upb w;
      while high > low
      do int mid = (low + high) ÷ 2;
         if t = w[mid] then low: = mid; high: = mid
            elif t < w[mid] then high: = mid − 1
            else low: = mid + 1
         fi
      od;
            t = w[low]
    end;
    comment Here begins the main program comment
    [1:30] int stocklist;
    for q to 30 do read(stocklist[q] od;
    int target;
    while read(target); target ≠ 0
      do if search(target, stocklist)
            then print((newline, target, "IS IN STOCK"))
            else print((newline, target, "ISN'T IN STOCK. SORRY!")
         fi
      od
end
```

Procedures can also be used as parameters to other procedures. A procedure amounts to a rule for doing something, and situations often arise where the precise rule to be followed depends on circumstances that cannot be known when the procedure is written. To give an example, suppose that you are writing a general-purpose tabulation procedure, which can be used to make tables of *any* function of an **int** variable over *any* range. The parameters of such a procedure will normally include

(a) a title for the tabulation
(b) a range for the variable (given as lower and upper bounds)
(c) a procedure that defines the rules by which the function is worked out; since this procedure will take an integer parameter and deliver an integer result, its mode is **proc(int)int**.

The following is a complete program that incorporates such a general-purpose tabulation procedure. It also includes two procedures—*square* and *factorial*—that are handed over as parameters. Study it carefully, particularly the way the parameters are specified and used.

```
begin
proc tabulate = ([ ]char title, int start, stop, proc(int)int rule):
  comment A general tabulation procedure. It prints the given
            title, and prints the values of rule (x) for all
            values of x between start and stop
  comment
  begin
    print((newline, newline, title, newline,
          "X  RULE(X)", newline
        )); for x from start to stop
    do print((newline, x, rule(x))) od
  end;
  proc square = (int j) int: begin j*j end;
  proc factorial = (int j) int:
  begin int s: = 1;
  for t to j  do s*: = t od;
    s
  end;
  tabulate ("LIST OF SQUARES", 10, 15, square);
  tabulate("TABLE OF FACTORIALS", 0, 8, factorial)
end
```

The output from this program would read

```
LIST OF SQUARES

   X        RULE(X)

  +10        +100
  +11        +121
  +12        +144
  +13        +169
  +14        +196
  +15        +225

TABLE OF FACTORIALS

   X        RULE(X)

  +0           +1
  +1           +1
  +2           +2
  +3           +6
  +4          +24
   5         +120
  +6         +720
  +7        +5040
  +8       +40320
```

As shown in chapter 10, programs can be built up like trees, with the various procedures using those below them in the structure. Can a procedure call itself? There are two questions to consider—is it possible, and is it sensible?

The rule of reaches says that an identifier exists from the point of its declaration up to the end of the relevant serial-clause. In the case of a procedure, the point of declaration is taken to be the place where the procedure identifier is first mentioned, so that the body of the procedure is within the reach of its own identifier. A procedure can call itself without violating the reach rules.

In order to examine the execution of a procedure that calls itself, it will be helpful to introduce the concept of *activation.*

When a procedure has been declared but not called, it is said to be *dormant.* There are no entries on the identifier table for any of its parameters or local names.

Suppose that the procedure is called. The parameters and local identifiers are declared, and the statements begin to be obeyed. The procedure is activated. Eventually it is completed; the parameters and local identifiers are wiped off the identifier table, and the computer returns to the point from which the procedure was called. The procedure returns to the dormant state.

The activation itself is an abstract notion, but the tangible evidence of activation is the presence, on the identifier table, of the parameters and identifier declared locally within the procedure.

What happens if a procedure calls another procedure? The second procedure is activated, but the first one does not return to the dormant state because its parameters and local identifier are still on the identifier table. The first procedure is merely suspended and, when the second procedure ends, the first one can continue with its original activation. In the case of a chain of procedures A, B, C and D that call each other, whenever D is activated, A, B and C must all be active but suspended. Hence it is evident that at one and the same instant, there may be several activations in progress, all but one of which are suspended.

It is now possible to examine the execution of a procedure that calls itself. Take as an example

```
proc sum = (int x) int:
  begin if x = 1 then 1 else sum(x-1)+x fi end;
```

and suppose that initially the procedure is called with a parameter of 3

```
sum(3)
```

When it is entered for the first time, the procedure declares its parameter, and sets up an entry in the identifier table — the mode is **int**, the identifier is *x* and the value 3.

Mode	Identifier	Value	
–	–	–	
int	*x*	3	(Activation of *sum*(*3*))

As the procedure runs, it soon finds that $x \neq 1$, so it attempts to evaluate *sum(3 − 1) + 3*. This involves *sum(2)*; so the machine follows the code exactly and calls the procedure *sum*. The first activation is suspended, and a second activation begins.

Mode	Identifier	Value	
—	—		
int	*x*	3	(Activation of sum(3) (suspended))
int	*x*	2	(Activation of sum(2))

There are now two simultaneous activations of the same procedure. Although the identifier table has two *x*s, there is no risk of confusion because the table is always scanned from the bottom upwards, and the more recent *x* hides the other.

The second activation again finds that $x \neq 1$, and so it tries to evaluate *sum(2 − 1) + 2*. This involves *sum(1)*; so the machine calls procedure *sum* a third time. The identifier table is now in the following state.

Mode	Identifier	Value	Value
—	—	—	
int	*x*	3	(Activation of *sum(3)* (suspended))
int	*x*	2	(Activation of *sum(2)* (suspended))
int	*x*	1	(Activation of *sum(1)*)

The third activation finds that *x* is *1*, and so it can produce the answer —1. It hands this value back to the second activation, and removes itself from the name table.

The second activation, which was suspended in trying to evaluate *sum(1) + 2*, is now told that *sum(1)* is *1*, and so it can continue. It reactivates itself, and completes the evaluation — 3. It then hands this number back to the first activation and removes its own entry from the name table, leaving only the original activiation. Finally, the first activation evaluates *3 + 3* or *6*, and passes this result back to the place from which it was called.

This explanation seems complicated, because the human mind is not well constructed to think recursively. Did you remember that the subject of the quotation at the head of this chapter was the cock that crowed in the morn? Nevertheless, it disposes of the first question — is it possible for a procedure to call itself and still to emerge with an answer? Apparently it is.

The second question asked whether it was sensible for a procedure to call itself. In some cases it is clearly absurd; if the square of a number were to be defined by

```
proc sq (int n) int:
   begin sq(n − 1) + 2 * n − 1 end
```

the procedure would keep on calling itself indefinitely without ever producing an answer. It is necessary to ensure that the procedure can only call itself a *finite* number of times, after which it must begin to unwind.

There is a wide variety of problems that can be solved by some variant of the following rule.

(a) If the problem is some particularly simple case, here is the explicit solution.
(b) Otherwise, here is how to work out a solution in terms of a different, but similar problem that is one step nearer to the simple case.

The following is an example.

To get to the bottom book out of a vertical pile of books

(a) if there is only one book in the pile, take it; it is the one you want
(b) otherwise, take off the top book and set it aside; then get the bottom book out of the vertical pile of books that remain.

Algorithms of this type are called *recursive.*

It has been proved that every iterative algorithm (that is, one that could be written with a **while – do – od** loop) has a corresponding algorithm that achieves exactly the same effect by recursion. Often the two algorithms are quite similar, but cases arise where one method is very much better than the other.

The following is a recursive version of the procedure *factorial* that was included in the general tabulation program

```
proc factorial = (int n) int:
   begin if n = 0 then 1 else factorial (n − 1) * n fi end;
```

The number of operations needed to calculate, say, factorial (6) is roughly the same in either version. In practice, however, the iterative version would win by a factor of two or thereabouts because of the operations on the identifier table implied by the extra procedure calls.

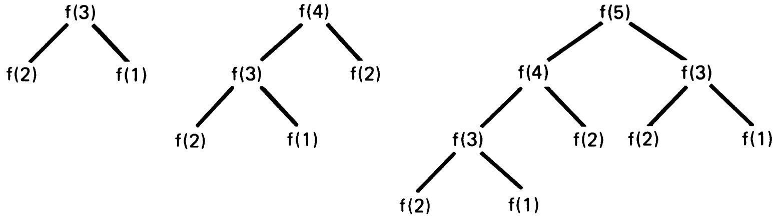
A set of numbers that often arises in various contexts (including some as unlikely as botany, breeding rabbits, picture framing, sorting and the analysis of algorithms) is this Fibonacci series. The first two terms of the series are both 1, and each subsequent term is the sum of the previous two. The series begins

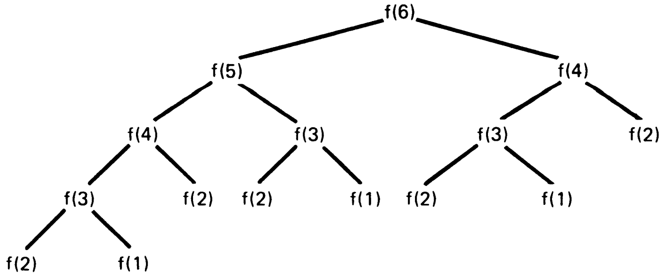
1 1 2 3 5 8 13 21 34 55 89 . . .

In Algol 68, the recursive definition of the n-th Fibonacci number is superficially attractive

```
proc fibonacci = (int n)int:
  begin if n = 1 or n = 2 then 1 else fibonacci(n−1)+fibonacci(n−2)fi
   end
```

Running a few trials of this procedure shows that it seems to work correctly when *n* is small, but that it begins to take an immense amount of computer time when *n* grows larger. On reflection, the reason is clear—for each activation when n > 2, there are *two* further activations, and so the amount of work grows extremely rapidly with each increment of *n*. The 'activation trees' for a few small values of *n* are as follows.





It is quite easy to show that, if a call of *fibonacci* with a parameter of 1 or 2 is described as an *elementary activation* (because in these two cases no further calls occur), it requires *fibonacci*(*n*) elementary activations to calculate *fibonacci*(*n*)! There is clearly massive duplication of work (for instance, in working out *fibonacci*(*6*), *fibonacci*(*3*) is worked out 3 times).
In this case an iterative solution is very much better

```
proc fibonacci = (int n) int:
  begin int a: = 1, b: = 1;
  for c from 2 to n do int dump = a+b; a: = b; b: = dump od;
  b
  end
```

If, say, the first 20 Fibonacci numbers are needed often, an even faster method would be to work them out once and for all, and to put them in a table, thus

```
int[1:20] fib;
fib[1]: = 1; fib[2]: = 1;
for c from 3 to 20 do fib[c]: = fib[c-1] +fib[c-2] od;
```

Subsequently, by writing, say, *fib*[*6*] instead of *fibonacci*(*6*), the required value could be in a single operation.

In other problems the balance lies the other way. The recursive solution is simple and straightforward, whereas the corresponding iterative solution is clumsy and hard to define.

In Hanoi (although some say it is in Benares), there is a temple where three diamond rods pointing to the sky are set in a sacred triangle in the ground. When the world was created, 64 pierced golden discs of graded sizes were placed on one of the rods, the largest next to the ground and the smallest at the top. The monks of the temple labour ceaselessly to transfer the discs to another of the three rods since, when all the discs have been moved, the world will end and the blessed state of nothingness will be attained. The immutable laws of the transfer say that only one disc shall be moved at a time, and that no disc shall ever cover another smaller than itself. For this reason the Creator in His wisdom provided three rods. With only two the task would have been impossible.

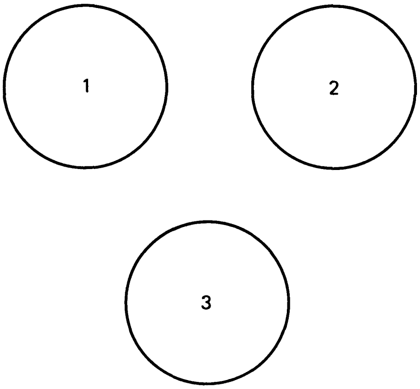


Figure 11.1

The sequence of moves needed is quite complex, and the minimum depends on the number of discs — with n discs the transfer cannot be completed in less than $2^n - 1$ moves. At this point, I would like you to try the problem for yourself. Please place four different coins in decreasing order of size (say 10p, 2p, 1p and $\frac{1}{2}$p) on the circle marked 1 in figure 11.1. Now transfer them to circle 3, following the sacred rules. Do not read on until you have tried!

If you succeeded in doing the task in the minimum number of moves, you would have done the following

$\frac{1}{2}$p from 1 to 3;	1p from 1 to 2;	$\frac{1}{2}$p from 3 to 2;	2p from 1 to 3;
$\frac{1}{2}$p from 2 to 1;	1p from 2 to 3;	$\frac{1}{2}$p from 1 to 3;	
10p from 1 to 2;			
$\frac{1}{2}$p from 3 to 2;	1p from 3 to 1;	$\frac{1}{2}$p from 2 to 1;	2p from 3 to 2;
$\frac{1}{2}$p from 1 to 3;	1p from 1 to 2;	$\frac{1}{2}$p from 3 to 2	

The algorithm for doing the transfer has a neat recursive form:

To transfer n discs from pin x to pin y

(1) if $n > 1$ then transfer $(n - 1)$ discs from pin x to the third pin (not x or y)
(2) move 1 disc from pin x to pin y
(3) if $n > 1$, transfer $(n - 1)$ discs from the third pin to pin y.

The algorithm is so simple that its correctness can be shown by an inductive argument.

(1) *Assume* that the algorithm is correct for some number of discs $n - 1$. In that case, it is also correct for n, because
   (a) during the whole of the transfer of $(n - 1)$ discs from pin x to the third pin, the disc that remains on pin x is the bottom one, and therefore larger than any that are moved; its presence will not lead to any violation of the basic rules, and the transfer of the $(n - 1)$ discs can go ahead as if the n-th disc were not there
   (b) at the end of the first stage, x will contain 1 disc, which is the largest one, y will be empty and the other $n - 1$ discs will be on the other pin; nothing prevents the transfer of the large disc from x to y
   (c) by a similar argument to (a) above, the $(n - 1)$ discs on the other pin can now be moved to y. The result is n discs in the right order.

   This proves that *if the method is correct for any number of discs n* $- 1$, *then it is also correct for n.*

(2) The algorithm clearly works for 1 disc, since the first and third moves are now unnecessary.

(3) For $n = 2$ the conclusion of (1) above is as follows: 'If the method is correct for 1 disc, then it is also correct for 2.' But it is known to be correct for 1; therefore it must be correct for 2. Similar arguments show that it must be correct for $n = 3, 4, 5$ or any other number of discs.

Using this algorithm, a robot could be programmed to move the discs by itself, but it is easier to write a program that prints out the necessary detailed instructions.

The algorithm will be embodied in a recursive procedure. Since, in its various activations, the procedure is called on to move discs between any two of the three pins, it will be convenient to take the source and target pin numbers as formal parameters, as well as the actual number of discs to be moved. If the pins are numbered 1, 2, 3 and *x* and *y* are any two of them, an expression for 'the other pin' is $(6-x-y)$.

```
begin
  proc move = (int n, x, y):
  comment This procedure generates instructions to move n
          golden discs in the tower of Hanoi from pin x to pin y
  comment
    begin if n ≠ 1 then move (n−1, x, (6−x−y)) fi;
          print((newline, "MOVE A DISC FROM PIN",
                 x, "TO PIN", y
               ));
          if n ≠ 1 then move (n−1, (6−x−y), y) fi
    end;
  move (3, 2, 3)
end;
```

The particular call asks for instructions to transfer 3 discs from pin 2 to pin 3. The procedure will print

```
MOVE A DISC FROM PIN  +2  TO PIN  +3
MOVE A DISC FROM PIN  +2  TO PIN  +1
MOVE A DISC FROM PIN  +3  TO PIN  +1
MOVE A DISC FROM PIN  +2  TO PIN  +3
MOVE A DISC FROM PIN  +1  TO PIN  +2
MOVE A DISC FROM PIN  +1  TO PIN  +3
MOVE A DISC FROM PIN  +2  TO PIN  +3
```

So far, all communication between procedures has been by parameter and result. This is a safe method, because each procedure has access only to the information necessary to do its own particular job. These limited and formalised communication channels act like bulkheads in a ship, and ensure that any damage caused by an error in one procedure can be quickly found and repaired, without the need to search the whole program. If a procedure is to be put in a library so that it can be used by any program, then the parameter – result method of communication is the only one possible.

In procedures that are designed for particular programs, on the other hand, there is the possibility of using a different communications system. Look at the following skeleton program

```
begin int a, x; [1:100, 1:100] bool q;
   proc first = . . .
   proc second = . . . .
   . . . . . . .
end
```

Here the reach of the identifiers *a*, *x* and *q* spans the whole of the program, including the bodies of the procedures *first* and *second* (always provided that they do not use *a*, *x* and *q* as local identifiers). *a*, *x* and *q* are called '*global variables* and together they constitute a *global area.* It would be possible to design procedures such that the ordinary communications mechanism was bypassed entirely—the procedures would collect their data from specified places in the global area, and signify their results by writing them back into selected global variables. For instance, one possible form of the *factorial* procedure would be

```
proc horrible = void:
comment This procedure calculates the factorial of the value in global
            variable x and puts its result in global variable y
   comment
   begin int     s: = 1;
       for t to x do s*: = t od;
       y: = s
   end
```

In use this procedure would be much more clumsy than one with the conventional communications mechanism. It would be necessary to declare *x* and *y* as global variables, and to write sequences like

```
x: = 6; horrible; print(y)
```

instead of

```
print(factorial(6))
```

It is true that successful programs can be written with this type of communication mechanism (indeed, it is all that a language like Basic provides) but a major drawback is that errors can be very hard to find. Consider a procedure A that calls another procedure B. In the conventional system, B has no access to the parameters of A and cannot possibly alter them (unless they are also deliberately handed on as parameters to B). In the global-variable method, B might well mistakenly alter the value of a global variable that was used as a 'pseudoparameter' to A so that, when control is returned to A, its environment has unexpectedly and subtly changed. It is as if communication by addressed notes in sealed envelopes were to be replaced by a huge blackboard on which everyone was free to write and to rub out as they pleased. It could work, but one incompetent would quickly drop the whole system into chaos. In general, the system of using global variables to communicate between procedures is not recommended.

EXERCISES

***11.1** (1) A recursive method for printing out a positive binary number *b* (as a sequence of 0s and 1s) is as follows

(1) if the number is 0, do nothing
(2) otherwise, if the number is 1, print a "1"
(3) Otherwise, print out the binary number $(b \div 2)$, followed by a "1" if *b* **mod** *2* = *1*, or a "0" if *b* **mod** *2* = *0*.

Write a recursive procedure called *binout*, which is callled with an **int** parameter that it prints out in binary.

***11.2** (4) By tracing or otherwise, discover the result of the following program, and write a suitable specification for the procedure in it.

```
begin
proc bubble = (ref[ ]int x) void:
   begin bool swap: = true;
      while swap do
                 swap: = false;
         for p from lwb x to upb x-1 do
            if x[p+1] < x[p] then int d = x[p]; x[p]: = x[p+1];
              x[p+1]: = d: swap: = true
            fi
                                od
                       od
   end;
[1:6] int z: = (3, 7, 12, 2, 1, 4);
bubble(z);
   for s to 6 do print(z[s]) od
end
```

***11.3** (2) Consider the program

```
(proc ssq = (int q) int:
   ((q = 1|1|q↑2+ssq(q-1)));
      print(ssq(1)); print(ssq(2)); print(ssq(5))
)
```

What is printed?

11.4 (3) Consider the procedure

```
proc multiply = (int f, g) int:
   (if f = 1 then g elif f = 0 then 0 else g+multiply(f-1, g) fi);
```

Is this ever a valid procedure for multiplying two numbers? If so, under what conditions? If valid, do you consider it efficient?

11.5 (2) Give the non-recursive equivalent of the procedure in exercise 11.4.

***11.6** (5) Give a non-recursive equivalent of the procedure asked for in exercise 11.3.

***11.7** (4) What do you expect the following program to print?

```
(proc search = (ref[ ]int a, x, proc(int, int)bool test) bool:
   (int j: = lwb a;
      while j < = upb a and not test (a[j], x) do j+: = 1 od;
            j < = upb a
   );
proc same = (int p, q) bool:(p = q);
proc less = (int p, q) bool: (q < p);
proc about = (int p, q) bool: (q > p-5 and q < p+5);
proc divides = (int p, q) bool: (p mod q = 0);
   [1:10] int j: = (22, 14, 173, 46, 307, 192, 72, 19, 88, 407);
   print(search(j, 46, same)); print(search(j, 87, same));
   print(search(j, 288, less)); print(search(j, 304, about));
   print(search(j, 301, about)); print(search(j, 96, divides))
)
```

***11.8** (4) (P) Some programs are difficult to analyse, and the programmer must resort to measurement to see how many times some statement or procedure is actually called. A method of counting the number of times a program passes a given point is to declare an **int** variable globally (so that it is accessible to all parts of the program) and to increment it at suitable places. For example, the procedure fibonacci discussed on p. 155 could have been monitored by writing

```
 1  begin int count;
 2     proc fibonacci = (int n) int:
 3     begin count+: = 1;
 4        if n = 1 or n = 2 then 1 else fibonacci(n-1)+fibonacci(n-2)fi
 5     end;
 6  print(("J      FIB(J)      CALLS OF FIB NEEDED", newline));
 7  for j to 10 do count: = 0;
 8              print((newline, j, fibonacci(j), count))
 9              od
10  end
```

In this program line 3 adds 1 to the count each time *fibonacci* is called. The results will be

```
     J              FIB(J)         CALLS OF FIB NEEDED

    +1               +1                    +1
    +2               +1                    +1
    +3               +2                    +2
    +4               +3                    +3
    +5               +5                    +5
    +6               +8                    +8
    +7              +13                   +13
    +8              +21                   +21
    -9              +34                   +34
   +10              +55                   +55
```

Ackerman's function takes two variables m and n, and uses the rule

```
$A(m,n) =$
  if $m = 0$ then $n+1$
  elif $n = 0$ then $A(m-1, n)$
  else $A(m-1, A(M, n-1))$
  fi
```

Use the method described to tabulate the number of calls needed to evaluate $A(1,1)$ to $A(3,3)$ (nine values in all).

12 *Real Numbers*

'Mr. Salter's side of the conversation was limited to expressions of assent. When Lord Copper was right he said, "Definitely, Lord Copper"; when he was wrong, "Up to a point".
"Let me see, what's the name of the place I mean? Capital of Japan? Yokohama, isn't it?"
"Up to a point, Lord Copper."
"And Hong Kong belongs to us, doesn't it?"
"Definitely, Lord Copper."'

Evelyn Waugh, *Scoop*

In the public eye computers are seen as the epitome of soulless accuracy — they cannot make any mistake, no matter how trivial or natural it may be.

On the other hand, manufacturers sometimes describe their computers as 'accurate to 10 decimal places' and, as has already been noted in this book, the over-all size of number that a computer can hold is limited, and a calculation that violates these limits will go catastrophically wrong. In other words, errors, small or large, are a distinct possibility.

Both views are substantially correct. To understand this apparent contradiction, it is necessary to consider the exact way in which information is represented inside a computer.

The elementary particle of information inside a computer is the binary digit, or '*bit*'. Each bit has only two possible values which are often called '1' and '0'. The bits are gathered together in groups of about six to represent characters, and about 24 — depending on the model of computer — to form integers in the binary scale. Since the number of different ways in which n binary digits can be arranged is 2^n, there are clearly certain limits on the information that can be stored. Six bits allows not more than 64 different characters, and integers are restricted to 10 million or so.

In many non-mathematical applications of computers, such as commerce, information retrieval or printing, these limits carry no disadvantages at all. The data types fit the problems exactly.

In scientific and engineering calculations, however, the requirements are somewhat different. In the first place, the numbers used vary from very large to very small — astronomers use light-years (9 457 280 000 000 000 metres) and computer engineers talk in terms of nanoseconds (0.000 000 001 seconds), picofarads (0.000 000 000 001 farads) and terabits (1 000 000 000 000 bits).

Secondly, the precision of the numbers varies but is not often particularly high. It is often said that a certain quantity is known to within x per cent. This means that, if y is an estimate of the number, the truc value lies somewhere between y(1 + 0.01x) and y(1 − 0.01x).

For example, the age of the earth is sometimes given as 4500 million years, plus or minus 5 per cent. This implies that the earth's true age is somewhere between 4275 and 4725 million years.

The precision of a quantity affects the number of decimal digits necessary to express its value. It would be wasteful and wrong to suggest that the age of the earth was 4 503 257 176 years because, if all these figures were really known, the precision of the estimate would not be 5 per cent but 0.000 000 001 per cent.

In practice, it is found that about nine or 10 decimal digits provide enough precision for almost every technical purpose.

In ordinary written calculations, it is usual to avoid numerous zeros and to keep the numbers easy to handle by using appropriate units — light years, nanoseconds or megabytes. These are constant multiples (or submultiples) of metres, seconds or bytes, which are the basic units. The name of the unit implies a scale factor, and it is also possible to handle such quantities by using the numerical scale factors explicitly. If the scale factor is a positive or negative power of ten, it may be convenient to use the 'scientific notation', as follows

$$50 \text{ picofarad} = 50 * 10^{-12} \text{ Farad}$$

or

$$128 \text{ megabytes} = 128 * 10^{6} \text{ bytes}$$

In this form, the sequence '* 10' is always present and carries no information. It is possible to drop it and to represent any number simply as a pair of integers

$$50, -12 \qquad \text{or } 128, 6$$

These two components are called the *mantissa* and the *exponent*, and the value of the number as a whole is taken as mantissa * 10^{exponent}.

Algol 68 offers a facility for handling these 'scientific' numbers. It is brought into use by the system word **real**. This is the name of a basic mode, similar to **bool** or **char** or **int**. Objects may be declared with arbitrary identifiers and modes **real** or **ref real** or [ ] **real** or **ref** [ ] **real**, etc., and then used as parameters or results of procedures. Do not be misled by the word **real**. It was chosen for historical reasons, and the objects declared to be of this mode are no more and no less real than objects of any other mode.

When a **real** variable is declared it also has a group of bits allocated to it; but they are divided, unequally, into two portions

$$\overbrace{\underbrace{\text{xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx}}_{\text{mantissa}}\qquad\underbrace{\text{xxxxxxxxxx}}_{\text{exponent}}}^{\textbf{real}\text{ number}}$$

The mantissa is represented as an integer in the 2s-complement notation but it generally has 30–40 bits, enough to represent decimal numbers of nine or 10 digits. The exponent is also a 2s-complement integer, but it is much shorter; it is usually limited to a range like -512 to $+511$.

Since both of the numbers involved are in binary notation, it is much easier for the system to use a scale factor that is binary rather than decimal. The value of the whole number is then taken as

$$\text{mantissa} * 2^{\text{exponent}}$$

For technical reasons, some machines use a fixed scale factor C as well, so that the value of every number is $C * \text{mantissa} * 2^{\text{exponent}}$. This can be disregarded at present.

Algol 68 allows all the normal operations of arithmetic to be performed on **real** numbers. The necessary mechanism is complex, but in most large computers it is provided as a physical attachment called a *floating-point arithmetic unit.* This unit is dedicated to calculating with **real** numbers, and it can generally do so nearly as fast as the rest of the machine can handle **ints**, **bools** and **chars**.

Smaller machines are not often fitted with floating-point units, and here **real** arithmetic has to be specially coded in terms of integers and booleans. The necessary program is built into the system in advance, so that it is just as easy for the Algol-68 programmer to use **real** quantities on a small computer as on a large one. The operations themselves are slower by a factor of 10 to 100. The decision as to whether to buy a floating-point unit for any machine is basically economic, and depends on the kind of use for which the machine is intended.

Real numbers can be written as literals in programs or supplied as data to be input by the read command. In each case they are normally written with decimal digits and the decimal point, but the decimal point may be omitted if it is not required. The $-$ sign may also be used where appropriate.

Real numbers are output by the print statement. They usually appear in scientific notation. For example, 128.37 would be printed as

$$1.2837{}_{10}2$$

(meaning $1.2837 * 10^2$) Note that the mantissa and exponent of the printed number are not the same as the mantissa and exponent of the representation inside the machine. The latter is binary, uses a whole number as the mantissa and has a scale factor that is a power of 2.

The version of division most commonly used with **real** numbers is written / (as opposed to $\div$). The result is in general fractional, and is calculated as accurately as possible. No remainder is thrown away.

The special operators $+:=$, $-:=$ and $*:=$ can also be used to alter the values of **real** variables. The special division operator is $/:=$, or **divab**, pronounced 'divide and becomes'. If x and y are **real** variables

$x/:=y$ is equivalent to $x:=x/y$

As an example, the following program calculates square roots by an iterative method; the program reads data items, and calculates and prints their square roots until it comes to an item that is zero or less.

```
1  begin real x:
2    while read (x); x > 0
3      do real y: = 1;
4        while if x > y↑2 then x − y↑2 else y↑2 − x fi > 0.01
5          do y: = 0.5 * (y + x/y) od;
6        print ((newline, "THE SQUARE ROOT OF", x, "IS", y))
7      od
8  end
   5.0  9.0  7.5  0            (Data)
```

The condition for repeating the inner loop is that the difference between the radicand (the number whose square root is to be found) and the square of the current estimate should not exceed 0.01. As the difference may be of either sign, the complex expression in line 4 must be used to ensure that the test works correctly.

All of the modes so far encountered represent objects that are different in kind from one another. Thus it makes no sense to assign a **char** value to an **int**, or to attempt to evaluate an expression like (*45* + **true**). However, **real** quantities form an exception to this rule, because from a logical point of view they include all the integers. Although the internal representation of integers and **real** quantities is entirely different, it makes good sense mathematically to add or multiply an **int** and a **real**, and so the machine arranges for conversion wherever necessary. This happens in two important situations.

(a) When an **int** value is assigned to a **real** variable, the **int** is converted to the representation of the **real** number with the same value. This coercion is called *widening*, and is the second so far encountered; the first was dereferencing, described in chapter 7.

(b) When an arithmetic operator like + or − has one **real** and one **int** operand, the **int** is converted into the corresponding **real** value. For technical reasons this is not called widening, but the effect is identical.

The effect of these two rules is to allow a mixture of **int** and **real** quantities to be used in the same expression. For example, the following is a procedure that calculates and sums the first eight terms of the series

$$x-x^3/3!+x^5/5!-x^7/7!+x^9/9!\ldots$$

For better efficiency the routine uses the fact that each term is linked to the previous term by a simple formula. If the first term is T_1, the nth term is given by the formula

$$T_n = -T_{(n-1)}*x^2/\{(2n-1)*(2n-2)\}$$

The procedure takes a **real** parameter *x* and delivers a **real** result.

```
    proc series = (real x) real:
      begin real t: = x, s: = x;
        for n from 2 to 8 do
          do  t: = -t*x↑2/((2*n-1)*(2*n-2));
                      s+: = t
          od;
          s
      end
```

In order to make fully effective use of **real** numbers it is necessary to know a few details about the way in which arithmetical operations work.

A fundamental aspect of **real** arithmetic is that numbers can have several different representations, all of which are correct. For instance, the number 19 can be recorded as (19, 0), or as (38, – 1), or as (76, – 2) or perhaps as (19456, – 10). Since the exponent stands for a power of 2, it is possible to double the mantissa and take 1 from the exponent as often as required without changing the value of the number represented. In mathematical terms

$$n * 2^{x} = 2n * 2^{(x-1)}$$

for all values of n and x. The only limit to this process is set by the number of binary digits in the mantissa; the value of the mantissa cannot exceed the largest number that these bits can hold.

The opposite process is also possible. Given a number with a large mantissa, it is possible to halve the mantissa repeatedly and to add 1 to the exponent in compensation. No error is introduced so long as the mantissa remains even (that is, divisible by 2). If the mantissa is odd, then the process will cause an error, but its relative size will depend on the size of the mantissa that remains. Consider the number 2 000 002, which can be represented as (2 000 002, 0). Since it is even, the process can be applied once without error; the result is (1 000 001, 1). Applying the process a second time gives (500 000, 2), an error of 0.0002 per cent. If the same process is applied to a small number, say 3, an error of 50 per cent is obtained immediately.

Since the mantissa is a binary number, the process of multiplication by two consists of shifting each digit up to the left by one place and writing a zero at the right. It is analogous to multiplying a decimal number by 10. Conversely, dividing by two is simply a matter of shifting the entire mantissa to the right, and throwing away any digits that fall off the right-hand side.

As will shortly become evident, the processes of **real** arithmetic sometimes require the mantissas of numbers to be shifted, usually to the right. Whenever it has a choice in the representation of numbers, the machine will select that with the largest possible mantissa (and therefore the smallest exponent). This ensures the smallest relative error if that mantissa is subsequently shifted to the right.

When integers are used in a program, it is known that all the arithmetical operations will be exact if the upper and lower bounds are not exceeded. This is not the case with **real** numbers, since the method of representation inevitably leads to certain errors.

There are two basic causes of inaccuracy. Firstly, many numbers cannot be represented exactly as floating-point quantities. These numbers fall into two groups — very large numbers and certain fractions. First consider large numbers. Assume (for the sake of argument) that the mantissa of your machine can hold integers up to 1 000 000. If a number exceeds this value, then it is scaled down by shifting it to the right (this amounts to a division by a power of two) and incrementing the exponent accordingly. Thus the number 8 000 000 would appear as (1 000 000, 3) or $1\,000\,000 * 2^3$.

The shift towards the right would in general lead to the loss of some of the least significant digits; for instance 8 000 003 would also appear as (1 000 000, 3). As far as this machine is concerned, the two numbers are indistinguishable.

It is worth noting that this type of error is quite different from that found in the case of integers. In one case the errors are catastrophic; in the other, they are guaranteed not to exceed some small fraction of the number represented. In the example given, the error is at most one part in a million.

Now consider fractions. The machine cannot be expected to store the values of irrational numbers like e and π exactly, because this cannot be done with a finite number of digits in any system of numeration. It may be a surprise to learn, however, that it is also impossible to represent accurately most 'ordinary' fractions, such as 1/3.

The reason lies in the use of two as a scale factor. Given any fraction to represent, the machine will try to convert it to an exact integer by doubling it and adjusting the exponent at each step. For instance, the number 17/64 would be doubled six times to give the representation $(17, -6)$, which is correct. Unfortunately the process only works when the denominator of the fraction is an exact power of two. In all other cases, the number can be doubled indefinitely, and the fraction will never disappear. When the number has reached the largest value that the mantissa can hold, it still has a fractional part, which must be discarded if less than 1/2 or rounded up to 1 if greater than 1/2. Consider the case of the number 1/3. After being doubled 21 times, it becomes 699 050 2/3. The best representation possible is $(699\,051, -21)$. The error is only 1/3 part of one in a million, but the representation is *not* exact. If the machine takes its value of 1/3 and adds it to zero three times, the result is *not* 1, although it is very close.

Ordinary decimals like 0.123 are a shorthand way of writing fractions whose denominators are powers of 10—0.123 means 123/1000. Unhappily, powers of 10 are not powers of two, and this implies that most decimal fractions cannot be represented exactly by binary floating-point numbers. The exceptions are those decimal fractions that are negative powers of two, such as 0.5, 0.25, 0.125, 0.0625, and their multiples.

Although exact representation is impossible, the relative inaccuracy is always small. In the above example, it would never exceed one part in a million, and in systems with mantissas capable of holding 12-digit decimal numbers, the error would be less than one part in an (English) billion. For most practical purposes this is quite negligible, but it means that great care is necessary when **real** variables are used to control loops. Consider the following programs, all of which are supposed to print out x^2 for all values of x between 0.1 and 1, going up in steps of 0.1.

```
(a) begin real x: = 0.1;
      while x < = 1 do print ((newline, x, x↑2)); x+: = 0.1 od
    end
```

This version, which looks so straightforward, may well go round the loop one time fewer than is intended. If the internal representation of the number 0.1 is very slightly larger than its true value (as would happen if the remaining fractional part were rounded up when the number was converted), then the result of adding 0.1 to the original value of *x* nine times would not be exactly 1, but some number that is a little larger (like 1.000000000003). This is enough to make the condition *x* < = *1* fail. In this example, where the program prints a line each time around the loop, the error would be noticed and corrected, but a similar calculation that was purely internal could lead to incorrect results.

```
(b) begin real x: = 0.1;
      while x < 1.05 do print ((newline, x, x↑2)); x+: = 0.1 od
    end
```

The second version will work correctly, but it has been necessary to take the trouble of using a value that is safely between two possible increments, so that the calculation can go ahead when *x* is about *1.0*, but will stop when *x* is about *1.1*.

```
(c) begin
      for j to 10
        do real x = 0.1 * j;
          print ((newline, x, x↑2));
        od
   end
```

Here the integer *j* is used to control the loop, and to generate a new value of *x* each time around. The accumulation of errors is avoided and the loop is guaranteed to work correctly. This is the recommended method; loops that are to be repeated a fixed number of times should always be controlled by integers.

The second source of error in **real** arithmetic lies in the arithmetic processes themselves. All of the four basic operations are subject to error, even if their operands are initially correct.

First consider division. If one number is divided by another, then in most cases the result will be neither an integer nor a submultiple of a power of two, and will not be susceptible to exact representation. The machine will have to use rounding to get the closest value it can.

The situation with multiplication is similar. If two numbers each with n-digit mantissas are multiplied, the product will have 2n digits, of which only the most significant n can be used in the result. The least significant half of the product must be thrown away.

Although division and multiplication usually give inexact answers, the error introduced is always relatively small—it is comparable to the error in representing a single quantity.

With addition and subtraction the position is even more treacherous. The two processes can be considered together, since subtraction may be regarded as negative addition.

Before the mantissas of two numbers can be added, it is necessary to arrange that they have the same exponent. (This is equivalent to 'lining up the decimal point'.) The machine takes the number with the smaller exponent, and shifts its mantissa to the right, incrementing its exponent by 1 at each step, until the exponent of the larger number is reached. As the mantissa moves down, its least significant digits are thrown away, introducing an error. In the limit, when the difference in the exponents is greater than the number of bits in the mantissa, the *whole* of the smaller number is thrown away and the 'addition' makes no difference to the larger number.

Under certain conditions this effect can make a substantial difference to the results of a calculation. Consider the situation in which many very small numbers x[1], x[2], . . . , x[n] are to be added to some large number y. Whereas adding each of the small numbers individually to y will leave y unchanged, adding all the small numbers together may produce a sum large enough to alter the value of y. In other words, the difference between

$$(x[1]+x[2]+x[3]+x[4]+\ \ldots\ +x[n])+y$$

and

$$(\ldots(y+x[1])+x[2])+x[3])+x[4])+\ \ldots\ +x[n])$$

is significant.

The general principle that emerges is that, where many numbers are to be added up, it is more accurate to start with the smaller ones.

Another danger with **real** addition and subtraction is the sudden increase in relative error that can take place if two numbers of almost the same size are subtracted from one another. This is not a result of the way numbers are represented, but is an inherent mathematical phenomenon. The effect occurs because the result is much smaller than either of the two numbers, whereas the absolute error that each one contributes remains the same. To give an illustration, consider a situation involving two numbers a and b. It is known that $a = 100 \pm \frac{1}{2}$ and $b = 90 \pm \frac{1}{2}$. The relative error in a is about 0.5 per cent, while that in b is about 0.55 per cent.

Consider evaluating the expression $(a-b)/(a\uparrow 2-b\uparrow 2)$. If, in working out each part, limits are set on its possible values, it will be possible to keep track of the relative precision as the calculation proceeds. For example, the smallest possible value of (a − b) is ($99\frac{1}{2} - 90\frac{1}{2}$) or 9, and the largest possible value is ($100\frac{1}{2} - 89\frac{1}{2}$) or 11. This means that (a − b) = 10 ± 1, which implies a precision of 10 per cent.

The various stages of the calculation are set out in table 12.1. It appears that, when the final value is reached, the relative error has increased by about 40 times.

If, however, the initial expression is rewritten as 1/(a + b), and evaluated in this form, a result of 0.005263 is obtained with a relative error of 0.5 per cent.

Quantity	Nominal value	Lower limit	Upper limit	Precision, per cent
a	100	$99\frac{1}{2}$	$100\frac{1}{2}$	0.5
b	90	$89\frac{1}{2}$	$90\frac{1}{2}$	0.55
(a − b)	10	$9(99\frac{1}{2} - 90\frac{1}{2})$	$11(100\frac{1}{2} - 89\frac{1}{2})$	10
a↑2	10 000	$9900\frac{1}{4}(99\frac{1}{2}^2)$	$10\,100\frac{1}{4}(100\frac{1}{2}^2)$	1
b↑2	8 100	$8010\frac{1}{4}(89\frac{1}{2}^2)$	$8190\frac{1}{4}(90\frac{1}{2}^2)$	1.1
a↑2 − b↑2	1 900	1710(9900 − 8190)	2090(10 100 − 8010)	10
a − b	10/1900	9/2090	11/1710	
(a↑2 − b↑2)	= 0.005263	= 0.004306	= 0.6433	20

Table 12.1

It is often possible to rearrange calculations so as to avoid taking the difference of two close numbers. If this cannot be done, it suggests that the problem that forms the subject of the calculation is incapable of accurate solution.

There follows a review of the arithmetic operators that can be used on **real** numbers.

The operators +, −, *, /, <, >, < = and > = all accept **real** and integer operands in any combination. The first operand of ↑ may be **real** or **int**, but the second one must be **int**—it is not permitted to raise numbers to **real** (possibly fractional) powers.

The results of +, − and * are **real** if either of their operands is **real**.

'/' always produces an **int** result. Thus a'/'b gives the number of times b goes into a, ignoring fractions.

/ always generates a **real** result that is as exact as it can be.

Since the operations on **real** numbers do not generally give exact results, it is unsafe to expect two quantities that are mathematically equivalent (such as 1 and 3/3) to have identical representations. The operators = and ≠ are therefore dangerous if used with **real** numbers, and in some implementations they are only defined for integer operands.

The rules are summarised in table 12.2.

Operator	Result produced with operands of given modes			
	int, int	**int, real**	**real, int**	**real, real**
+	**int**	**real**	**real**	**real**
−	**int**	**real**	**real**	**real**
*	**int**	**real**	**real**	**real**
/	**real**	**real**	**real**	**real**
↑	**int**	not defined	**real**	not defined
<	**bool**	**bool**	**bool**	**bool**
>	**bool**	**bool**	**bool**	**bool**
< =	**bool**	**bool**	**bool**	**bool**
> =	**bool**	**bool**	**bool**	**bool**
=	**bool**	unsafe	unsafe	unsafe
≠	**bool**	unsafe	unsafe	unsafe

Table 12.2

Monadic operators that can be used on **real** quantities include

abs x (modulus of x)
sign x (gives an **int**—$+\mathit{1}$ if $x > \mathit{0}$, $\mathit{0}$ if $x = \mathit{0}, -\mathit{1}$ if $x < \mathit{0}$)
entier x (gives an **int**—the whole number part of x)
round x (gives an **int**—the whole number nearest to x)

In many mathematical calculations there is a need for various algebraic functions like logs, square roots or sines and cosines. Algol 68 provides a set of these as standard procedures. Each one has an identifier (like *sqrt* or *sin*) and can be used exactly as if it had been declared as a procedure of mode **proc(real)real.**

The list of standard procedures includes

sqrt (x) the (positive) square root of x (a dynamic fault occurs if $x < \mathit{0}$)
exp (x) the exponential function e^x
ln (x) log x (to the base e) (a dynamic fault occurs if $x < = \mathit{0}$)
sin (x) sine (x)
cos (x) cosine (x) } (x is in *radians*, not degrees)
tan (x) tangent (x)
arcsin (x) $\sin^{-1}$ (x)
arccos (x) $\cos^{-1}$ (x)
arctan (x) $\tan^{-1}$ (x).

In the inverse trigonometrical procedures, the result is delivered in radians. With arcsin and arctan, the result is always in the range $-\pi/2$ to $+\pi/2$, and with arccos it is in the range 0 to π. A dynamic error is signalled if the parameter of arcsin or arccos is outside the limit -1 to $+1$.

The distinction between monadic operators and standard procedures is somewhat blurred. The difference is that operators like **abs** can apply to more than one type of operand, and their meaning depends on the type of operand used, whereas standard procedures always expect a **real** parameter and deliver a **real** result. Of course, it is possible to write, say, *sqrt* (*5*), but here the 5 would be widened to a **real** quantity before the procedure was executed.

It is worth remembering that, in writing programs, procedure identifiers are never underlined (or enclosed in primes), and they always require brackets round their parameters; operators written with letters are always underlined and do not need brackets to enclose their operands, although it is never wrong to use them if you wish.

The ideas discussed in this chapter can be illustrated using a practical problem taken from surveying. Land agents often need maps of flat but irregular areas like fields, and one quick method of making such a map is called a *subtense traverse.*

The process begins by partitioning the border of the area into a number of straight-line sections, making each section small enough to ensure that together they form a sufficiently accurate representation of the actual boundary. This is illustrated in figure 12.1, where the boundary is divided into six sections (shown as dotted lines). On the ground the ends of the sections are marked with pegs.

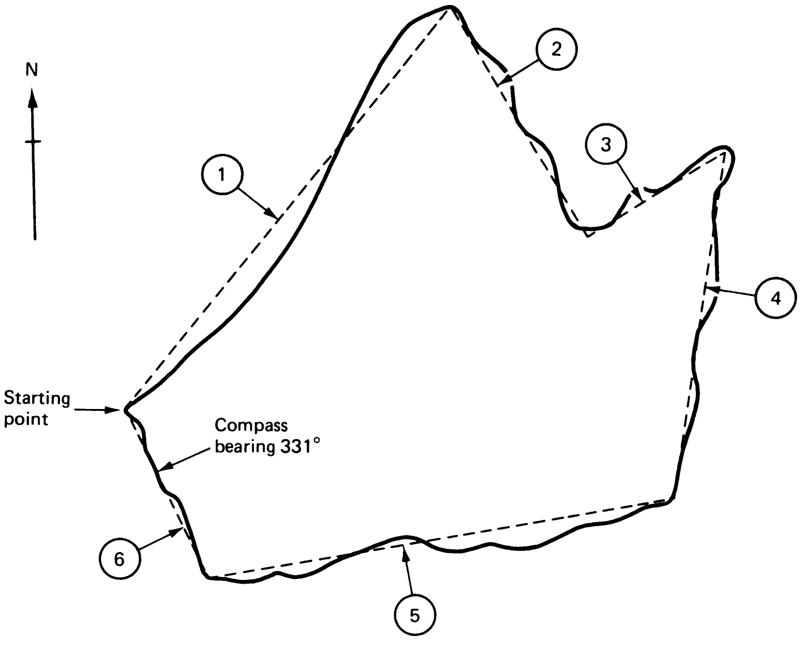


Figure 12.1

The survey goes clockwise, starting from a point at the end of one of the sections. Before it starts, the bearing of the section just preceding the starting point is measured with a magnetic compass or, if the orientation of the area with regard to North is not important, this bearing is simply guessed. Bearings are always expressed in terms of degrees East of North, so that the measured bearing in figure 12.1 is about 331°.

Next, each of the sections is surveyed in order. The measurements are made with a theodolite and a subtense bar.

A theodolite is essentially a telescope fitted with cross-hairs and mounted on a horizontal protractor. It is used to measure angles. In figure 12.2, the theodolite is supposed to be set up at A. First it is pointed at C, and the reading of the protractor is taken. Then it is swung to B (without moving the protractor) and another reading taken. The difference in readings is a measure of the angle CAB.

Modern theodolites read in degrees and decimals of a degree, and the best ones are accurate to 1/1000 of a degree or better. The theodolite used in the present example is assumed to be a cheap instrument that is only calibrated in fifths of a degree. If it is read to the nearest division, the error in any reading can be up to 0.1°.

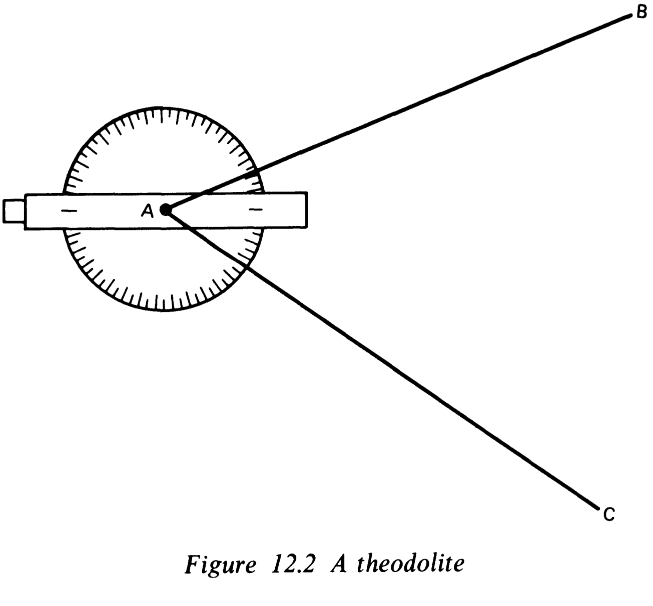


Figure 12.2 A theodolite

A subtense bar is simply a horizontal beam of wood with circular targets mounted at its ends (see figure 12.3). The centres of the targets are exactly two metres apart. The beam is fixed to a tripod and is pivoted about a vertical axis. At the centre of the beam, and at right angles to it, there is a sight like that on a gun.

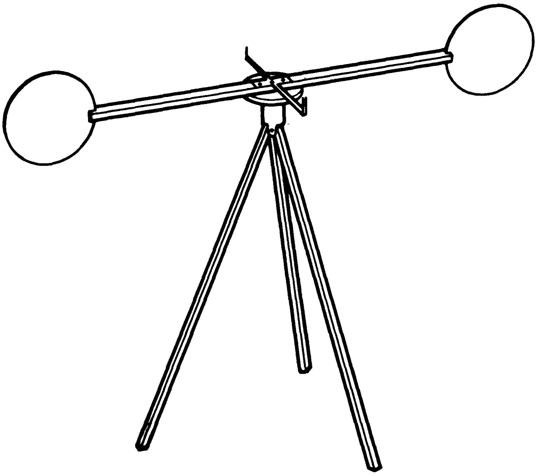


Figure 12.3 A subtense bar

The measurement of a section is shown in figure 12.4. The theodolite is set up at point A at one end of the section, and the subtense bar is placed at the other. The surveyor's assistant swings the bar until he can see the theodolite in the sight, and then clamps it; this ensures that the bar itself is perpendicular to the line of sight between A and B. The surveyor measures two angles, both involving the end of the 'previous' section X. They are recorded in a notebook as the 'inside' and 'outside' angles, respectively. Figure 12.5 shows the relevant page from the notebook when the circuit of the area is complete.

When the measurements are finished, they are taken back to the office where the surveyor (or more likely, his assistant) has the job of 'computing' them so that the map can be drawn.

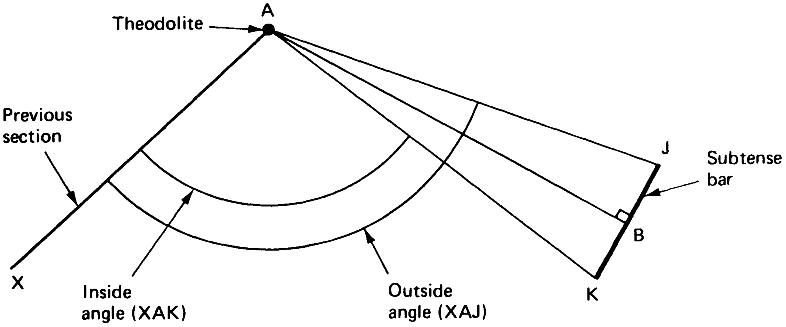
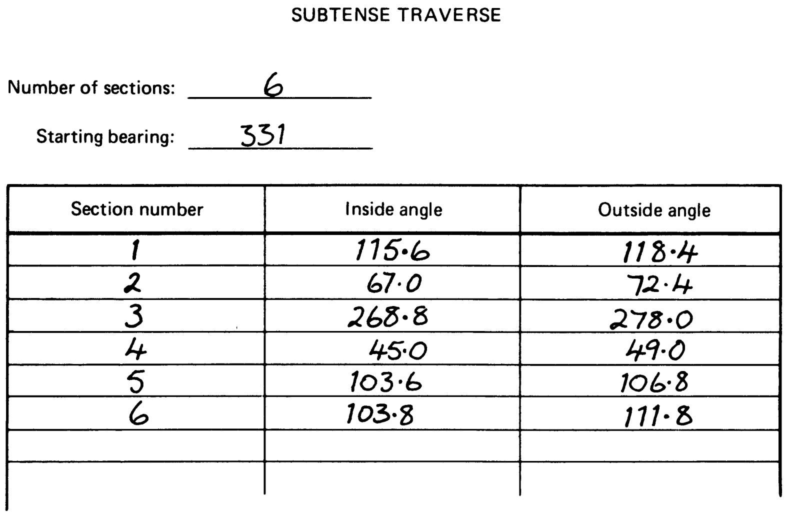


Figure 12.4 Measuring a section



SUBTENSE TRAVERSE

Number of sections: 6

Starting bearing: 331

Section number	Inside angle	Outside angle
1	115·6	118·4
2	67·0	72·4
3	268·8	278·0
4	45·0	49·0
5	103·6	106·8
6	103·8	111·8

Figure 12.5 Specimen notebook page

The calculations are done on a 'work-sheet', as shown in figure 12.6. They involve several stages.

(1) The observed figures are copied in from the field notebook.

(2) The bearing, or direction with respect to North, of each section is determined. Referring back to figure 12.4, the angle XAB is one of the interior angles of the polygon that describes the area. Its value is the mean of the inside and outside angles at A.

If the bearing of XA is known, then the bearing of AB can be found using simple geometry. It turns out to be the bearing of XA, plus 180°, minus the interior angle. If the result is greater than 360, it is reduced by 360.

To give an example, the interior angle at A is $\frac{1}{2}(115.6+118.4)$ or 117.0°. The bearing of the 'previous' section was measured by compass and is 331°, so that the bearing of the first section is $331+180-117\ (-360)$ or 34°. Using this figure, the bearing of the next section can be found, and so on. The bearing of the last section, assumed as 331 at the beginning of the calculation, turns out to be 330.9, but this discrepancy is too small to worry about at the moment.

(3) The length of each section is calculated. Again referring to figure 12.4, the length BJ is always 1 metre, and the angle BAJ is $\frac{1}{2}(\text{XAJ}-\text{XAK})$, or half the difference between the inside and outside angles at A. Again using simple trigonometry, the distance AB is cot (BAJ) or cot $\{\frac{1}{2}(\text{outside}-\text{inside})\}$. Thus the length of the first section is cot $\{\frac{1}{2}(118.4-115.6)\}$ = cot (1.4°).

First bearing: 331

Number of sections: 6

Sec	i	e	Int	b	$\frac{1}{2}$sub	dist	sin(b)	cos(b)	East	North	X	Y	X_c	Y_c
1	115.6	118.4	117.0	34	1.4	40.92	0.5592	0.8290	22.88	33.92	22.88	33.92	22.73	33.68
2	67.0	72.4	69.7	144.3	2.7	21.20	0.5835	−0.8121	12.37	−17.22	35.25	16.70	34.96	16.22
3	268.8	278.0	273.4	50.9	4.6	12.43	0.7760	0.6307	9.65	7.84	44.90	24.54	44.45	23.81
4	45.0	49.0	47.0	183.9	2.0	28.64	−0.0680	−0.9977	−1.95	−28.57	42.95	−4.03	42.36	−5.00
5	103.6	106.8	105.2	258.7	1.6	35.80	−0.9806	−0.1959	−35.11	−7.01	7.85	−11.04	7.10	−12.25
6	103.8	111.8	107.8	330.9	4.0	14.30	−0.4863	0.8738	−6.95	12.50	0.89	1.45	0.00	0.00

Notes: e is the 'outside' angle, and i is the 'inside' angle.
Int is the interior angle of the polygon: $\frac{1}{2}$(i + e).
b is the bearing of the section, expressed as degrees East of North.
$\frac{1}{2}$sub is half the angle subtended by the subtense bar: $\frac{1}{2}$(e − i).
dist is the length of the section: cot($\frac{1}{2}$sub).
East is the 'Easting' of the section in metres, and North is its 'Northing'.
X and Y are the computed coordinates, and X_c and Y_c are the coordinates after correction.

Figure 12.6 Specimen work-sheet

(4) The position of the end of each segment relative to its own start is calculated, expressed as so many metres of 'Easting' and so many metres of 'Northing'. If b is the bearing of a section, and dist its length, then the Easting is given by the expression dist.sin (b) and the Northing by dist.cos (b). The formulas work for all values of b. In certain quadrants either or both of the Easting and Northing will be negative.

(5) Assuming that the starting co-ordinates are (0, 0), the co-ordinates of the end-points of each of the sections are calculated as the cumulative totals of the Northings and Eastings in working round the area. The results are given in the columns marked X and Y.

(6) At this point, for the first time, some kind of check can be made for accuracy. In theory the end-point of the last section should coincide with the assumed starting coordinates (0, 0). In practice, of course, it will never do so because of the inevitable errors in the observations. Two types of error can be distinguished. One type is imposed by the limitations of the instruments used. For example, the theodolite cannot be read with a precision of more than 1/10 of a degree, and over a section of 40 metres an error of 0.1° in the apparent angle of the subtense bar would lead to an error of about $1\frac{1}{2}$ metres in the length of the section. Discrepancies of this order are therefore to be expected.

The other type of error is called a *blunder*, and consists of a gross misreading of the theodolite, or a mistake in writing or copying figures or a mistake in calculation. Some blunders will affect the result so slightly as to be indistinguishable from observational errors, but most will produce large and obvious discrepancies.

In the present example, the actual discrepancy in the final position is 0.89 metres of Easting and 1.45 metres of Northing. This represents a positional error of $\sqrt{}(0.89^2 + 1.45^2)$, or 1.702 metres. This seems to be in line with the expected error, and so it can be accepted as having been caused purely by the imperfections of the instruments.

If the error had been much larger (say, 8 to 10 metres), it would have been necessary to assume the existence of a blunder. In that case, the appropriate procedure is to repeat all the calculations and, if the error refuses to go away, to return to the area and perform the whole survey again.

(7) Even when the final error is deemed acceptable, the calculated coordinates in columns X and Y cannot be used, for a map drawn with them would not close, but would have a small gap requiring the insertion of a purely imaginary extra section. Furthermore, on the assumption that the errors accumulate in working round the area, the coordinates of the first few sections will be relatively accurate, but matters will then become steadily worse, and the last section will be the least accurately placed of all.

These difficulties can be avoided by a process called 'distributing the errors'. It is assumed that each section introduces an equal amount of the final error; hence in this example, each section introduces 0.89/6 too many

> units of Easting and 1.45/6 too many units of Northing. The appropriate amount is then subtracted from each section and the coordinates are recalculated: the result is shown in columns X_c and Y_c. The outline now closes accurately and there is no bias against the later sections of the survey.

As you can imagine, the process of computation is slow, tedious and prone to error. If you have many subtense traverses to compute, you might wish for a friendly computer to read the data as recorded on the field notebook and print out the corrected coordinates of all the points directly. An Algol-68 program for this task is as follows.

```
begin int n; read (n); real b; read (b);
   print ((newline, newline, "SUBTENSE TRAVERSE
      CALCULATION",
      newline, "NUMBER OF SECTIONS = ", n, newline,
      "FIRST BEARING = ", b));
   [1:n]real x, y;
   real xhere: = 0, yhere: = 0;
   real qq = pi/180;
   print ((newline,        "INNER ANG. OUTER ANG.", newline));
   for j to n
      do real p, q; read (p); read (q);
         print ((newline, p, q));
         real dist = 1/tan(0.5 * (p - q) * qq);
         real intang = 0.5 * (p + q);
         b: = b + 180 - intang; (b > 360| b -: = 360);
         xhere +: = dist * sin (b * qq);
         yhere +: = dist * cos (b * qq);
         x[j]: = xhere; y[j]: = yhere
      od;
   print ((newline, "TOTAL ERROR IN EASTING = ", xhere,
          newline, "TOTAL ERROR IN NORTHING = ", yhere,
          newline, "OVERALL ERROR =", sqrt (xhere↑2 + yhere↑2)
          ));
          print ((newline, "CORRECTED COORDINATES ARE"));
   for j to n
      do print ((newline, x[j] - xhere * j/n, y[j] - yhere * j/n))
      od
end.
```

The output of this program is designed to detect as many errors as possible. The raw data are printed out so that they can be checked against the entries in the field notebook, and the final discrepancy is also presented. It is up to the user to decide whether to accept the results or to look for blunders in the original observations—the calculations themselves can be taken as reliable.

Inside the program, the constant *qq* is used to convert angles from degrees to radians wherever necessary. It is set up using the predefined identifier *pi*, which has mode **real** and the value 3.14159265359 . . . The variables *p* and *q* represent the internal and external angles for each observation, and *xhere* and *yhere* keep track of the current (uncorrected) co-ordinates as the calculation moves round the surveyed area. As each new set of co-ordinates is reached, it is recorded in the appropriate elements of the **real** arrays *x* and *y*.

The output of the program for the given example is shown in figure 12.7.

```
SUBTENSE TRAVERSE CALCULATION
NUMBER OF SECTIONS = +6
FIRST BEARING = +3.3100000000&+2
INNER ANG.               OUTER ANG.

1.1560000000&+2          1.1840000000&+2
6.7000000000&+1          7.2400000000&+1
2.6880000000&+2          2.7800000000&+2
4.5000000000&+1          4.9000000000&+1
1.0360000000&+2          1.0680000000&+2
1.0380000000&+2          1.1180000000&+2

TOTAL ERROR IN EASTING = -8.9086696605&-1
TOTAL ERROR IN NORTHING = -1.4510498919&+0
OVER-ALL ERROR = +1.7027007194&+0
CORRECTED COORDINATES ARE
-2.2732248370&+1         -3.3680229943&+1
-3.4957732045&+1         -1.6218198695&+1
-4.4454603875&+1         -2.3814920087&+1
-4.2358422958&+1         +4.9968612674&+0
-7.1033977931&+0         +1.2253683297&+1
+0.0000000000&+0         +0.0000000000&+0
```

Figure 12.7 Computer printout for subtense traverse calculation

This example illustrates a number of points. In the first place, the precision of the machine is much higher than the accuracy of the observations, so that no noticeable new error is introduced from this source. Second, the program includes the subtraction of two nearly equal numbers. Since the measured angles can lie anywhere in the range 0 to 360°, and have a maximum error of 0.1°, their relative accuracy might be taken to be 0.1/360, or about 0.03 per cent. Nevertheless, the error expected in the length of a section is 1.5 metres, or about 4 per cent of its total length. The reason is clear — the angle subtended by the bar is the difference between the inner and outer angles, which are bound to be close. The larger the section being measured, the closer they will be, and the greater the relative error in their difference.

The calculation cannot be blamed for this inaccuracy. To obtain a more precise map, it will be necessary to buy a better theodolite, or to get a larger subtense bar, or to use shorter sections and more of them — a new computer will not help.

The final part of this chapter reviews the whole question of **real** numbers and their precision.

In **real** numbers, the machine provides a basic mode that can handle a wide range of numbers with a fixed relative precision of about 10 to 12 decimal digits. In most practical calculations the limits on accuracy are set by the observed data, and in almost every case the inherent precision of the data is less than the accuracy provided by the **real**-number mechanism on the computer. Provided that operations on **real** numbers are never expected to be *exact*, they can be used for nearly every scientific and engineering calculation.

The conclusion is that the computer, like Lord Copper, is indeed accurate up to a point.

EXERCISES

12.1 (4) A computer uses **real** numbers arranged as follows

$$\underbrace{\text{xxxxxxxxxx}}_{\substack{\text{mantissa}\\ \text{(10 bits)}}} \qquad\qquad \underbrace{\text{xxxxxx}}_{\substack{\text{exponent}\\ \text{(6 bits)}}}$$

The value of any **real** number is taken as

mantissa $*\ 2^{\text{exponent}}$

Show the best representation for the following numbers

77 10 1 0.25 1000 1001 −128 0.1

(Hint: Remember that both the mantissa and the exponent are signed binary numbers (2s-complement notation) and that in every case the mantissa should be as large as possible.)

12.2 (2) Using a calculator or slide rule, trace the square-root program on p. 167 for all the data given.

***12.3** (2) The hyperbolic functions are defined as follows

$$\sinh(x) = \tfrac{1}{2}(e^x - e^{-x});\ \cosh(x) = \tfrac{1}{2}(e^x + e^{-x});\ \tanh(x) = \sinh(x)/\cosh(x)$$

Write a program that tabulates these functions for values of x between 0 and 10, going up in steps of 0.1.

12.4 (5) Write a program that reads in three coefficients a, b and c, and solves the quadratic equation $ax^2 + bx + c = 0$. (Strong hint: Remember all the special cases!)

***12.5** (5) (P) (a) Write a procedure with the following heading

```
proc squarewave = (real x, int n) real;
  comment This procedure calculates and sums n terms of the series
          sin(x) + sin(3x)/3 + sin(5x)/5 + ...
  comment
```

(b) Study the following program, which prints out graphs of sin (x) and cos (x)

```
([0:100] char line;
  for j from 0 by 10 to 720
    do real x = j * pi/180;
      int s = entier (50 * (sin (x) + 1)),
          c = entier (50 * (cos (x) + 1));
      for k from 0 to 100 do line [k]: = " " od;
      line [s]: = "S"; line [c]: = "C";
      print ((newline, j, line))
    od
)
```

(c) Using the general method of this program, together with the procedure you wrote for (a), construct a program to print out graphs of the family of functions

$$y = \sum_{n=0}^{q} \frac{\sin[(2n+1)x]}{(2n+1)}$$

for q = 0(1)5. Vary x between 0 and 720° in steps of 10°.

13 *Some Useful Constructions*

'A wandering minstrel I, a thing of shreds and patches.'
W. S. Gilbert, 'The Mikado'

Most languages offer several ways of saying anything that you may want to express. Algol 68 is no exception: it has a number of constructions that, although not essential to programming, enrich the language and can be used to make programs more elegant and compact.

The 'extra' features are not connected to one another in any structured way; they are more of a heap than an ordered collection. The order of presentation here has no significance.

First, consider the values of statements. You are already familiar with the idea that some units yield values when they are executed. Good examples are the boolean-expression in a **while**–**do**–**od** construction or the integer-expression that delivers the result in a procedure of mode **proc int** or **proc**(. . .)**int**. It turns out that *all* units generate values. In most cases, there is an obvious value to be produced; in an assignment statement, for example, the value is the same as the object that was assigned. Thus the value of *p*: = *17* is 17. In a few instances, such as the *read* or *print* commands, calls to procedures of mode **proc void** or **skip**, no actual value is yielded. These unitary clauses are said to yield the special value **void**, which means 'nothing'.

Usually it is only the *last* value of a serial-clause that is of interest. The value of every other unit is thrown away as soon as the program passes a semicolon. Consider the following phrases:

int *x*, *y*, *z*; *x*: = *2*; *y*: = *x* + *7*; *z*: = *y* ÷ *x*

Declarations do not yield values. The first unit clause to be evaluated is *x*: = *2*. It sets the variable *x* to *2*, and also yields the value 2, but this value is discarded as the machine passes the semicolon.

The next unit is *y*: = *x* + 7, which sets *y* to *9* and yields the value *9*, only to be thrown away again as the computer passes the following semicolon. The final value, which is not discarded, is 9 ÷ 2 or 4.

The general rule is that the value produced by a unit is always thrown away if the unit is followed by a semicolon.

If an assignation is enclosed in brackets, it can be included in an expression like any other operand. For instance

```
read(a[p +: = 1])
```

is equivalent to

```
p+: = 1; read(a[p])
```

while

```
x: = (y: = 17)+3
```

is equivalent to

```
y: = 17; x: = y+3
```

and

```
a: = (b: = (c: = 0))
```

has the same effect as

```
c: = 0; b: = c; a: = b
```

In the last case, the brackets can be removed without ambiguity, giving forms like *a*: =*b*: = *c*: =*0*.

If the variables on the left of such a group of assignations are of different modes (say, **ref int** and **ref real**), then a coercion will take place between the various assignations, and care must be taken to write the variables in the right order. An **int** value can be widened to **real**, but the result cannot be 'narrowed' back to **int**. Study the following examples, remembering the order of assignation

```
int x, y; real q;        int x, y; real q;
q: = x: = y: = 99       x: = q: = y: = 99

    (legal)                  (illegal)
```

The combined operators + : =, − : =, ∗ : =, ÷ : = and /: = also yield values that can be used in expressions. However you must remember that the priority of all the combined operators is 1 (not 6 or 7), and if they are used as parts of expressions brackets will often be needed. Compare

```
y: = x+ : = 3+2
```

(which is similar to *x*: = *x*+*3*+*2*; *y*: = *x*) and

```
y: = (x+: = 3)+2
```

(which is equivalent to *x*: = *x* + *3*; *y*: = *x* + *2*).

The combined operators are fussy about the modes of their operands. The left-hand operand must always be a **ref int** or a **ref real**, otherwise the assignation could not take place. It must be a **ref real** if the right-hand operand is of mode **real** or **ref real**, and the left-hand operand of /: = must be of mode **ref real** in all cases. These rules are of course similar to those that apply to the left-hand sides of assignations

The main advantage of a combined operator is that it prevents the address of a variable from being calculated twice. This is really only significant if the variable in question is an array element with a complicated suffix expression. Suppose that two procedures row and col, both of mode **proc(int, int)int**, have been defined. It would then be good sense to replace

```
a[row(p, q), col(p, q)]: = a[row(p, q), col(p, q)] + 1
```

by

```
a[row(p, q), col(p, q)] +: = 1
```

since — whatever the procedures may do — each is called once only, instead of being called twice with identical parameters.

A feature of Algol 68 that is useful in the area of decision-making is the **case** construction. This is written as follows

```
case integer-serial-clause
  in unit (1), unit (2), unit (3)
     unit (4), . . .
     . . ., unit (n – 1), unit (n)
  out serial-clause (d)
esac
```

where **case**, **in**, **out** and **esac** are system words.

In obeying this construction the computer first works out the integer-unit between **case** and **in**, and gets an integer value. It then uses this value to select *one* of the units that follow: the first for 1, the second for 2 and so on up to the nth for n (where n is the number of units between **in** and **out**). If the value of the integer-unit is outwith the bounds 1 to n, the computer chooses the default serial-clause between **out** and **esac**.

The construction is equivalent to a long chain of **if** commands, but considerably more compact and efficient. Thus

```
case x in
      (print("PASS"); a: = 5), x + : = 1, (print("FAIL"); a: = 0)
      out print ("X NOT 1 2 OR 3")
esac
```

(where the brackets ensure that their contents are taken as units) could also be set out as

```
if x = 1 then print("PASS"); a: = 5
  elif x = 2 then x + : = 1
  elif x = 3 then print("FAIL"); a: = 0
  else print("X NOT 1 2 OR 3")
fi
```

In terms of the railway notation introduced in chapter 4, the construction appears as in figure 13 .1.

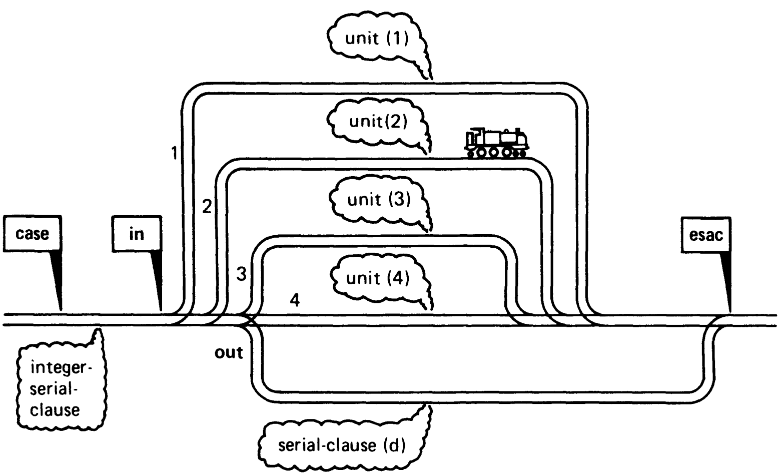


Figure 13.1

Permitted alternatives for the symbols **case**, **in**, **out** and **esac** are (, |, | and). Although these signs are never ambiguous in a correct program, the use of the contracted forms is a matter of personal taste—you may find them unreadable.

Sometimes you can be sure that your integer-unit will never be outside the expected bounds, and you can then omit **out** and the default option. If—contrary to expectation—the value of the integer unit is out of range none of the units is obeyed and the program passes on to the statement after the closing **esac**.

In the following simple example, the program reads a decimal number supposedly in the range 1 to 99 and prints its name in English.

```
begin int n; read (n);
   if n < 20
      then print(case n
         in "ONE", "TWO", "THREE", "FOUR", "FIVE",
            "SIX", "SEVEN", "EIGHT", "NINE", "TEN",
            "ELEVEN", "TWELVE", "THIRTEEN",
              "FOURTEEN", "FIFTEEN",
            "SIXTEEN", "SEVENTEEN", "EIGHTEEN",
              "NINETEEN"
         out "NUMBER LESS THAN 1"
                 esac
                  )
         else print(case n ÷ 10 − 1
       in "TWENTY", "THIRTY", "FORTY", "FIFTY",
            "SIXTY",
              "SEVENTY", "EIGHTY", "NINETY"
          out "NUMBER GREATER THAN 99"
              esac
              );
          if n > 99 then n: = 0 fi; co look after improper value of n co
          print(case   n mod 10 + 1
            in " ", "ONE", "TWO", "THREE", "FOUR",
               "FIVE", "SIX", "SEVEN", "EIGHT",
                 "NINE"
                         esac
                         )
    fi
end
```

The case construction is useful whenever the possibilities to be considered can be mapped on to (or *transformed* to) the numbers 1 2 3 . . . This can usually be done, even when the decisions to be taken seem arbitrary and complicated.

The last two topics in this chapter are slicing and trimming. These two operations can be used on arrays with elements of any mode, to select subsets of them in various ways.

Slicing applies to arrays of two or more dimensions. It can be made to choose any group of elements that have one (or more) of their index values in common. Thus a slice of a two-dimensional array can either be a row where all the elements have the same *first* index, or a column in which all the elements have their *second* index in common.

A slice of an array is written down in exactly the same way as a single element, except that the index that varies along the slice is left out. Consider the array declared by

```
[1:3, 1:4] int ax
```

which is assumed to have the value

```
 3   7   4  12
18   2  13   4
21  13   7  10
```

One possible slice from this array is the third column

```
 4
13
 7
```

The index that varies along this slice is the row index (since each element comes from a different row) and the slice is therefore written as

```
ax[ ,3]
```

(Note the gap before the comma; it is not essential, but helps to make the text more readable.)

Similarly, the row

```
3  7  4  12
```

would be selected as *ax*[1,].

The slice of an array is a full array in its own right. It can be given as a parameter to a procedure or assigned to another array variable of appropriate mode and bounds. It will always have one or more dimensions fewer than the array from which it was sliced, but in those dimensions that remain the bounds of the indices will be the same as in the parent array. For example, any row chosen from *ax* will always have four elements, and any column of *ax* will always have three.

Although the above example used only a two-dimensional array, the extension of the rules to arrays of three or more dimensions is quite straightforward. Thus there are six possible slices in which an element of a three-dimensional array could be incorporated. Three of them are mutually perpendicular planes of elements, and the other three are mutually perpendicular lines.

One trivial but useful application of slicing is in printing the rows of a two-dimensional character array. In chapter 9, for example, the map of Great Britain was printed by the somewhat cumbersome sequence

```
for v from 100 by -1 to 1 do
            print(newline);
            for w to 90 do print(map[v, w]) od
      od;
```

Since the *print* command can handle a one-dimensional array of characters correctly, the use of slicing allows a more compact form to be written, as follows

```
from j to 100 do print((newline, map[101 -j,])) od
```

To *trim* an array is to cut off one or both of its ends (in any dimension, if there is more than one). Consider the character array declared by

```
[1:12] char x: = "CANTANKEROUS"
```

Some trimmed versions of this array are "CAN", "ANT", "TANK", "US". The subset actually required is indicated by giving the subscripts of the first and last elements involved. Thus,

```
x[1:3]   = "CAN"
x[2:4]   = "ANT"
x[4:7]   = "TANK"
x[11:12] = "US"
```

Again, each subset of the original array is an array in its own right. The elements are numbered from 1 upwards; for instance, *x*[*2*:*4*] [*1*] = "*A*". This is important when a trimmed array is assigned to an array variable, because the subset and the array variable must agree in mode and number of dimensions, and must also have identical bounds. Consider the following assignations in the context of the character array *x* defined above.

(1)	(2)	(3)
[1:4] **char** *q*;	[*1*:*2*] **char** *r*;	[*0*:*3*] **char** *s*;
q: = *x*[*4*:*7*]	*r*: = *x*[*4*:*7*]	*s*: = *x*[*4*:*7*]

Assignation (1) is legal, since *x*[*4*: *7*] has four elements numbered 1 to 4, and so has *q*. Assignation (2) is illegal, since it attempts to assign an array with four elements to a variable with 12 elements. Assignation (3) is also illegal. Although s has four elements, they are numbered 0 to 3, and so there is still a mismatch between the bounds of the two arrays. However, in this case the bounds could be matched by using the special operator @ (sometimes written **at**), which shifts the bounds of any array bodily, by specifying a new lower bound. For instance, *x*[*5*: *10*] has six elements, and the suffix of the first is 1, whereas *x*[*5* : *10*]@ 3 also has six elements but the suffix of the first is now 3 (and that of the last is 8). Hence, the assignation [*0*:*3*] **char** *s*; *s*: = *x*[*4*: *7*]@ *0* would be legal.

The lower bound specified by @ can, as usual, be a unit of any complexity.

Trimmed arrays can be used on the left-hand sides of assignations. In this context remember that, in arrays specified by row-displays or string literals, the first subscript is always 1. This feature ties in well with the rule that the first element in a trimmed array also has the subscript 1. For instance, it would be legal to write

x[*5*:*10*]: = "*ERBURY*"; *print*(*x*[*1*:*10*])

The result would be CANTERBURY.

Trimming is useful in a number of ways. As a simple example, consider a character array *n* that contains a person's name and address. The arrangement is such that the country of residence is always in positions 60 to 72. If it is required to print the actual country (and nothing else), this can be achieved by writing

print(*n*[*60*: *72*])

The following discussion gives a much more complex illustration.

One of the most important jobs in computing consists of sorting items — numbers or character arrays — into order. Sorting is a key part of the process of using and maintaining sets of information such as stock records, bank statements or names and addresses. Sorting is so universally needed, and so time consuming, that many computers devote the major part of their time to it. The use of efficient sorting methods is therefore of great economic importance.

Many different methods of sorting have been invented. Knuth (1973) describes and analyses 59 of them, but in the present chapter only two will be discussed.

Consider an array of numbers in completely random order. Perhaps some numbers occur more than once

37
19
8
22
19
46
4
12
41

A very simple method of sorting consists of the following steps.

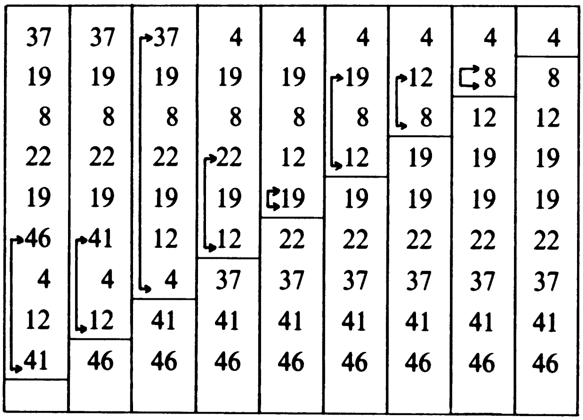
To sort an array with n elements

(1) find the largest element in the array, and interchange it with the element at the bottom of the array; the largest element must now be in its correct position, and can be excluded from further consideration

(2) repeat this process for the first n – 1 elements of the array, omitting any reference to the last element (which is already in the correct place); this will place the second largest value in the second last position, so that it too can now be left alone

(3) –(n – 1) repeat the process for subarrays of n – 2, n – 3, . . ., 2 elements; at this point the array will be correctly ordered.

The sample array contains 9 elements, and so (9 – 1) or 8 of these steps will be necessary to sort it.



37	37	37	4	4	4	4	4	4
19	19	19	19	19	19	12	8	8
8	8	8	8	8	8	8	12	12
22	22	22	22	12	12	19	19	19
19	19	19	19	19	19	19	19	19
46	41	12	12	22	22	22	22	22
4	4	4	37	37	37	37	37	37
12	12	41	41	41	41	41	41	41
41	46	46	46	46	46	46	46	46

(0) (1) (2) (3) (4) (5) (6) (7) (8) (number of steps)

The diagram shows the successive stages of the process. Initially, the largest element of all is 46, and it is interchanged with the 41 at the bottom of the array. Next the 41, which is the largest remaining element in the unsorted part of the array, is interchanged with the 12,which is the element at the bottom of this part. At the third stage, the 37 is swapped with the 4, and so on. The diagram shows how the sorted array is built up from the largest element upwards and how the size of the unsorted array is decreased by one at every stage. In two cases the largest element happens by chance to be in the correct position, and so it is simply swapped with itself.

To program the method, consider first the inner loop, which corresponds to one step of the sort process. The loop consists of two portions

(1) find the location of the largest element in the part of the array that has not yet been sorted
(2) interchange it with the last element of the unsorted array.

The search is performed by scanning the array, element by element. The variables *maxval* and *pos* will be used to keep track of the value and position of the largest element found so far. Initially, *maxval* is set to the first element of the array; this is better than setting it to 0 because the algorithm will still work even if all the elements to be sorted are negative.

Assume that the array to be sorted is called *a*; that its lower bound is 1; and that variable *j* (declared globally) gives the number of elements in the unsorted section. Then the inner loop is

```
int maxval: = a[1], pos: = 1;
for k from 2 to j
   do if a[k] > maxval then maxval: = a[k]; pos: = k fi od;
a[pos]: = a[j]; a[j]: = maxval
```

The entire sort algorithm simply consists of this inner loop, together with a control sequence that executes it for every value of j from the full size of the array down to 2. It can be set down as a procedure, taking a parameter of mode **ref[ ]int**

```
proc sort = (ref[ ]int a) void:
co This procedure sorts the elements of a into ascending order.
   The lower bound of a should be 1
co
(if lwb a ≠ 1
  then print((newline, "LOWER BOUND INCORRECT"))
  else for j from upb a by −1 to 2
    do int maxval: = a[1], pos: = 1;
       for k from 2 to j
           do if a[k] > maxval then maxval: = a[k]; pos: = k fi od;
       a[pos]: = a[j]; a[j]: = maxval
    od
```

This sorting method is one of the simplest in existence, but it is not widely used by professional programmers; it turns out to be far too slow.

To analyse the method, look first at the inner part of the algorithm. Finding the largest of a set of j elements requires j – 1 comparisons; ($a[k] > maxval$). The code that initialises the variables *maxval* and *pos* is only executed once, and the same is true of the instruction at the end of the inner part, which makes the interchange. As for the group of instructions obeyed after a successful comparison ($maxval := a[k], pos := k$), it is impossible to say how many times they are executed since this depends on the data. However, unless by some fluke the data are already in order, the number of times is usually much less than the number of comparisons. In the given example, the numbers of comparisons and of successful comparisons at each stage are as shown in table 13.1.

Stage number	Total comparisons	Successful comparisons
1	8	1
2	7	1
3	6	0
4	5	2
5	4	1
6	3	1
7	2	1
8	1	1

Table 13.1

The total number of operations in the inner part will be at least $j + 3$, but is unlikely to be very much more. When j is large, a rough but useful approximation is to take the total number of operations simply as j.

Now consider the whole algorithm. If the array to be sorted has n elements, the inner part will be executed for all values of j between n and *2*. The total number of operations for the complete sort will be roughly

$$n+(n-1)+(n-2)+\ \ldots +2$$

This is an arithmetic progression, and its sum is $\frac{1}{2}n^2+\frac{1}{2}n-1$. When n is large, the $\frac{1}{2}n^2$ term outweighs the others, and the number of operations needed to sort an array of n elements by this method will be not less than $\frac{1}{2}n^2$. This number increases rapidly with n, as shown in table 13.2.

To give a practical illustration of this effect, suppose that you had tested and timed the procedure on an array of 100 elements, and found that it took a second to sort them. You might be dismayed to discover that to sort 10 000 items (by no means a large number in many applications) required almost three hours of computer time.

Most practical sorting methods are much more efficient, but they pay for it by their complexity. One algorithm that combines relative simplicity with good efficiency is based on the idea of merging.

n	$\frac{1}{2}n^2$
10	50
100	5 000
1 000	500 000
10 000	50 000 000

Table 13.2

Consider two arrays, not necessarily of the same size, each containing a sorted set of elements. The operation of merging consists of combining all the elements present into one large sorted array. For example, the result of merging

(5 17 36 107)

and

(7 12 35 36 72)

is

(5 7 12 17 35 36 36 72 107)

The process can be expressed in Algol 68 by defining a procedure called *merge*. The procedure takes two parameters, each of which is an array of integers, and produces another integer array that is the result of merging the first two. It assumes that the lower bounds of the arrays are 1, and that the elements in both the operand arrays are already sorted. The size of the resulting array must clearly be the sum of the sizes of the operand arrays.

```
proc merge = (ref[ ]int a, b) [ ] int:
(int 1a = upb a, 1b = upb b;
int pa: = 1, pb: = 1; [1: 1a + 1b] int c;
for pc to 1a + 1b
   do(pa > 1a | c[pc]: = b[pb]; pb +: = 1
     |: pb > 1b | c[pc]: = a[pa]; pa +: = 1
     |: a[pa] > b[pb] | c[pc]: = b[pb]; pb +: = 1
     )             | c[pc]: = a[pa]; pa +: = 1
   od;
   c
 )
```

If you have any difficulty in understanding this merging algorithm, get two unequal packs of playing cards (say, of 7 and 10 cards, respectively), and sort each pack into order. Merge the two packs by taking one card at a time off the appropriate pack until both are exhausted. Then read on.

In the above procedure *c* is a local array used to assemble the merged elements. It is declared to be of the correct size, and indexed by *pc*, which moves up the array by one element at a time. At the end of the algorithm the value of *c* is taken as the result of the procedure.

The pointer *pa* indicates the next element to be used from array *a*, and *pb* the next from *b*. Each time an element is taken from one of these arrays, the appropriate pointer is advanced by 1.

The rules for selecting the next element to be entered into *c* are as follows. If array *a* is exhausted, take the next element from *b*; otherwise, if array *b* is exhausted, take the next element from *a*; otherwise, compare the next element from *a* and *b*. If the element from *b* is smaller, take it; otherwise, take the next element from *a*.

Once the merging procedure has been established, the sort algorithm has a neat recursive form.

To sort an array of *n* items
(1) if $n = 1$, do nothing
(2) otherwise, divide the array into two halves that are as equal as possible; sort each one independently and merge the results.

In Algol 68 this becomes

```
proc rmsort = (ref[ ]int a) void:
( if lwb a ≠ 1
    then print((newline, "ERROR IN LOWER BOUND"))
    else int n = upb a;
      if n ≠ 1
        then rmsort (a[1:n÷2]);
          rmsort (a[n÷2+1:n]);
            a: = merge (a[1:n÷2], a[n÷2+1:n])
      fi
  fi
)
```

n	$5n[\log_2 n]$	$\frac{1}{2}n^2$
10	200	50
100	3 500	5 000
1 000	50 000	500 000
10 000	700 000	50 000 000

Table 13.3

The analysis of this algorithm is not difficult, and shows that the number of steps needed to sort *n* items is about $5n[\log_2 n]$, where $[\log_2 n]$ is the smallest integer equal to or greater than $\log_2 n$. It is now possible to calculate figures that compare directly with the simple sort on p. 192, as shown in table 13.3.

To return to the practical example discussed earlier, the programmer who used a merge sort would have found that his test problem with 100 items ran in 0.7 seconds (instead of 1), while his production run with 10 000 items was finished in about 3 minutes instead of $2\frac{3}{4}$ hours .

EXERCISES

***13.1** (1) Given the following declarations

int *a*: = *4*, *b*: = *7*; **real** *x*: = *3.5*

what would be printed in each of the following cases?

(a) *a*+: = *4*; *print*(*a*)
(b) *b* ∗: = *a*+*3*; *print*(*b*)
(c) *x*: = *a*: = *b*: = *7*; *print*((*x*, *a*, *b*))
(d) *print*(((*a*: = *3*)>*6*|"*YES*"|"*NO*"))

13.2 (4) Write a procedure called *minor* with parameters ([,]**real** *a*, **int** *x*, *y*) and result [,]**real**, where *a* is always a square array with the lower bounds of both dimensions set to one. The procedure builds a new array one place smaller in each dimension, by removing row *x* and column *y* from the original. (Hint: Use trimming. No **for** statements are necessary.)

***13.3** (3) Write a procedure called *censor*, with parameter **ref** [ ] **char** and result **void**. The procedure is to scan the array of characters and replace

CONSERVATIVE	by	************
LABOUR	by	******
LIBERAL	by	*******
NATIONALIST	by	***********.

(Hint: Think about your end conditions.)

***13.4** (6) (P) Write a program that reads a set of (British) car numbers supplied as a data stream and sorts them into their years of manufacture. Each car number is up to seven characters long, and the rules for ordering are as follows

first, numbers that start with a letter and end with a digit
then, numbers that start with a digit
then, numbers that start with a letter and end with A
then, numbers that start with a letter and end with B

and so on. The final order of numbers within any year is not significant.

In the data, the car numbers are supplied in groups of 10 to each line. Each number is separated from the next by one or more spaces. The last number is followed by the dummy ZZZ999Z. There are fewer than 3000 numbers. Your answers should appear in the same general format. (Hint: There are enough numbers in the data to make the use of an efficient sorting method essential. Structure your program as follows

(1) a procedure that reads the data in to a suitable array
(2) a procedure that compares any two car numbers and determines their correct order
(3) a procedure to sort the list of numbers
(4) a procedure to print the results.

For (1) and (2) study and use the procedures given below.

```
proc alpha = (char a) bool:
co Determines whether a is a letter co
(a> = "A" and a< = "Z");

proc digit = (char d) bool:
co Determines whether d is a decimal digit co
(d> = "0" and d< = "9");

proc numin = [ ]char;
co Reads the next car number in the data and delivers it as a seven-
   character array. The characters are left-justified and padded out with
   spaces
co
begin [1:7] char x; int p: = 1; char z;
   for j to 7 do x[j]: = " " od;
   while read(z); not (alpha(z) or digit(z))do skip od;
   x[1]: = z;
   while read(z); alpha(z) or digit(z) do x[p+: = 1]÷= z od;
   x
end;

proc cardate = ([ ]char d) int:
co Entered with a car number in d, this procedure calculates a 'date' as
   follows
   for car numbers that end with a digit, 0
   for car numbers that start with a digit, 1
   otherwise, 2 for numbers ending with A, 3 for those ending with B,
   and so on
co
begin int pd: = 1;
   while pd ≠ 7 and d[pd] ≠ " " do pd+: = 1 od;
   if d[pd] = " " then pd-: = 1 fi;
   if digit(d[pd]) then 0
```

```
        elif digit (d[1]) then 1
        else abs d[pd] - abs "A" + 2
    fi
  end;

  proc compare = ([ ] char a, b) bool:
  co Compares the dates of two car numbers a and b. Gives true if a was
     registered in an earlier year than b; otherwise, false
  co
  (cardate(a) < cardate(b));
```

Your tutor will ensure that these procedures are available in machine-readable form.)

14 *Records and Files*

> 'The single letter surname O, of which 13 examples appear in the telephone directory in Brussels, besides being the commonest single letter name is the one obviously causing most distress to those concerned with the prevention of cruelty to computers. There exists among the 42,500,000 names on the Ministry of Social Security index four examples of a one-lettered surname. Their identity has not been disclosed, but they are "E", "J", "M", and "X". Two-letter British surnames include By and On.'
>
> *The Guinness Book of Records*

From the beginning, mathematical and scientific programmers have tended to organise their data into arrays — lists, tables and matrices. On the other hand programmers concerned with data processing have normally used an organisation that depends on records and files.

These two styles of organisation are essentially different, and they complement each other. Both are necessary if a wide range of problems is to be covered.

A *record* is a group of values that, when taken together, form the description of an entity such as an object or person. A library catalogue, for example, would have one record for each book. The record might contain the title, the author's name, the publisher's name, the book's classification, its date of purchase, its cost and a marker to say whether it could be borrowed by children under 18. Each of these quantities is called a *field* within the record. The complete set of records (in this case, the catalogue) is known as a *file*.

Algol 68 provides a record facility. A record may include any number of fields, and each field may be of any mode, including arrays. Records are declared by using the system word **struct**. The following record declaration might occur in a program designed to follow the progress of your local football team

struct ([*1*:*12*] **char** *opponents*, **int** *for*, *against*, **bool** *home*) *game*

This declares a variable called *game*, which is used to store the particulars of any one match. Like an array, it is a compound object consisting of several components; but the components are not all of the same mode, and they are identified by tags instead of subscripts.

The fields of the variable *game* are as follows

(a) a 12-character array to store the name of the opposing team
(b) an integer to show the number of goals scored by your team

(c) another integer to show the number of goals scored by the other side
(d) a boolean that is **true** if the game was played at home, and **false** if it was played away.

The tags *opponents*, *for*, *against* and *home* are called field-selectors. They are chosen arbitrarily in the same way as other identifiers (there is no clash between *for* and **for**).

Once a record has been declared, it can be used in various ways.

(1) Its value can be set up with a structure-display that contains the values of the fields in the right order

game: = ("*RAMSGATE F. C*", *10*, *3*, **false**)

The declaration and initialisation can be combined

struct ([*1* : *12*] **char** *opponents*, **int** *for*, *against*, **bool** *home*) *game*
: = ("*TINTAGEL UTD*", *0*, *0*, **true**)

Note that the field that specifies the opponents is exactly 12 characters long in both cases. The field in the record has been declared to be of this length, and in practice most opposing team names would have to be abbreviated or padded out with spaces so as to conform.

(2) The value of a record can be read from a data card

read (*game*)

The machine expects data that will match the pattern of the record exactly. Thus the first 12 characters (whatever they are) will be taken as the opponents' name. The machine will then look for two integers separated by a space to supply the number of goals for and against, and finally for a truth value (T or F) to give a value for the field home.

(3) It is possible to output the value of the record by using the *print* command. The value of each field will appear in its normal fashion. For example

print(*game*)

might give

```
ALL ENGLAND   +0   +23   T
```

as the output

(4) If two record variables have precisely the same declaration (same fields, same field selectors), then the value of one record can be assigned to the other. This is illustrated in the program to find the best game considered below.

(5) Any individual field can be extracted from the record variable by using a field-selector and the qualifier **of**. Thus

for **of** *game*

refers to the score made by our side, and

opponents **of** *game*

is the name of the other team.
These fields can be used in exactly the same way as variables. They can enter into expressions

bool *win*; *win*: = *for* **of** *game* > *against* **of** *game*

or values can be assigned to individual fields

opponents **of** *game*: = "*LUSS ROVERS* "

When a field is an array, its elements may be selected by using ordinary subscript brackets. However, brackets (of any sort) have a higher priority than **of** and so, to preserve correct grammar, round brackets must be placed around the group formed by the selector name, **of** and the record name. Thus

(*opponents* **of** *game*) [*2*]

is legal (the second character of the opponents' name), but

opponents **of** *game* [*2*]

is illegal, since *game* is not an array.

The program that follows is designed to read the descriptions of a number of games and to pick out and print the 'best' one. This is defined as the game with the largest (favourable) difference between the scores of the two sides. In the case of a tie between two or more games, those played away are given preference, as are those played later in the season. The data are presented as an integer *n* followed by *n* game records, which are assumed to be in chronological order.

```
begin
struct([1:12]char opponents int for, against, bool home)game,
   bestgame;
int n, gs, bs;
read(n);
read((newline, bestgame));
bs: = for of bestgame − against of bestgame;
for j from 2 to n
   do read((newline, game));
      gs: = for of game − against of game;
      if gs > bs or (gs = bs and not home of game)
         then bestgame: = game; bs: = gs
      fi
   od;
print((newline,"THE BEST GAME OF THE SEASON WAS",bestgame))
end.
```

```
6
BALLOCH F.C.       0  5  T
BALMAHA UTD        3  3  F
ROWARDENNAN        4  3  F        (Data)
LUSS ROVERS        7  6  T
INCHAILLOCH       10  5  T
ROSS PRIORY        4  0  F
```

In this program the variable *game* is used to record the current game, and *gs* stores its winning margin. Similarly, *bestgame* remembers the best game so far, and *bs* is used for the best winning margin to date.

In lines 5 and 8, the *read* command is in the form

read ((*newline*, *bestgame*))

This form is necessary whenever a character array starts on a new card, unless it is the first card in the data. When reading numbers, truth values or even single characters, the system will automatically take a new card if one is needed but for character arrays it must be told to do so in specific terms. The first field in the record definition is, of course, a character array.

Before moving on, it is appropriate to mention a few formal properties of records.

The *mode* of a record variable includes the tags of the field-selectors. Thus the full mode of game is

(**ref struct**([ ] **char** *opponents*, **int** *for*, **int** *against*, **bool** *home*)

and the extended form of the declaration (which is so clumsy that it would never be used in practice) is

ref struct([ ] **char** *opponents*, **int** *for*, **int** *against*, **bool** *home*)*game*
= **loc struct**([*1* : *12*] **char** *opponents*, **int** *for*, **int** *against*, **bool** *home*)

(It is interesting to compare this with the full form of an array declaration — for example, **ref** [,] **int** *q* = **loc**[*2*:*5*, *1*:*7*] **int**. Remember that, when modes refer to arrays, they *never* include the actual bounds!)

The selected field of a record variable is called a *secondary*, and its mode (before coercions such as dereferencing) always starts with **ref.** Table 14.1 gives the modes of the fields in game.

Field	Mode
opponents **of** *game*	**ref** [ ] **char**
for **of** *game*	**ref int**
against **of** *game*	**ref int**
home **of** *game*	**ref bool**

Table 14.1

It is also possible to declare a record constant, such as

```
struct (int q, [ ] char ps) wp = (6, "PLAIN")
```

In this case, *q* **of** *wp* would have mode **int** and *ps* **of** *wp* would have mode [ ] **char**. In practice, record constants are not used as much as record variables.

In many problems the data are much better represented by records of a suitable shape and size than simply by unstructured collections of **ints**, **reals**, **bools**, and **chars**. For example, a program that was concerned with making technical drawings would find it convenient to regard a point (specified by its x and y coordinates) as a basic data item.

Algol 68 allows the programmer to declare new modes, and so to create new types of data item. A mode-declaration (which is subject to the normal rules of reach) is written

```
mode modename = new mode
```

for example

```
mode point = struct (real x, y)
```

Here **mode** is a system word. The new mode name can be any underlined word (or word in bold type) that is not already in use. The new mode itself is usually (but not necessarily) a record definition starting with **struct**.

Once a mode has been declared, and provided it is within reach, it can be used to declare new variables and for all other purposes, just as if it were a basic mode like **int** or **real**. For example, the word **point** now stands for **struct** (**real** x, y), and it is possible to write

```
point origin = (0, 0)
point marker: = (5.7, 812)
```

Another useful aspect of this facility is that new modes may be used to specify the parameters, operands and results of procedures. Thus it would be possible to declare a procedure like

```
proc midpoint = (point j, k) point:
  comment Finds the midpoint on the line joining j and j and k comment
  (  (0.5 * (x of j + x of k), 0.5 * (y of j + y of k)  )  );
```

Here the procedure takes two **points** as parameters, and produces the **point** that represents the position half-way in between. For example, the instructions

```
point q: = (5, 7), s: = (8, 3);
print (midpoint (q, s))
```

will produce something like

```
6.5    4
```

The body of the procedure is supposed to produce an object of mode **point**. It does so by a structure display, which specifies each of the necessary components. The first is *0.5* * (*x* **of** *j* + *x* **of** *k*) and the second is *0.5* * (*y* **of** *j* + *y* **of** *k*).

There is, of course, no need for the result of a procedure (or of an operator for that matter) to have the same mode as any of its operands. For example, it would be possible to define a procedure *less* to use two **points**, producing a boolean

```
proc less = (point a, b) bool:
   comment Compares two points. The relation is satisfied if the first point
                  is nearer the origin than the second
   comment
(x of a↑2 + y of a↑2 < x of b↑2 + y of b↑2)
```

In discussing new mode declarations it was implied that the object on the right of the = sign is the description of a mode in the full sense of the term. This is largely true, but there is one important difference—any square brackets in the new mode definition must be supplied with actual bounds unless the square brackets are immediately preceded by **ref** (a situation that you will not yet have encountered). For example, in declaring a mode **match** suitable for the variables game and bestgame used earlier the correct form would be

```
mode match = struct ( [1 : 12] char opponents, int for, againstbool home)
```

The form

```
mode match = struct ([ ] char opponents, int for, against, bool home)
```

is wrong, *even though* it represents a formal mode more precisely than the first version.

In most practical applications, collections of records of the same 'shape' are organised into files. Files can be represented in many ways—they can, for instance, be punched on cards, or printed on line-printer paper, one record to a line. They can also be recorded in the backing store of a computer and read back as if they were punched on cards. In the working store of a computer, however, a file is best represented as a one-dimensional array of records.

An array of records (which is, of course, distinct from an array *within* (a record) is declared in exactly the same way as an array of elements of any other mode. Some examples are

```
[1:20] struct (int q , r; real zz) ab
```

or

```
[1:3, 1:6] point plan
```

The first declaration sets up an array of 20 records. Each record has three fields—*q*, *r* and *zz*. The **real** field of the last record in the array would be referred to as *zz* **of** *ab* [*20*]. No brackets are necessary.

The second declaration assumes that mode **point** has already been declared. Here, *plan* is an array with 18 elements.

Files are usually ordered with respect to the contents of one of the fields in the records. A file that was a list of employees in a factory would probably be arranged in increasing order of 'works number' (a unique number given to each employee when he joins the company) or possibly in alphabetical order of names. An ordered file makes it much easier to locate particular entries, for a logarithmic search can be used.

The field used to order a file is called a *key field*, and any particular value of this field is a *key*.

The processes of searching and sorting involve comparisons of keys, in order to decide whether one is equal to or 'less than' another. If the keys are integers (like works numbers) there is no difficulty but, if the keys are alphabetical, considerable care is needed. It is essential to start with proper definitions of 'equals' and 'less than'.

Equality is the simpler of the two. Two character strings are deemed to be equal only if

(a) they have the same length, and
(b) every character in one is identical to the corresponding character in the other.

The operation 'less than,' when applied to pairs of character arrays, could have several possible interpretations. The most natural is the operation that tests for alphabetical order, so that "ss" < "tt" implies that "ss" would be found nearer the first page of the dictionary than "tt". The normal conventions about alphabetical ordering imply that "s" < "ss".

Comparisons between strings are not trivial. Nevertheless, Algol 68 includes built-in definitions that allow the operators =, ≠, <, >, < = and > = to be used on character strings without formality.

Sometimes the rules for ordering are even more complex. An obvious example is a field that contains a person's first and second names. Ordering is usually done on the second name, the first name being used only as tie-breaker if the second names of two people are the same. The following procedure determines whether two fields of this type—*a* and *b*—are in alphabetical order. It is assumed that in each case the first name starts at the element with subscript 1, and that the second name is separated from the first by a single space. The procedure works by locating the position of the space in each string and comparing slices of the fields in the appropriate order.

```
proc compare = ([ ] char a, b) bool:
  (int sa, sb;
    for j while a [j] ≠ " " do sa: = j + 2 od;
    for j while b [j] = " " do sb: = j + 2 od;
    if a [sa: upb a] ≠ b [sb: upb b]
      then a [sa: upb a] < b [sb: upb b]
      else a [1: sa − 2] < b [1: sb − 2]
    fi
  );
```

In practice, the process of comparing people's names is complicated even further because some use only their initials, some give their middle names, and some the initials of their middle names. This is the kind of difficulty that many commercial programmers spend time solving!

If the records in a file are not in order, they must be sorted. Merge sorting (described in chapter 13) is an excellent method but has the drawback that it requires a considerable amount of extra space. If you look at the definition of the procedure *merge* on p. 194, you will see that, whenever it merges two sequences of length m and n, it creates (albeit temporarily) an array of $m+n$ elements. This implies that, in order to sort 1000 elements, at some stage 2000 storage locations will be required. If each element is a large object (like a record) this may well overtax the capacity of the machine and the program will break down.

A good solution to this problem lies in sorting not the records themselves but much smaller *tokens*. Each token is an integer that represents the position of a record in the main array. Initially the tokens are set up in their own array, in the same order as the records that they represent — that is, 1, 2, 3, . . . Such an array might be as follows.[1]

Main array		Token array	
Key field	Other fields (not shown)	Initial order	Final order
BURL NOGGLE	-------------------	1	2
NEWT LOKEN	-------------------	2	7
WALTER SCHUYLER	-------------------	3	4
PAUL NIGGLI	-------------------	4	1
MILDRED STRUNK	-------------------	5	3
UNITY STACK	-------------------	6	6
MUSTAPHA MBOOB	-------------------	7	8
ADRIAN STRUNK	-------------------	8	5

When the tokens are sorted, they are ordered not by their own (integer) values but by the key fields of the records they stand for. Thus (for example) element 7 should come before element 5, because MBOOB comes before STRUNK in dictionary order. The order of the tokens after sorting is shown in the last column of the table.

From the sorted tokens it is now possible to find the key order of the records in the main array. For example, the first in dictionary order turns out to be record number 2. To print out the whole file would require a loop such as

```
for j to upb mar do print ((newline, mar [xx[j]]))od
```

where *mar* is the identifier of the main record array and [*1* : **upb** *mar*] **int** *xx* is the array of tokens.

The mental equipment provided in this chapter is adequate for the solution of a simple but realistic problem in data handling.

[1] The names in this chapter are taken from *The Guardian*, 1 February 1975.

A university maintains a card file giving details of all the students currently attending. The file is kept in the order of the students' official numbers, which are assigned to them when they first register. Each record in the field has the structure

> **struct** (**int** *number*, [1:24] **char** *name*, [1:4] **char** *style*, [1:12] **char** *department*);

(In practice a student file would have many more fields, but the abbreviated record will serve for our illustration.)

A section of the file might be

721053	CLOPPER ALMON	MR	PHYSICS
721062	MONETEE REDSLOB	MR	COMPUTING
721345	ADELHEID POPP	MRS	ENGLISH
721350	DESIREE TUITS	MISS	MATHS
721477	JOHN SWEENEY	MR	COMPUTING
721580	MERLE FAINSODD	MS	COMPUTING
721600	RANDY HADDOCK	MR	CHEMISTRY
731003	FRED PLOG	MR	MATHS
731094	XENOBIA FATT	MRS	COMPUTING
731184	ELGIN GROSECLOSE	SIR	COMPUTING
731294	FRIEDRICH NAUSEA	MR	PHYSICS
731455	JUSTUS JEEP	MR	COMPUTING
731712	ANTHEA GROSECLOSE	LADY	COMPUTING
731904	MOHAMMED MCDOOM	MR	ENGLISH
741094	TIZIANA SOZZI	MS	ITALIAN
741127	FLORENCE MOOG	MISS	COMPUTING
741277	ISRAEL SMUT	MR	CHEMISTRY
741394	HARDY WORM	MR	MATHS
741644	IRVING FATT	MR	COMPUTING
999999	DUMMY		

The exact number of records in the file is not known precisely, since it changes daily; but the file is terminated with a dummy record with 'student official number' set to 999999.

At various times, professors are sent lists of all the students registered in their departments. These lists generally appear in alphabetical order of the students' names, and not according to their numbers as in the main file. The problem to be considered is that of using the main student file to produce an ordered list for the department of computing.

The initial approach will be from the top downwards. At the highest level, the algorithm needed is a simple one.

(1) Read the entire student file, and pick out all the records that refer to students in the computing department, setting them aside in an internal file or array.

(2) Sort the internal file, using the name field as a key.
(3) Output the sorted records with suitable labelling.

It is assumed that there are not more than 500 students in the computing department. (This assumption allows arrays of fixed size to be set up.) A skeleton program is now written, as follows.

```
begin
  mode student = struct(int number,[1:24]char name,[1:4]char
     style,[1:12]char department);
  [1:500]student list; int n: = 0; student next;
  while read(next); number of next ≠ 999999
     do if department of next = "COMPUTING   " then
              list[n+: =   1]: = next;
           if n> = 500  then print((newline, "TOO MANY
              COMPUTING STUDENTS"))
        fi
     od;

     sort(list[1:n]);

  print((newline,"ALPHABETISED LIST OF COMPUTER SCIENCE
        STUDENTS",
        newline, "NUMBER   NAME   STYLE   (DEPT)"
     ));

  for j to n do  print the jth record in list  od

end
```

The areas still to be filled in are concerned with sorting and printing; they are circled in the skeleton program.

The records will be sorted using some of the methods already discussed — by setting up a file of *integer* tokens and using the merge-sort method. Of course it will be necessary to redefine the procedure *merge* so as to order elements according to the key fields of the records to which they point.

The entire program is as follows.

```
begin
  mode student = struct(int number, [1:24]char name,
     [1:4]char style, [1:12]char department);
  [1:500] student list;
  proc comp = ([ ]char p, q)bool:
     comment Compares two names p and q, and produces true if
                 p is alphabetically less than q
     comment
     ( int sa, sb; for j while a[j] ≠ " " do sa: = j+2 od;
                   for j while b[j] ≠ " " do sb: = j+2 od;
```

```
                if a[sa:upb a] ≠ b[sb:upb b]
                  then a[sa:upb a] < b[sb:upb b]
                  else a[1:sa−2] < b[1:sb−2]
                fi
  );
proc merge = (ref[ ]int a, b)[ ]int:
  comment Merges two ordered arrays of tokens, a and b comment
  (int pa: = 1, pb: = 1;[1:upb a+upb b] int c;
  for pc to upb a+upb b
    do if pb > upb b then c[pc]: = a[pa]; pa + : = 1
        elif pa > upb a then c[pc]: = b[pb]; pb + : = 1
        elif comp(name of list [a[pa]],name of list[b[pb]])
        then c[pc]: = a[pa];pa + : = 1
        else c[pc]: = b[pb]; pb + : = 1
      fi
    od
  );
proc tokensort = (ref[ ]int x)void:
  (if upb x > 1
    then tokensort(x[1:upb x ÷ 2]);
      tokensort(x[upb x ÷ 2 + 1:upb x]);
    x: = merge(x[1:upb x ÷ 2], x[upb x ÷ 2:upb x]
  fi
  );
proc sort = (ref [ ] student w) [ ] int:
  comment Sorts an array of student records by their names and
          produces a sorted list of tokens. w is unchanged
  comment
  ([1:upb w] int t;
    for j to upb w do t[j]: = j od;
      tokensort(t); t
  );
  comment Now the main program can start comment
  int n: = 0; student next;
  while read (next); number of next ≠ 999999
do if department of next = "COMPUTING   " then
  list [n+: = 1]: = next;
    if n > = 500 then print ((newline, "TOO MANY COMPUTING
                                 STUDENTS"))
    fi
  fi
od;
[1:n] int xx: = sort (list [1:n]):
print ((newline, "ALPHABETISED LIST OF COMPUTER
          SCIENCE STUDENTS",
            newline, "NUMBER   NAME   STYLE   (DEPT)"
         ));
    for j to n do print ((newline, list [xx[j]])) od
    end
```

In this program the read command is *read* (*next*), not *read* ((*newline*, *next*)) because each record starts with an integer rather than a character array.

The results expected from this program are as follows.

```
ALPHABETISED LIST OF COMPUTER-SCIENCE STUDENTS

NUMBER          NAME                STYLE      (DEPT)

  721580  MERLE FAINSODD             MS      COMPUTING
  741644  IRVING FATT                MR      COMPUTING
  731712  ANTHEA GROSECLOSE          LADY    COMPUTING
  731184  ELGIN GROSECLOSE           SIR     COMPUTING
  731455  JUSTUS JEEP                MR      COMPUTING
  741127  FLORENCE MOOG              MISS    COMPUTING
  721062  MONETEE REDSLOB            MR      COMPUTING
  721477  JOHN SWEENEY               MR      COMPUTING
```

In most data-processing installations, procedures for sorting and comparing names would be included in a standard library. Many computers are equipped with special-purpose data-retrieval languages, which allow a list of this kind to be generated by a few simple commands

```
select on (department = "COMPUTING"); sort on name; print
```

These languages save time on specialised applications, but do not offer the power of a more generalised system like Algol 68.

EXERCISES

***14.1** (2) A file of exam results contains the following items of information for each student

name (maximum of 24 characters)
department (maximum of 12 characters)
marks (out of 100) for four different papers in English, Mathematics, Computer Science and Philosophy.

(a) Define a suitable structure for a record in this file, and declare an array of 100 such records.

(b) Assuming that 100 such records have actually been read in and set up in your array, write a sequence of code that finds and prints out the name of the student with the highest aggregate score.

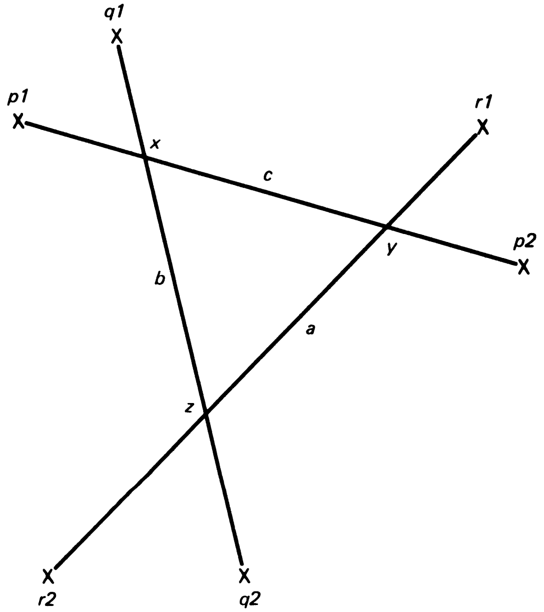
*14.2 (5) In two-dimensional coordinate geometry, a point is characterised by two real numbers—its x and y coordinates. Likewise a line is also characterised by two real numbers—m and c in the line equation y = mx + c.

(a) Declare modes **point** and **line** for variables that represent points and lines, respectively.

(b) Consider the following diagram. The positions of points P1 P2, Q1 Q2 and R1 R2 are given as data. The segment of program below the diagram calculates the area of triangle XYZ by the formula

$$A = \sqrt{\{s(s-a)(s-b)(s-c)\}}$$

where s is the semiperimeter and a, b and c are the lengths of the three sides.



point *p1, p2, q1, q2, r1, r2, x, y, z*; **line** *p, q, r*;
read ((p1, p2, q1, q2, r1, r2));
p: = *join(p1, p2)*;
q: = *join (q1, q2)*;
r: = *join (r1, r2)*;
x: = *crosses (p, q)*;
y: = *crosses (p, r)*;
z: = *crosses (q, r)*;
real *a* = *distance (z, y)*, *b* = *distance (z, x)*, *c* = *distance (y, x)*;
real *s* = *0.5* * *(a + b + c)*;
real *area* = *sqrt (s * (s − a) * (s − b) * (s − c))*;
print (("AREA IS", area))

Write suitable definitions for the procedures *distance*, *join* and *crosses*.

***14.3** (5) (P) It is proposed to set up a navigational surveillance system over a roughly square area of the Atlantic Ocean, 1000 × 1000 km in size. The system uses a file in which each record represents a ship. The following information is stored

name (20 characters)
position at midnight (in 1 km coordinates)
course at midnight (in degrees East of North)
speed at midnight (in km/hour).

The file is updated every midnight.

Write a 'rescue' program. The data are the position and time of an event (such as a ship-wreck or ditched aircraft). The program is to scan all the shipping records, choose the ship able to be on the scene in the shortest possible time and print out

(a) its name
(b) its new course
(c) its expected arrival time at the scene.

The data for the shipping file will be provided on cards. The name of each ship will be 20 characters long, and will be followed by four numbers giving the position, course and speed at midnight. The file ends with a dummy record, in the form of the ship's name ZZZZZZZZZZZZZZZZZZZZ. The time of the event will be supplied in hours after midnight.

You may assume

(a) that the earth is flat
(b) that a ship can (and will) alter course instantaneously
(c) that all ships in the area are represented in the file
(d) that no ships have changed course since midnight.

15 *Problem Definition and Program Documentation*

"That's very nice, but it's not what I wanted'

Most of this book has been devoted to the art of programming. The subject has been treated very much in isolation — the problems to be programmed just appeared, as it were by magic, at the ends of the chapters. It has been implied that, once a program has worked correctly, that is an end to the matter as far as the programer is concerned.

This last chapter is an attempt to put the whole subject of programming into its correct context. As noted at the beginning of the book, computers are machines for solving problems, and programming is only one of the steps needed to advance from the first recognition of a problem to its eventual solution.

It is usual to regard the entire sequence of steps as having three major phases

(a) systems analysis — defining the problem and deciding how to solve it
(b) programming the proposed solution
(c) operations — running the program on a regular basis.

In many cases each of the three phases is done by a different person. In most computing organisations, 'systems analyst' and 'programmer' are two distinct job titles, and the 'operations manager' who runs the computer is again a separate appointment.

In the systems-analysis phase, the client and the solver (using the names introduced in chapter 1) get together to decide exactly what is to be done, and to work out a way of doing it. At first the client's description of what he needs will be vague, and the solver must be able to ask the right questions and sift the replies, until a clear requirement emerges. Eventually the client and solver will produce an agreed problem definition. The following are some of the items that it should contain.

(1) A clear statement, in plain language or mathematical terms, of the purpose of the system. There must be an unambiguous statement that describes the intended input, and shows how the results are to be derived from the data.

(2) A brief description of the general hardware configuration to be used (type

of central processor and peripherals). If the program is meant to be portable (that is, capable of being run on computers of different types) the problem definition should say so. Ideally, the programming language should be chosen freely by the solver and need not enter into the problem definition; but in practice many clients will require the use of a particular language, so as to conform to a commercial policy.

(3) The general format of any files to be kept in the backing store.

(4) A detailed description of the input format for the data, with examples.

(5) A description of the proposed output format, if possible illustrated with a mock-up drawn on squared paper.

(6) The actions to be taken for illegal input.

(7) An agreed statement about the importance of the data being correct. This may vary from one problem to another, and between different parts of the same problem.

(8) Some indication of the following points.
 (a) When is the system to be ready for use?
 (b) What is its expected life, and how often is it to be used?
 (c) What is the expected volume of data?
 (d) What is the expected performance of the system (in terms of cost per run)?
 (e) What arrangements are to be made to maintain the system after it is put in to use? (Maintenance includes the correction of residual errors and the making of alterations to handle unforeseen new requirements.)

The problem-definition document should eventually be signed by both client and solver, so that it can be used as the basis of an agreement. There is plenty of experience to suggest that, if this is not done, the client and the solver may eventually disagree on whether the problem has been solved correctly.

The drafting of the problem definition is certainly the most difficult task in the whole process of problem solving. It calls for experience, tact, modesty and in many cases a good technical knowledge of specialised areas such as medicine or production engineering. There are three cardinal rules.

First, question everything the client tells you, and believe nothing until you have checked it for yourself. The client must be able to satisfy you on such matters as why he wants the system, whether he can pay for it, whether he and his staff have the experience to use it, and whether the various decision rules are really as clear-cut as he may claim. If the client is not himself a person with computer experience, he will probably have a simplified idea of the way his organisation actually runs, and he may be passing this inadequate information on to you in good faith.

Second, consider the impact of the system on the whole organisation. Will it make more work for some of the members? Will it endanger the security of confidential information? Will it generate unemployment? Both you and the client should be aware of these difficulties since they may well influence the over-all design of the system.

Third, be totally cynical of your own ability as a system designer, and of the speed and accuracy of the person who is to do the actual programming, whether you or someone else. Make a careful, honest and reasoned estimate of the time you need to design and build the system—and then quadruple it to get a realistic figure that can be incorporated in the agreed problem specification. Work completed ahead of its schedule is always admired, but work that is late often causes great expense and bitterness. It is always best to be pessimistic about your own abilities.

Similar considerations should apply when you predict the performance of your system. Your best private estimate should be downgraded by a large factor before it goes into the problem definition.

The whole question of input format is one where you must take account of many practical factors. If the client has an operations manager, it is best to consult him—he may well give valuable advice about card layouts and the volume of data that his department can be expected to prepare each week.

A key decision to be made is the standard of accuracy needed in the data. An easy answer is: 'It must all be exact'. In practice, errors often occur in card punching and data transmission, and their elimination can be very expensive. By considering different kinds of problem, it is possible to define a whole spectrum of different needs.

At one end, there are interactive systems where errors are immediately obvious and simple to correct. Consider an airline booking system. If a travel agent mistakenly types a date as '3/55/76' the system will immediately respond with "IMPOSSIBLE DATE" and nothing will have been lost except a few seconds of the agent's time.

At the other extreme, data errors can have irreversible and often tragic consequences. Consider a town where there is a computerised medical-data bank. Every inhabitant wears a metal 'dog tag' that gives his medical registration number. When the victim of an accident is brought in to a hospital, his medical record can be retrieved in seconds so that he can be given, say, a blood transfusion without delay. If a mistake is made in keying the number into the computer terminal, the machine will display the record for a different person, who may have an incompatible blood group. Such an error would eventually be discovered, but too late for the unfortunate patient, who might already be dead.

Most applications fall between these two extremes. In deciding on an appropriate degree of accuracy, two factors must be taken into account.

(a) What are the consequences of an error? Being left off a mailing list for advertisements is not as drastic as having your salary calculated on the wrong basis.

(b) If the errors affect an individual person, does he have the opportunity to check the matter for himself? For instance, all bank customers eventually receive printed statements from the computer that they can verify for themselves; but on the other hand, examination results and medical records are generally kept secret.

Data preparation is, as has been noted, a chancy matter. Fortunately, there are several techniques that reduce the rate of undetected errors.

Most errors arise in the manual transcription of data from written sheets to punched cards or other keyboard devices. The best way of avoiding keying mistakes is to eliminate the keying process altogether. This can be done in many circumstances. Thus all automatic measuring and recording devices, such as might be part of an automatic weather station, can be made to produce records on paper or magnetic tape, which can be read directly by a computer without human intervention. Again, documents can often be printed with magnetic characters or specially shaped numerals that can be read directly by robot scanners and fed to a computer without manual keypunching. Library books and personal library cards can be fitted with magnetic strips. When you take a book out of the library, your card is presented to a scanner together with the book you are borrowing. The scanner is connected to a computer that can keep library records without the need for every title and every borrower's name to be typed by hand.

If the key punching is unavoidable, there are still several ways of reducing the error rate.

The simplest fundamental precaution is to make sure that all the data being keyed is legible to the keyboard operator. You should arrange for the use of carefully designed forms, and insist that the data be written in a clear hand with stylised characters that emphasise the differences between letters O, I, Z and S on one hand and the digits 0, 1, 2, and 5 on the other.

Above all, find out and follow your local rules for telling letter 'oh' from digit 'zero'. There are two conventions, each believed, by those who practise them, to be 'universally accepted': one group maintain that Ø means 'oh', while the other, with equal fervour, will say that it means 'zero'. The local operations manager will be able to help you in this basic but vital area.

Another widely used method of improving accuracy is called *verification.* It consists of having every item of data typed twice, by different people. The two streams of data are compared automatically and any difference is brought to the attention of the operator, who can make the appropriate correction.

In systems that involve databases, the keys of the individual records may be used repeatedly. In a banking system, for example, the key would be an account number and it would be punched every time that the corresponding account was used.

If straightforward consecutive integers are used as account numbers, there is every chance that a simple slip — such as putting '37748' instead of '37448' — will lead to an entry being made in the wrong account. A bank that did this often would soon lose all its customers.

To help with this problem, it is customary, when first assigning a new account number, to add a *check digit*. This is an extra digit whose only purpose is to detect keying errors. The check digit is worked out from the other digits in the number by a fixed rule, such as
'Take the first digit, add twice the second, three times the third, and so on. Discard the tens digit of the answer'.
For instance, if the basic number were 37448, the calculation would be

$$1*3+2*7+3*4+4*4+5*8 = 85$$

and so the check digit is 5 (carding the 8) and the augmented account number is '374485'.

Later, whenever any account number is keyed, the check digit is recalculated and compared with the one supplied. Any discrepancy must be due to an error. Thus if the number keyed is 377485, the check digit based on the first five digits turns out to be 4, and so there is clearly an error.

This simple system will detect most but not all errors. The level of security can be increased indefinitely by adding more check digits worked out by different rules.

The method of sum checks cannot be applied if the data consist of completely arbitrary numbers, like sums of money. An alternative system is *batch totalling*. Here the data are split up into 'batches' of 100–200 numbers each and, before being keyed, the numbers in each batch are added up on a desk calculator. The total is entered into the computer separately. Eventually, the computer can check that the total of the numbers in each batch does indeed agree with the preliminary tally, and can reject any batch that shows a discrepancy. Again, the system is not totally error-proof, since mutually cancelling errors can still occur, but it offers a worthwhile degree of protection.

To summarise, there is, unfortunately, no method of data preparation that offers absolute certainty of accuracy. Where correctness is of vital importance, you would do well to ensure that the ultimate responsibility for the data lies not with you but with the primary source from which the data first came. To illustrate this point, consider a university where exam results are processed by computer. Errors in the marks cannot be questioned by the students concerned and could lead to serious injustices. A suitable processing system begins in the usual way, with the marks being collected from the various departments, keyed and verified. At this stage, before any permanent decision about any student is made, the marks are printed out, department by department, with the inscription

I CERTIFY THESE MARKS AS CORRECT

The lists are sent for scrutiny, possible correction and signature, to the heads of the various departments concerned. The decision program is not run until all the sheets are signed and returned.

This arrangement has two advantages. It protects you from blame if any mistakes should subsequently be discovered, and it forces one last check to be carried out just before the marks are used.

Another important aspect of problem definition is the design of the output format. Here you can be guided by the client's needs and a few simple rules.

(a) Every item to be printed must be labelled; numbers by themselves are meaningless.
(b) Avoid huge tables of numbers; graphs or charts, no matter how crudely printed, are always easier to assimilate.
(c) Make full use of preprinted stationery if appropriate. This again is an area where the operations manager can help you.
(d) Prepare a sketch of your proposed output on squared paper, so that you can see how the various items fit. Discuss the layout with the client.
(e) Save paper; do not produce large amounts of output unless you are sure that someone is going to read it. A busy manager or professor may well find time to look at a two-page summary, but he is most unlikely to read a detailed 100-page report — he will simply throw it away.

Part of the problem definition should deal with the handling of errors in the data. Whilst some mistakes in the data are logically undetectable, you should ensure that all the errors that can be found are found. The system must deal gracefully with errors; it should print out a warning message and if possible continue processing the data. In some circumstances, such as specifying a wrong file name, it may be necessary for the program to close down; but it should never simply collapse, even with the most unlikely errors in the data.

These ideas are illustrated by the following problem definition, for a very simple system derived from the label-printing program discussed in chapter 10.

LABEL PRODUCTION. PROBLEM DEFINITION

General purpose of system

This system is used to produce address labels from files of names and addresses originally punched on cards and permanently recorded in a backing store. The labels can be stuck to envelopes and used to circulate information to students, customers, club members, etc.

To use the system, the client will supply a list of names and addresses in the format given below. They will be punched and verified, and a compact listing will be returned to him for checking. After any necessary corrections have been made and checked, the data will be recorded in the backing store for future use.

At any time, the client may request a production run. Each address in the file will then be printed on special stationery that consists of blank sticky labels stuck to a waxy backing sheet.

The system comprises two separate programs — one that reads names and addresses from cards and lists them on ordinary stationery for checking, and one that reads data from the file store and prints the sticky labels.

Configuration

The program runs on an ICL-1900 computer with card reader, a line printer and a backing store.

Input format

The data are presented as a series of cards, or card images from the backing store. Each address is punched on one card, and consists of up to five lines. The lines will be of arbitrary length subject to the following limits

line 5: 25 characters
line 2: 21 characters
line 3: 17 characters
line 4: 13 characters
line 5: 25 characters

Each line is terminated by a + sign, which is not part of the address. Empty lines must still be indicated by + signs.

The last address is followed by a card which has a '/' in its first column.

For obvious reasons, the symbol + may not occur anywhere in the address, and / may not occur as the first character of the first line.

A sample set of data is

```
O.KLEMPERER+12 VINE ST.+LONDON EC1++ENGLAND+
C.M.GIULINI+7 OLD KENT RD.+MAIFSTONE+KENT+ENGLAND+
B.WALTER+17 BOND ST.+EDINBURGH++SCOTLAND+
M.SARGENT+PARK LANE+NEATH+CARDIGAN-SHIRE+WALES+
```

Output formats

The monitor program, which lists names and addresses for checking purposes, uses the following format for each address

first, a blank line
next, a copy of the data card, exactly as punched
next, another blank line
next, the name and the address correctly laid out. The lines will be indented as follows

line 1: 0 spaces
line 2: 4 spaces
line 3: 8 spaces
line 4: 12 spaces
line 5: 0 spaces

Where the data card contains a detectable error, the laid-out address will be replaced by a suitable warning message. An example of output is

```
O.KLEMPERER+12 VINE ST.+LONDON EC1+ +ENGLAND+

              O.KLEMPERER
                 12 VINE ST.
                    LONDON EC1

              ENGLAND


C.M.GIULINI+7 OLD KENT RD.+MAIFSTONE+KENT+ENGLAND+

              C.M.GIULINI
                 7 OLD KENT ROAD
                    MAIFSTONE
                       KENT
              ENGLAND


B.WALTER+17 BOND ST.+EDINBURGH+SCOTLAND+ +

              B.WALTER
                 17 BOND ST.
                    EDINBURGH
                       SCOTLAND



M.SARGENT+PARK LANE+NEATH+CARDIGAN-SHIRE+WALES+

***** LINE 4 OF ADDRESS TOO LONG (MAX=13 CHARACTERS)


G.MAHLER+ . . .
```

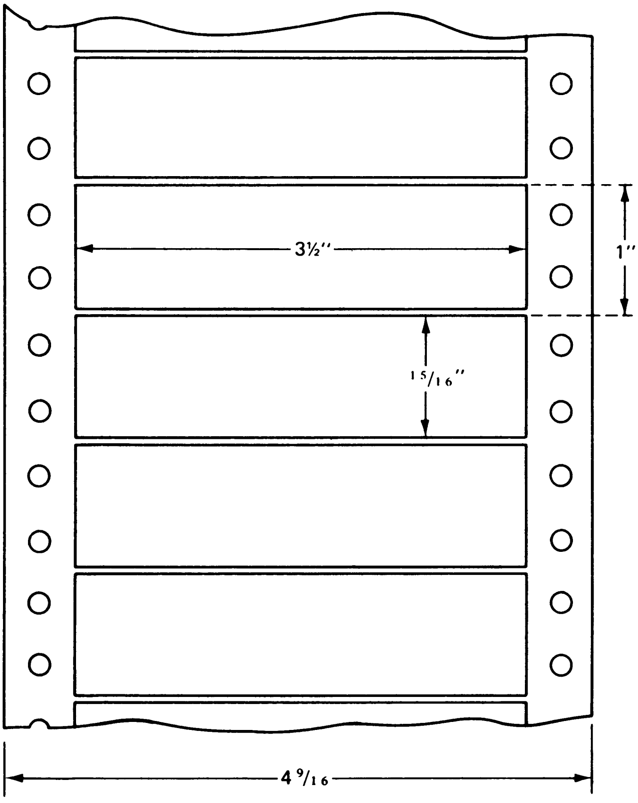
The main program, which generates the adhesive labels from data that have already been checked, uses stationery of the standard size shown opposite. (This is available from Z. Bloggs and Sons Ltd (computer stationers) as catalogue item number StL747.)

The layout of each label is identical to that described for the monitor program. The original data are not listed at all. The data should contain no errors but, if they do, the corresponding label will be printed with the words 'ERROR IN ADDRESS'.

Error detection

The system can detect the following faults in the data.

(a) Wrong number of lines in the address. (There should be five, including empty ones.)
(b) Too many characters in any one line. (The address should conform to the rules given under the section on input format.)



Other mistakes (such as mis-spellings or duplication of the same name and address) will not be found automatically.

Accuracy

The system ensures that the labels printed will be identical to the addresses listed by the monitor program. The responsibility for checking and correcting these addresses will lie with the *client.*

Other considerations

The system described in this definition will be ready 4 months[1] after the definition has been agreed. Its expected life is at least 5 years. The estimated load is as follows.

The file store will eventually hold about 100 different address files with a mean of 3000 names in each. On average, each list will be printed four times a year. The system will therefore be used to generate some 1 200 000 labels a year.

[1] Private estimate: 1 month.

The over-all production cost of each label will not exceed $2\frac{1}{2}$p[1] (at 1976 prices).

Any errors discovered in the system up to *one year* after it is delivered will be corrected as quickly as possible. Changes to the definition of the system, and the correction of errors that come to light after the first year of use, will be the subjects of separate, new agreements.

Signed (Client)

(Solver)

April 24th, 1976

When the specification has been agreed, the necessary program can be written. Enough has been said about this topic in the earlier chapters, and so it is possible to pass directly to the final stage of problem-solving.

The last stage in getting the actual solution to any problem is operational — running the system on a regular basis.

Programmers usually have little personal contact with the computer operators, data-preparation staff, and other people who actually use their programs. The key to smooth and successful operation is good documentation.

The documentation of a system is the collection of papers that describes its purpose, design, construction, use and maintenance. Many programmers think of documentation as an extra job to be tackled when the development of a new system is complete. They are anxious to get started on the next project,

[1] Private estimate: 1.1p, calculated as follows

Estimated CPU time for 3000 labels: 20 seconds, cost (at £12.00 per minute)	£4.00
Number of lines of output (including blank lines) needed per label = 6	
Output cost (at 0.05p per line)	£9.00
Cost of stationery (at 0.5p per label)	£15.00
Cost of mounting special stationery	£5.00
Total production cost for 3000 labels	£33.00

£33/3000 = 1.1 pence

Note that these private estimates do *not* form part of the program specification.

so that writing up the previous one is seen as a time-wasting chore. This view is gravely mistaken; no matter how brilliant a system may be, it is entirely valueless unless there are good instructions on how to use it. As will quickly become apparent, a well-planned project generates its own documentation, so that all that is necessary when the system itself is complete is a careful editing job.

It is worth remembering that few people will ever study your programs in detail, but many will read your documentation. In the long run it is by documentation — in the form of user guides, reports or scientific papers — that your competence as a professional computer programmer will be judged.

A system should be documented at three levels

(a) manager level
(b) user level
(c) maintenance level.

Each level serves a different purpose and is aimed at a different readership, so that its style should be chosen accordingly.

The *manager-level* documentation should state briefly what the system does. The description should consist of no more than a title and a short paragraph, and be suitable for inclusion in a program catalogue. The aim is to attract the attention of someone who might eventually use the program — no more.

Examples of manager-level documentation are

LABELS

This program prints adhesive address labels from a list of names and addresses originally punched on cards and held in a file

and

COIN ANALYSIS

This program reads in a set of records that give the net pay of a number of employees, and determines how many notes and coins of each denomination are needed to make up the wage packets

The *user-level* documentation amounts to a 'user's guide'. It should tell the client (or his employees) exactly how to use the system — nothing less and nothing more. In particular, it must not include any descriptions of how the system actually works, since this would serve no purpose and might actually confuse the users.

When you write user documentation, you assume that the reader is familiar with general computer procedures like punching cards and submitting jobs to the local system, but that he knows nothing about your particular system. Everything must be explained in detail, with examples. (Many people do not even bother to read the text, but just copy the examples you give, so the presence of good, representative examples is vital.)

The essential elements of good user documentation are descriptions of

(a) the general purpose of the program
(b) data preparation and formats
(c) output formats
(d) job submission.

A good basis for each of these areas can be borrowed from the original system specification. Thus the sections on the general purpose of the system and the output formats can be copied over as they stand. The section on data can begin with the part of the specification deeling with the input format, but should also include material that shows in detail how to set up a typical data card from hand-written or typed records.

The section on job submission should discuss the way in which the program and data are to be presented to the computer. It must consider the necessary job description and give rules to relate the resource requests (for CPU time, store and output volume) to the characteristics of the actual batch of data being used. The section must incorporate the listing of a complete sample job, including its job description.

Every new programmer believes that he can write complex programs correctly at the first try; but he discovers, painfully and tediously, that he is prone to make mistakes and that, until these mistakes have been eliminated, the program is of no practical use whatever. Exactly the same applies to user documentation; the first draft will contain mistakes and the system as a whole will be entirely useless. Intellectually, the mistakes will usually be trivial (like specifying the order of two cards incorrectly) but like 'silly' program errors they will effectively prevent the system from working. User documentation must therefore be *debugged*, very much like a program.

An important difference is that the instructions in the user documentation are addressed to a person instead of a computer, and you must therefore find a human guinea-pig on whom to try them out. The person chosen should preferably have a 'devil's-advocate' mentality, and deliberately misinterpret and misunderstand anything that is ambiguous or doubtful.

Once you have written the first draft of your user documentation and found a willing subject, the correct procedure is to set him a problem designed to exercise as much of your system as possible, to give him the instructions and then to watch in silence. If you see your subject in doubt, or doing the wrong thing, resist the temptation to correct him or to explain what the instructions actually 'mean'. What you are seeing is a 'bug' in your own documentation.

After the trial run, correct your documentation as necessary and repeat the entire procedure, preferably with another subject who is entirely new to the system and has not been 'corrupted' by contact with an incorrect description. Continue this process until the user documentation is really unambiguous and correct — until it 'works' for all the sample users you can find.

The *maintenance-level* documentation is a complete technical description of your program. The reader will be a qualified programmer who has the job of correcting errors in your system or changing it to meet new specifications.

If your system development has gone smoothly, most of the maintenance documentation should already exist when the system is complete. The documents should include

(a) a complete listing of all the programs, with sample data and outputs
(b) a description of the purposes of all the variables used in the program
(c) a description of each procedure used in the program
(d) any charts, decision tables or flow diagrams that you may have used in writing the program
(e) a description of the over-all program structure, showing how the various procedures use one another
(f) an account of any mathematics that you may have incorporated in the program.

This information can either be incorporated as comments in the program listing itself, or given in a separate 'program description'. A common method is to include minimal descriptions of variables and procedures in the program itself, and to write up the other items separately.

The end of the book is approaching. It only remains to give a few final comments on the techniques of solving real problems in information handling and processing.

If you are to solve a problem, you need to understand it and to know why it has arisen. Whenever you are about to start writing a program or designing a system you should ask yourself, why, ultimately, am I doing this? Will it solve the problem, or will it make it worse? Is the problem worth solving? Is the cost acceptable?

Solving problems requires skill in the use of tools. Some of the tools available in the Algol-68 system, such as expressions, procedures and arrays have been discussed in this book, but there are a number of other facilities that have not even been mentioned, simply because they are of most use in solving advanced problems that arise in specialised areas.

In solving problems you must be aware of what others have done. Very few problems are really new; nearly always someone has been here before.

By reading books and technical journals you may often find a method, a procedure or a complete program that can be incorporated in your system.

In certain areas such as sorting or numerical analysis (a branch of mathematics) all known methods have been carefully documented and analysed, and are available as procedures in program libraries. Writing new programs to solve problems in these areas is *always* a waste of time.

Above all, to solve problems, you need a quality called *gumption.* The recipe for gumption reads like that of a magic potion, and some of the ingredients may not be available. The main constituents are the following.

(1) care (check everything you do)
(2) creative cynicism (never believe anything that has not been proved)
(3) calm (work in a quiet place, and shut out distractions)
(4) leisure (do not hurry but take your time—correctness is better than speed)
(5) sobriety (if you have had anything to drink, do not even try to work, but sleep it off instead; a procedure that I once wrote after drinking half a bottle of burgundy at lunch had so many mistakes that they were still coming to light more than a year afterwards).
(6) peace of mind (be happy; if you are worried or distracted about anything, go away and wait until you feel better about it).

Farewell, reader, and plenty of gumption!

EXERCISES

15.1 (9) Write a critical review of this book.

Further Reading

FURTHER STUDY OF ALGOL 68

F. G. Pagan, *A Practical Guide to Algol 68* (Wiley, London, 1976).

A. Learner and A. J. Powell, *An Introduction to Algol 68 through Problems* (Macmillan, London and Basingstoke, 1974).

These are both introductory books, covering somewhat different areas of Algol 68 than the present volume.

A. S. Tanenbaum, 'A Tutorial on ALGOL 68',*ACM comput, Surv.*, 8 (1976) p. 155. An excellent article giving an overview of the entire language.

A. D. McGettrick, *ALGOL 68—A First and Second Course* Cambridge University Press, 1977.

C. H. Lindsey and S. G. Van Der Meulen, '*Informal introduction to ALGOL 68*, revised edition (North Holland, Amsterdam, 1977).

These are texts for more advanced study of Algol 68.

P. M. Woodward and S. G. Bond, *ALGOL 68-R Users' Guide* (H.M.S.O., 1974). A full introduction to the 68-R dialect of Algol 68.

A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens and R. G. Fisker, 'Revised Report on the Algorithmic Language ALGOL 68' *Acta Inf.*, 5 (1975). This is a formal definition of Algol 68, expressed in metamathematical terms. It is recommended only to advanced readers with a special interest in Algol 68.

PROGRAMMING IN GENERAL

D. Knuth, *The Art of Computer Programming*, vols 1 and 3 (Addison Wesley, New York, 1973).

E. S. Page and L. B. Wilson, *Information Representation and Manipulation in a Computer* (Cambridge University Press, 1973).

O. J. Dahl, E. W. Dijkstra and C. A. R. Hoare, *Structured Programming* (Academic Press, New York, 1972).

Articles by P. J. Denning, P. J. Brown, J. M. Yoke, N. Wirth, D. E. Knuth, B. W. Kernighan and P. J. Plaugher, 'Special Issue: Programming', *ACM comput. Surv.*, 6 (1974).

M. Jackson, *Principles of Program Design* (Academic Press, New York, 1975).

BACKGROUND READING

G. M. Weinberg, *The Psychology of Computer Programming* (Van Nostrand Reinhold, New York, 1971).
Robert Pirsig, *Zen and the Art of Motorcycle Maintenance* (Bodley Head, London, 1974)

Answers to Selected Examples

CHAPTER 1 (p. 10)

1.3 The four ways of dealing are summarised in the following table.

Tariff	Method of Charging	If minimum amount used			If maximum amount used		
		cost	receipts	prof./loss	cost	receipts	prof./loss
1	Flat rate (£5 p.a.)	0	250	+250	500	250	−250
	Metered rate (3p a unit)	100	0	−100	600	750	+150
2	Flat rate (£5 p.a.)	100	250	+150	350	250	−100
	Metered rate (3p a unit)	200	0	−200	450	750	+300

Tariff 1 (metered) or Tariff 2 (flat rate) offer the least risk for the landlord.

1.5 No. You might shake for ever without throwing a 6.

1.6a

x	x^2	7/x
1	1	7
4	16	1.75
2.875	8.266	2.435
2.655	7.049	2.636
2.646	7.001	2.646

so $\sqrt{7}$ to 2 places is 2.65.

1.6b

x	x^2	9/x
1	1	9
5	25	1.8
3.4	11.56	2.647
3.023	9.139	2.977
3.000	9.0	3.000

so $\sqrt{9}$ to 2 places is 3.00.

1.7 What happens is

x	x^2	$-2/x$
1	1	-2
-0.5	0.25	$+4$
$+1.75$	3.063	-1.142
$+0.304$	0.092	-6.579
-3.1375		

and so on. The process does not converge, because there is no number which, when squared, gives -2. To convert the procedure to an algorithm, we should add an instruction at the beginning, saying: "**if** n is negative it has no square root; do not attempt to find it; otherwise:".

CHAPTER 2 (p. 21)

2.2 0 1 2 3 4 5 6 7 8 9 . —

2.3 $\dfrac{5\,000}{800 \times 12} = 521$ ft. A 600-ft tape would suffice.

CHAPTER 3 (p. 40)

3.1

System words	Symbols of constant meaning	Identifiers	Literals
begin	[	*word*	*1*
char	:	*z*	*20*
int	]	*dict*	*10000*
struct	;	*pd*	*1250*
proc	:=	*back*	*80*
if	(	*for*	"*0*"
upb	,	*count*	"*Z*"
then	)	*pdict*	"*A*"
clear	>	*size*	*4*
else	+:=	*tree*	"*ZZZZ*"
or	<	*line*	"*NOTEXT*"
fi	=	*lp*	
end	≠	*tp*	
bool	+	*q*	
while	−	*read*	
do		*newline*	
skip		*print*	
od		*nextword*	
of		*charfetch*	
void		*nodeout*	
not		*x*	
true		*treeprint*	
false		*newnode*	
elif		*make*	
		j	
		k	
		w	
		f	

3.2

```
begin [1:20] char word; int z; [1:10000] char dict; int pd: = 1;
|   ---------
|
|  proc make = void:
|  begin int j, k, w; bool f;
|  |  if not nextword
|  |  |  then print((newline, "NO TEXT"))
|  |  |  else newnode;
|  |  |  while nextword do j: = 1; f: = true;
|  |  |                  |  while f do k: = pdict of tree[j];
|  |  |                  |         |  w: = size of tree[j] + k - 1;
|  |  |                  |         |  if word[1:z] = dict[k:w]
|  |  |                  |         |  |  then f: = false;
|  |  |                  |         |  |       count of tree[j] +: = 1
|  |  |                  |         |  |  elif word[1:z] < dict[k:w]
|  |  |                  |         |  |  then if back of tree[j] ≠ 0
|  |  |                  |         |  |       |  then j: = back of tree
|  |  |                  |         |  |       |  else f: = false;
|  |  |                  |         |  |       |     back of
|  |  |                  |         |  |       |        tree[j]: = tp;
|  |  |                  |         |  |       |     newnode
|  |  |                  |         |  |       fi
|  |  |                  |         |  |  elif for of tree[j] ≠ 0
|  |  |                  |         |  |     then j: = for of tree[j]
|  |  |                  |         |  |  else f: = false;
|  |  |                  |         |  |       for of tree[j]: = tp
|  |  |                  |         |  |       newnode
|  |  |                  |         |  fi
|  |  |                  |         od
|  |  |                  od;
|  |  |  treeprint(1)
|  |  fi
|  end;
|  make
end
```

Lines to show alignment only.

3.6 *pat* is declared twice, in different modes. *susan* is declared twice in the same mode.

3.7

(1) 5	(2) 24	(3) 1	(4) 1	(5) 31
(6) 31	(7) −6	(8) 0	(9) −1	(10) 3
(11) 1	(12) 64	(13) 1		

3.8 (a) **chair** is misspelled
(b) x is declared twice
(c) A bracket is missing in the expression **abs**$(w+3t-47\div w$
(d) $3t$ is not allowed; $3*t$
(e) No semicolon between y: = "Q" and y: = t
(f) You can't assign an **int** to a **char**
(g) t is undefined
(h) The string constant has no closing quotation mark
(i) *print* must not be in bold type.

3.10 This program reads 4 characters and prints them out backwards. In this case it gives DOOM.

3.11

```
begin int a, b, c;
    read (a); read(b);
    print(("A" = ", a, "B = ", b)); c: = a+b;
    print((newline, "A + B = ", c)); c: = a – b;
    print((newline, "A – B = ", c)); c: = a * b;
    print((newline, "A * B = ", c)); c: = a↑b;
    print((newline, "A↑B = ", c))
end
```

This program is followed by the two data numbers a and b.

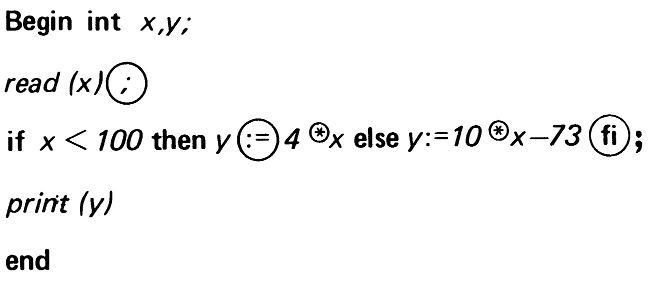
Note: A solution not using a third variable c, but having statements like

print ((*newline*, "$A+B=$ ", $a+b$))

is also acceptable.

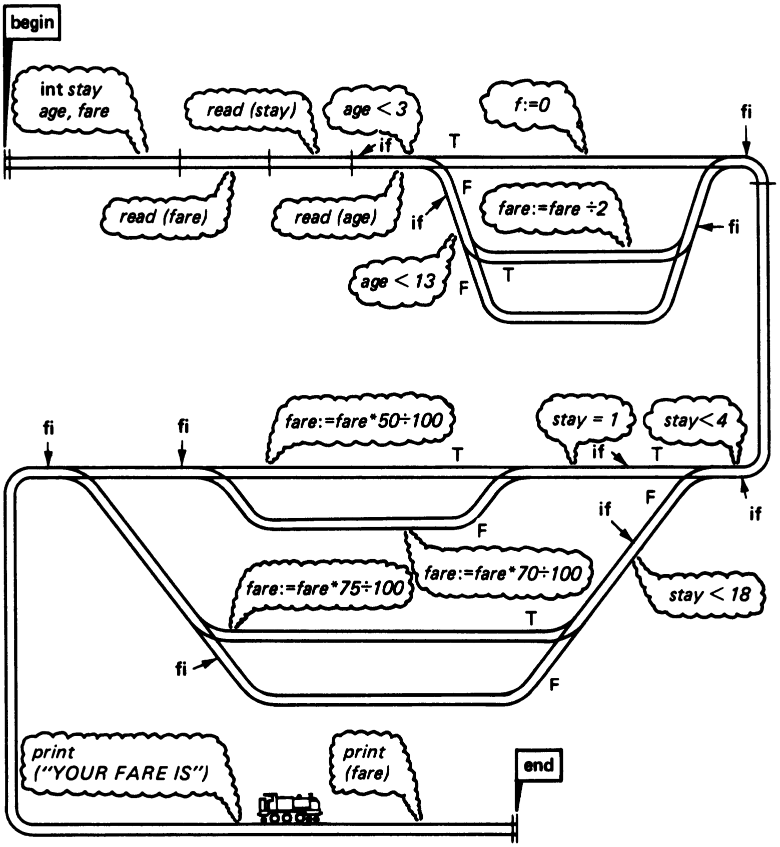
CHAPTER 4 (p. 52)

4.2



```
Begin int x,y;

read (x);

if x < 100 then y:=4 ⊛x else y:=10 ⊛x–73 fi;

print (y)

end
```

4.5



Route Number	*Age*	*Stay*	*Length*
1	< 3	1	11
2	<3	2–3	11
3	< 3	4–16	12
4	< 3	> = 17	11
5	3–13	1	12
6	3–13	2–3	12
7	3–13	4–16	13
8	3–13	> = 17	12
9	> 13	1	11
10	> 13	2–3	11
11	> 13	4–16	12
12	> 13	> = 17	11

4.6

```
begin int a, b, c;
  read((a, b, c));
    print(("THE THREE NUMBERS ARE", a, b, c, newline));
  if a > = b+c or b > = a + c or c > = a+b
    then print(("THEY DO NOT FORM A TRIANGLE"))
    else print(("THEY FORM A TRIANGLE"))
  fi
end
```

CHAPTER 5 (p. 68)

5.1

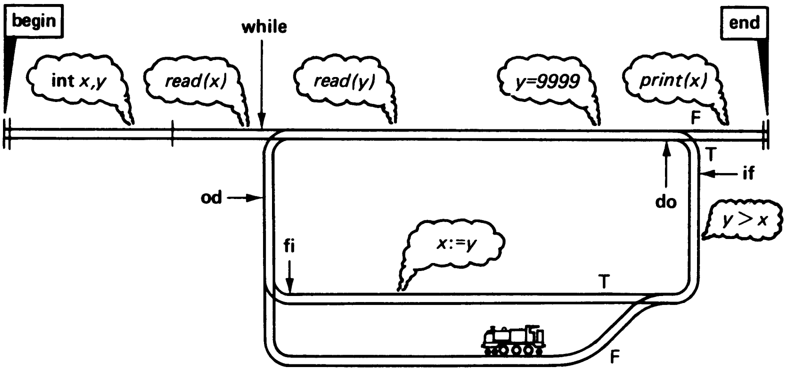
~~45~~ ~~77~~ ~~32~~ ~~123~~ ~~44~~ ~~84~~ ~~39~~ ~~2~~ ~~9999~~ (input)

123 (output)

x	y
~~45~~	~~77~~
~~77~~	~~32~~
123	~~123~~
	~~44~~
	~~84~~
	~~39~~
	~~2~~
	9999

This program reads a stream of numbers terminated by 9999 and prints the largest one (not including the terminal 9999).

5.2



5.4

```
5
```

input

```
1   1
2   3
3   6
4  10
5  15
```

output

a	b	s	t
~~1~~	~~1~~	~~0~~	5
~~2~~	~~2~~	~~1~~	
~~3~~	~~1~~	~~0~~	
~~4~~	~~2~~	~~1~~	
~~5~~	~~3~~	~~3~~	
6	~~1~~	~~0~~	
	~~2~~	~~1~~	
	~~3~~	~~3~~	
	~~4~~	~~6~~	
	~~1~~	~~0~~	
	~~2~~	~~1~~	
	~~3~~	~~3~~	
	~~4~~	~~6~~	
	~~5~~	~~10~~	
	~~1~~	~~0~~	
	~~2~~	~~1~~	
	~~3~~	~~3~~	
	~~4~~	~~6~~	
	~~5~~	~~10~~	
	6	15	

5.5 Frequencies derived from traces with different values of t.

	t					
	1	2	3	4	5	
begin int *a, b, s, t*;	1	1	1	1	1	α
read(*t*);	1	1	1	1	1	α
a:= 1;	1	1	1	1	1	α
while *a* < = *t*	2	3	4	5	6	β
do *s*: = *0*;	1	2	3	4	5	γ
b: = *1*;	1	2	3	4	5	γ
while *b* < = *a*	2	5	9	14	20	δ
do *s*+: = *b*	1	3	6	10	15	ε
b+: = *1*	1	3	6	10	15	ε
od;						
print(*a*);	1	2	3	4	5	γ
print(*s*);	1	2	3	4	5	γ
print(*newline*);	1	2	3	4	5	γ
a+: = *1*	1	2	3	4	5	γ
od						
end						

In general, "*s*+: = *b*" is executed $\frac{1}{2}t(t+1)$ times, so that when t = 100, the number of executions is 5050.

In general,

instructions in group α are executed 1 times
instructions in group β are executed $(t+1)$ times
instructions in group γ are executed t times
instructions in group δ are executed $\frac{1}{2}(t+1)(t+2)-1$ times
instructions in group ε are executed $\frac{1}{2}t(t+1)$ times

$$\begin{aligned}\text{Total for } t = 100 &= 3+(t+1)+6t+\tfrac{1}{2}(t+1)(t+2)-1+2\times\tfrac{1}{2}(t(t+1))\\ &= 3+101+600+5150+10100\\ &= 15954 \text{ instructions}\end{aligned}$$

5.7c

```
begin int p, q, a, b;
  print(("        ")); q: = 100;
  while q < 110 do print (q); q+ : = 1 od;
  p: = 1000;
  while p < 1100
    do print ((newline, p)); q: = 100;
      while q < 110
```

```
            do a: = p; b: = q;
               while a * b ≠ 0
                  do if a > b then a: = a mod b
                                else b: = b mod a
                     fi
                  od;
               if a = 0 then print(b) else print(a) fi;
               q + : = 1
            od;
          p + : = 1
       od
  end
```

CHAPTER 6 (p. 81)

6.1 This program only reports whether the last data number is greater than 50. This can be determined by tracing.

6.2 *j*: = *x* = *17*

6.3

a	*b*	*c*	**not**(*a* **or** *b* **and** *c*) **and** (*a* **and** *b* **or not** *c*)
t	t	t	f
t	t	f	f
t	f	t	f
t	f	f	f
f	t	t	f
f	t	f	t
f	f	t	f
f	f	f	t

6.4

```
      not(a or c)
(or)  not a and not c
```

CHAPTER 7 (p. 95)

7.1 a, c, d and f are in short form. Expanded, they become

```
ref int k = loc int
ref bool x = loc bool: = true
ref char q = loc char: = "X"
ref bool seven = loc bool
```

7.2

identifier	*mode*	*reach (lines)*
n	**int**	2–15
p	**int**	3–14
q	**int**	4–14
r	**int**	5–14

7.4 The program prints any number below 1000 whose value is equal to the sum of the cubes of its digits.

7.7

```
(a)  +3   +5   +7    +9
(b)  +1   +1   +1    +1
(c)  −1   +1   +3    +5    +7    +9    +11
(d)  +1   +8   +15   +22   +29
(e)  +2   +5   +5    +8    +10   +13
(f)  +1   +2   +3    +4    +5    +45
```

7.8 1024 = 16 x 64, so this program will be organised as two nested loops. The inner one will run from 1 to 16, and the outer one from 0 to 1008 in steps of 16.

```
begin int c, x;
  for a from 0 by 16 to 1008
  do print(newline);
    for b to 16
    do c: = a + b;
      for d to 3
      do x: = c ÷ 256; c: = 16 * (c mod 256);
           if x < = 9 then print(repr(abs "0" + x))
                      else print(repr(abs "A" + x − 10))
           fi
      od;
    print("  ")
    od
  od
end
```

CHAPTER 8 (p. 110)

```
8.1 ref [ ] int quirk = loc [1:25] int
    ref [ ] bool able = loc [1:1000] bool
```

8.2 *t* > = *1700* **and** *t* < = *2435* **and** (*t* – *1700*) **mod** *15* = *0*

8.4 Since the large majority of requests are for the same small set of items, it would pay to keep them in decreasing order of query frequency, and to search from the top of the list downwards.

8.6

```
8.6 begin [1:100] int x; int n: = 1;
       while read(x[n]); x[n] ≠ 0 do n +: = 1 od;
       bool b: = true;
        while b
          do b: = false;
            for p to n−2
              do if x[p + 1] < x[p]
                   then int d= x[p]; x[p]: = x[p+1]; x[p+1]; = d;
                          b: = true
                 fi
              od
          od;
       for p to n – 1 do print((newline, x[p])) od
    end
```

CHAPTER 9 (p. 126)

9.3

```
    [1:8, 1:8] bool chess;
      for j to 8 do for k to 8 do chess [j, k]: = (j+k)mod 2 = 0 od od
```

9.8

```
9.8 begin co we use two arrays, one for the current state, and one for the next
generation

co
    [1:15, 1:15] char now; next;
             co the following sets up the initial picture co
             for j to 15 do for k to 15 do now[j, k]: = " " od od;
             for j from 6 to 10 do for k from 7 to 9 do
               now[j, k]: = "*" od od;
             for c to 25 co set up 25 cyles co
                do print((newline, newline));
                   for j to 15 co print current state co
                      do print (newline);
                         for k to 15 do print(now[j, k])od
                      od;
                   for j from 2 to 14 co now work out next generation co
```

```
                    do for k from 2 to 14
                       do int n: = −abs(now[j, k] = "*");
                          for x from j−1 to j+1
                             do for y from k−1 to k+1
                                do n+: = abs(now[x, y] = "*")
                                od
                             od;
                          next[j, k]: = "  ";
                          if(n = 2 and now[j, k] = "*")or n = 3
                            then next [j, k]: = "*"
                          fi
                    od
                 od;
                 for j from 2 to 14
                 co move next generation to current state co
                 do for k from 2 to 14 do now[j, k]: = next[j, k] od
                 od
         od
   end
```

CHAPTER 10 (p. 146)

10.6 The procedure tries to change the values of two constants *a* and *b*. A better version of the procedure is

```
   proc hcf = (int a, b) int:
      begin int p: = a, q: = b;
         while p ≠ q do (p > q | p : = p−q | q: = q−b) od;
        p
      end
```

10.7

```
   proc letter = (char x) bool:
      (x < = "Z" and x > = "A")
```

10.8

```
    proc fineprint = (int n, s; bool sign) void:
       begin int x: = abs n, ac: = 1; bool plus = n > 0;
          co x starts with absolute value of number to be printed
             ac will hold the true number of digits required (max. 7)
             plus is the sign of the number to be printed
          co
          [ ] int power = (10, 100, 1000, 10000, 100000, 1000000);
```

```
        co 0 is a special case co
        if n = 0
          then to s − 1 do print("   ") od; print("0")
          else
            while x > powers[ac] do ac +:= 1 od;
            if ac + 1 > s
              then to s do print("*") od
              else to s − (ac + 1) do print("   ") od;
                   if sign then print((plus | "+" | "−"))
                           else print((plus | "   " | "−"))
                   fi;
                   while ac > 1
                      do print(repr(abs "0" + x ÷ powers[ac]));
                         x := x mod powers [ac]
                      od;
                   print (repr (abs "0" + x))
            fi
        fi
      end
```

CHAPTER 11 (p. 161)

11.1

```
proc binout = (int b) void:
   ((b > 0 | binout (b ÷ 2); print(repr(abs "0" + b mod 2))))
```

11.2

−1 +2 +3 +4 +7 +12

procedure *bubble* is given a reference to an array of integers as its parameter. It sorts the array into ascending order.

11.3 +1 +5 +55

11.6

```
proc ssq = (int q) int:
   begin int s := 0;
         for t to q do s +:= t↑2 od;
         s
   end
```

11.7 T F T T F T

11.8

```
begin int c;
   proc a = (int m, n) int:
      begin
         c+:= 1; co counts entries co
         if m = 0 then n+1
         elif n = 0 then a(m-1, 1)
         else a(m-1, a(m, n-1))
         fi
      end;
   print((“  M   N   A(M,N)   NO OF CALLS”));
   for j to 3
      do for k to 3
         do c:= 0;
            print((newline, j, k, a(j, k), c))
         od
      od
end
```

CHAPTER 12 (p. 182)

12.3 Either of the following is acceptable

(a)

```
begin real e, s, c, t;
   print((newline, “X SINH(X) COSH(X) TANH(X)”));
   for j from 0 to 100
      do e: = exp (0.1*j);
         s: = 0.5*(e-1.0/e);
         c: = 0.5*(e+1.0/e);
         t: = s/c;
         print((newline, 0.1*j, s, c, t))
      od
end
```

(b)

```
begin
   proc sinh = (real x) real:(0.5*(exp(x)-exp(-x)));
   proc cosh = (real x) real:(0.5*(exp(x)+exp(-x)));
   proc tanh = (real x) real:(sinh(x)/cosh(x));
   for j from 0 to 100
     do real x = 0.1*j;
        print((newline, x, sinh(x), cosh(x), tanh(x)))
     od
end
```

12.5c

```
begin
    proc squarewave = (real x, int n) real:
      begin real s: = 0.0;
          for j by 2 to 2*n-1
              do s+: = sin(j*x)/j od;
          s
      end;
    [0:100] char line;
    for q from 0 to 5
      do print((newline, "GRAPH FOR Q = ", q));
         for j from 0 by 10 to 720
            do real x = j*pi/180;
               int s = entier (30*squarewave(x, q+1));
               for k from 0 to 100 do line[k]: = "  " od;
               line[s]: = "*"
               print((newline, j, line))
            od
      od
end
```

CHAPTER 13 (p. 196)

13.1

(a) +8; (b) +49

(c) +7.0000000$_{10}$0 +7 +7

(d) NO

13.3

```
proc censor = (ref[ ]char s) void:
   begin proc scan = (ref[ ] char t) void:
     begin co this 'internal' procedure searches the array s for an
              occurrence of the sequence t. If found, it replaces the
              occurrence by stars.
           co
           int ss = lwb s, sf = upb s, ts = lwb t, tf = upb t;
           for j from ss to sf-(tf-ts)
              do if s[j] = t[ts]
                    then int k: = ts+1;
                     while s[j+k-ts] = t[k] and k < = tf
                          do k+: = 1 od;
                     if k > tf
                       then for k from ts to tf do s[j+k-ts]: = "*" od
                     fi
```

```
                fi
            od
    end;
    scan ("CONSERVATIVE");
    scan ("LABOUR");
    scan ("LIBERAL");
    scan ("NATIONALIST")
end
```

13.4

```
begin
  co here follows the texts of procedures alpha, digit, numin, cardate and
    compare as given on pp. 197 and 198.
  co
  proc carmerge = (ref[,] char a, b) [,] char:
    co merges two arrays containing car numbers co
    (int la = 1 upb a, lb: = 1 upb b;
     int pa: = 1, pb: = 1, [1:la+lb, 1:7] char c;
     for pc to la+lb
       do (pa > la | c[pc,]: = b[pb,]; pb+: = 1
          |: pb > lb | c[pc,]: = a[pa,]; pa+: = 1
          |: compare (b[pb,], a[pa,] | c[pc,]: = b[pb,]; pb+: = 1
                                     | c[pc,]: = a[pa,]; pa+: = 1
          )
       od;
    c
   );
  proc carsort = (ref[,] char a) void:
     co sort the list of cars a. Assumes that lwb a = 1 co
     (int q = 1 upb a; int k = q÷2;
      if q > 1
        then carsort (a[1:k,]); carsort(a[k+1:q,]);
             a: = carmerge(a[1:k,], a[k+1:q,])
      fi
     );
  [1:3000, 1:7]char cars; int n: = 0; [1:7]char next;
  while next: = numin; next ≠ "ZZZ999Z do n+: = 1; cars[n,]: = next
    od;
  carsort(cars[1:n,]);
  print((newline, "SORTED CAR REGISTRATIONS ARE",
    newline, newline));
  int lc: = 0;
  for j to n
    do print((cars[j,], "   ")); lc+: = 1;
```

```
          (lc = 10)|print(newline); lc: = 0)
        od
    end
```

CHAPTER 14 (p. 210)

14.1a

```
    mode student = struct([1:24]char name, [1:12]char dept,
            int eng, maths, cs, phil);
            [1:100] student file;

(b) proc aggregate = (student x) int:
      co calculates the aggregate marks of a student co
      (eng of x+maths of x+cs of x+phil of x);
    int max: = aggregate(file[1]), where: = 1;
      for j from 2 to 100
        do int score = aggregate(file[j]);
          if score > max then max: = score; where: = j fi
        od;
    print((newline, "STUDENT WITH HIGHEST AGGREGATE IS",
            name of file[where]
          ))
```

14.2

```
    mode point = struct(real ax, ay);

    mode line = struct (real m, c);

    proc distance = (point p, q)real:
     co uses Pythagoras' theorem to find distance between two points co
     (sqrt((ax of p−ax of q)↑2+(ay of p−ay of q)↑2));
     proc join = (point p, q)line:
     co fits a line to the two points p and q according to the formula

          $\frac{y-y_1}{x-x_1}=\frac{y_2-y_1}{x_2-x_1}$

     co
     (real slope = (ay of q − ay of p)/(ax of q − ax of p);
      real intercept = ay of p − slope * ax of p;
      (slope, intercept)
     );

    proc crosses = (line p, q) point:
```

```
  co calculates the point where lines p and q cross co
  (real x = (c of q - c of p)/(m of p - m of q);
   real y = x * m of p + c of p
   (x, y)
  )
begin
  mode ship = struct( [1:20] char name, real xpos, ypos, course, speed);
  proc fleetin = (ref[ ]ship fleet)int:
    co this procedure reads in details of the fleet at midnight and stores
       them in the formal parameter fleet. It yields the number of ships
       present
    co
    begin ship next; int n: = 0;
      while read(next); name of next ≠
        "ZZZZZZZZZZZZZZZZZZZZ"
        do fleet[n +: = 1]: = next; read(newline) od;
      n
    end;
  proc when = (ship s, real t, posx, posy) real:
    co the time now is t. An accident has just happened at (posx, posy).
       The procedure works out how soon ship s can be on the scene,
       assuming it goes there directly from its present position
    co
    begin co first we work out present position co
      real ppx = xpos of s + speed of s * sin(course of s * pi/180),
           ppy = ypos of s + speed of s * cos(course of s * pi/180);
      co next we calculate the distance to be covered to the disaster
      co
      real dist = sqrt((ppy-posy)↑2 + (ppx-posx)↑2)
      co finally the time in hours co
      dist/speed of s
    end;
    co here begins main program co
    [1: 100]ship fleet; int n = fleetin;
    real swx, swy, swt; co details of disaster co
    read((swx, swy, swt));
    real bt: = when(fleet[1], swt, swx, swy); int which: = 1;
    for j from 2 to n co now choose best ship co
      do real tt = when(fleet[j], swt, swx, swy);
        if tt < bt then bt: = tt; which: = j fi
      od;
    co now extract details of best ship co
    print((newline, "THE SHIPWRECK HAPPENED AT",
      swt, "HOURS", newline,
         "ITS COORDINATES WERE", swx, swy));
    print((newline, “THE BEST SHIP TO GO TO THE
       RESCUE IS”,
          ((name of fleet[which], newline,
               “WHICH IS PRESENTLY AT”,
             xpos of fleet[which], ypos of fleet[which]));
    real course: = 180 * arctan((swx − xpos of fleet[which]/
                                   swy < ypos of fleet[which]));
    co correct in case this points directly away! co
    (swy < y pos of fleet[which] | course + : = 180);
    (course < 0 | course + : = 360);
    print((newline, “THE CORRECT COURSE IS”, course,
       “DEGREES E OF N”));
    print((, newline, “ESTIMATED ARRIVAL TIME IS”,
         swt + sqrt((swx-xpos of fleet[which]) ↑ 2
         + (swy-ypos of fleet[which]) ↑ 2)
               /speed of fleet[which], “HOURS AFTER MIDNIGHT”
         ))
end
```

Index

Page numbers in italic are pointers to extensive discussions of the items involved. Certain key concepts, such as ints chars expressions and procedures are used so frequently that to give a reference to every occurrence would not be practicable.