# COMPUTER PROGRAMMING AND AUTOCODES

# COMPUTER PROGRAMMING AND AUTOCODES

D. G. BURNETT-HALL, M.A.

*Lecturer-in-charge, Computation Laboratory,*
*University of Hull*

L. A. G. DRESEL, M.A., Ph.D.

*Director, Computer Unit,*
*University of Reading*

P. A. SAMET, B.Sc., Ph.D.

*Director, Computation Laboratory,*
*University of Southampton*

THE ENGLISH UNIVERSITIES PRESS LTD
102 NEWGATE STREET · LONDON E.C.1

First printed 1964

*Printed in Great Britain for the English Universities Press Ltd.
by C. Tinling & Co. Ltd., Liverpool, London and Prescot*

# PREFACE

This book is intended to serve as an introduction to the programming of automatic computers. In our experience beginners usually find most difficulty in the organizational aspects of a calculation, rather than in choosing the correct arithmetical operations. For lack of time, courses on programming given by manufacturers, or at universities and technical colleges, tend to concentrate on the coding and conventions of their particular machine, and the beginner is left to pick up general ideas for himself. Similarly, many books that aim at giving an insight into general principles postulate a hypothetical computer and then proceed to give a coding course for this "machine".

We have tried to overcome these difficulties by dividing this book into three parts. In Part I, we discuss the planning of a calculation in general terms with no particular machine or programming system in mind. This is done entirely by means of examples, and we make no apology for deliberately using examples that are mathematically very simple. For many people, all contact with a computer is through a simplified programming system or "autocode", and although the task of coding a problem is then much simpler than with machine code, it is often forgotten that the initial planning needs just as much care and thought. In Part II we describe in some detail three of the autocodes in common use in Great Britain at the present time. We have kept to the autocodes with which we are most familiar, and these are also typical of the present state of development. Finally, in Part III, we give an account of the international programming language ALGOL. Wherever possible in Parts II and III we have used the same examples as in Part I. The chapters of Part II, as also the chapter on ALGOL, are entirely independent of each other and need not be read consecutively.

It should be stressed that the descriptions of Part II, although detailed, are not intended to be complete working manuals. It is essential that anyone intending to *use* a computer should consult the staff of a computing laboratory *before* spending time on writing a programme. Operating systems are not always the same, new facilities may be added, or a programme for the particular calculation may already be available for general use.

We are grateful in particular to: Elliott Bros. Ltd. and Ferranti Ltd., for permission to publish information about the autocodes for their machines; Mr. M. Woodger, National Physical Laboratory, for a most helpful discussion regarding the chapter on Algol; Mrs. H. W. Chamberlain and Mrs. A. M. Dunmur, who helped with the testing of Algol programmes; Mrs. M. Jalový and Mrs. J. Chapman, who produced a typescript from almost illegible notes. Finally, it is pleasant to acknowledge the helpfulness and efficiency of the staff of the publishers and printers.

D.G.B.-H.
L.A.G.D.
P.A.S.

Southampton.
January, 1963.

# CONTENTS

# PART I.
# PROGRAMMING

CHAPTER 1

# AUTOMATIC COMPUTERS

An automatic computer is a machine for doing arithmetic very rapidly and without intervention by an operator. The speed of modern computers is enormously high and brings within reach the solution of problems that would otherwise have occupied scientists for very long periods, possibly months or even years; certainly speed is one of the most striking features of the machines and often the most publicized. Just as important as the speed, however, and very intimately connected with it, is the fact that a computer will work automatically once it has been correctly set up to perform a certain job. This "setting up" procedure, which involves careful planning and investigation of a complete calculation, is called *programming*. It is the aim of this book to give an introduction to this subject.

Before going into any details of how to use a computer, we must describe some of the internal organization of the machine and how this affects what we can do with it. Everything in this chapter applies basically to all computers currently available.

We shall start from the observation that a computer works automatically and rapidly, and see what this implies.

The first deduction is that *any* fast machine must of necessity be automatic.* The interval between successive operations must be small, otherwise the benefit of the speed of these operations is lost. It is therefore not possible to have manual step-by-step control of a fast machine. It follows that the sequence of operations must be stored inside the machine in some manner. This implies a *store* or *memory unit* to hold the instructions and also a *control unit* to select and execute these instructions in the required sequence.

The primary purpose of employing a computing machine is to work with numbers. Generally we shall require several numbers during a calculation. It therefore becomes necessary to hold numbers as well as instructions in the store, and in most computers the store used for numbers is physically the same as that used for instructions. The size of the store depends on many factors, not least the money available to buy a machine. Faster machines need larger stores so that larger problems can be tackled. It is generally reckoned that the store must be able to hold at least some thousands of numbers or instructions.

We shall need *input* and *output* devices for our machine, so that information can be passed from the outside world into the store of the computer and vice versa. Curiously enough, even in this electronic age, the cheapest methods of rapid input and output involve holes in paper or cardboard. It is possible to convert patterns of holes in paper tape or cards into electrical pulses for input and to punch such holes from electrical pulses for output. The tape or cards can be prepared on devices rather like typewriters and various printers can be operated by punched tape or cards.

It goes almost without saying that there must be an *arithmetic unit* which will perform the arithmetic operations specified under direction from the control unit. The basic arithmetical repertoire of a computer is very small. Complicated operations, such as finding a square root or logarithm, have to be made up from suitable combinations of simpler steps. It is interesting to notice that with a small number of basic operations, combined in a suitable sequence, we can do the most complex calculations. This is not all that surprising: an alphabet of 26 letters together

* The converse is not true. An automatic machine need not be fast.

with some extra symbols enables us to write English, and most other European languages as well. The sequence of instructions required for a complete calculation is called a *programme* or *program* (which is not an exclusively American term).

Some, at least, of the machine's operations must be administrative, not arithmetical, in nature. In particular, there must be a facility for choosing one of two (or more) possible instructions as the next to be obeyed. There are many occasions when the course of action required depends on a number that has been calculated. For example, the quadratic equation $ax^2+bx+c = 0$ has real or complex roots depending on whether

$$b^2-4ac\geqslant 0 \quad \text{or} \quad b^2-4ac<0,$$

and we have to do different operations in the two cases. Such situations occur very frequently.

Since the most easily understood way of writing a programme is to give the instructions in their natural sequence it has become common practice to call such a test instruction a *conditional jump*. The usual convention is that the jump occurs if the condition is satisfied, otherwise the programme continues with the next instruction in the sequence. The usual criteria for conditional jumps include the following: a number being equal to zero, a number being not equal to zero, a number being positive, equality between two numbers, one number being greater than another.

The conditional jump is also important for another reason. Calculations tend to be repetitive and it is often necessary to perform the same sequence of operations many times. By introducing a *counter* to count the number of times we have obeyed the sequence of instructions, together with an instruction to test the counter, we can have a *loop* of instructions which is repeated until the counter has reached a desired value. A loop gives us a programme with fewer instructions, thereby saving effort and also storage space, an important consideration as the store is of limited capacity.

The choice implied by a conditional jump instruction is usually one of two alternatives. It is important to realize that a choice of many alternatives can be reduced to a series of such simple choices. For example, to determine the suit of a playing card we could ask the questions:

$$\begin{tikzcd}
 & & & \text{Is the card red?} \arrow[dll, "\text{Yes}"'] \arrow[drr, "\text{No}"] & & & \\
 & \text{Is the suit Hearts?} \arrow[dl, "\text{Yes}"'] \arrow[dr, "\text{No}"] & & & & \text{Is the suit Spades?} \arrow[dl, "\text{Yes}"'] \arrow[dr, "\text{No}"] & \\
\text{Hearts} & & \text{Diamonds} & & \text{Spades} & & \text{Clubs}
\end{tikzcd}$$

This, incidentally, is the basis of the popular game of "Twenty Questions" when only "Yes" and "No" are allowed as answers: twenty such answers cover 1048576 alternatives. The same technique also provides a highly efficient method of searching.

It is sometimes required to go to a particular instruction as the next one to be obeyed, irrespective of any condition. Although this could be achieved by a conditional jump in which the condition is automatically satisfied, it is more usual to include a special instruction, called an *unconditional jump*.

To sum up, we have shown that to be fully automatic a computer must have

i. a store for instructions
ii. a store for numbers
iii. a control unit

iv. an arithmetic unit, usually capable of addition, subtraction, multiplication and division.
v. input and output devices
vi. conditional (and unconditional) jump instructions.

The way that the various units are connected is shown schematically in Figure 1.1.

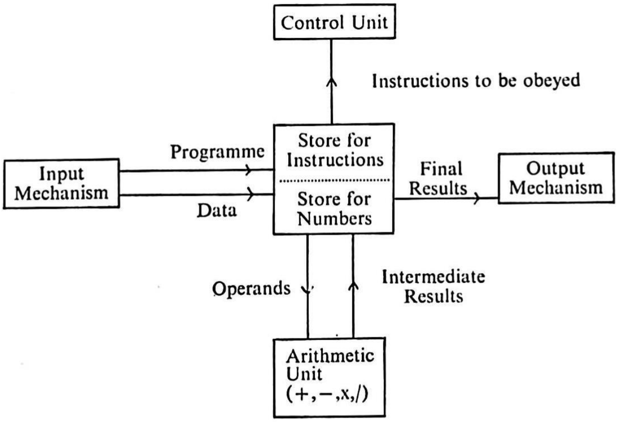


Fig. 1.1. Schematic Diagram of a Computer. (Control signals are not shown.)

Computers vary enormously in size and speed. However, the features given above are common to all computers and are the only essential ones.

CHAPTER 2

# LOOPS AND COUNTERS

Because a computer is automatic it is necessary to plan the whole course of the calculation beforehand. This planning is generally known as *programming*. As in the planning of any operation we distinguish two quite distinct parts, the strategy and the tactics. The strategy depends on the chosen method of solution and has nothing to do with the particular computer that happens to be at hand. The tactics used to achieve the solution, on the other hand, depend on the facilities available and should be chosen to exploit these to the best advantage. It is common to call both parts of this planning programming, but this is unfortunate. In this book we shall use the word *programming* for the strategic planning, reserving the term *coding* for the tactical part of the exercise. This chapter and the next are concerned with programming.

It should be obvious that the first thing to be done when programming a calculation is to write out, in some detail, what steps are necessary in the solution of the problem. Curiously enough, many programmers do not regard this as obvious and will start by writing the instructions the computer will have to perform. For small programmes, written by experienced programmers, this may be permissible—experience helps one to convert the chosen strategy directly into the instructions required. Unfortunately it is only too easy to make mistakes in writing programmes, and such errors are found more quickly if the whole plan of the programme is available. The principal lesson to be learnt is that mistakes in programming occur mostly in the organization of a calculation. Such mistakes can often be avoided by presenting the plan of the programme in a form that is easily checked.

To illustrate this point we shall now consider a simple problem, namely finding the least prime factor (other than 1) of an integer $n$. One possible approach would be to divide $n$ by prime numbers in turn until a factor is found. This is not really practical because tables of primes are necessarily limited and would take too much storage space even for moderate values of $n$*. So we look for a method that is applicable to all integer values of $n$.

With the exception of the number 2, all primes are odd numbers. Division by odd numbers in turn therefore will include division by all odd primes (and by a lot of other numbers as well) and so will enable us to find prime factors. There is the possibility that $n$ itself is an odd prime. We could continue dividing by odd numbers until we are dividing by $n$ itself. However, if we have found no factors by the time we reach $\sqrt{n}$ then $n$ must be a prime. We can even avoid calculating $\sqrt{n}$ by noting that we have just passed $\sqrt{n}$ when the quotient becomes less than the divisor.† Schematically we can write the whole process in the form of a *flow diagram*, where each calculating step is enclosed in a box. Steps requiring decisions are enclosed in boxes with rounded ends.‡ The flow diagram of Fig. 2.1 illustrates our procedure. We denote a candidate for a factor by $m$.

It is not difficult to generalize the process to give *all* prime factors of $n$. This is done in Fig. 2.2. Basically, our process is still to find the least factor of a number. Whenever a factor $p$ has been found we replace $n$ by the quotient $n/p$. We then search for a prime factor of this quotient starting with $p$ as first candidate, since there can be no smaller factor.

* For example, the thousandth prime is only 7907.
† This uses the fact that if $n = pq$, one factor exceeds $\sqrt{n}$ and the other is less than $\sqrt{n}$.
‡ We shall use this convention throughout the book.

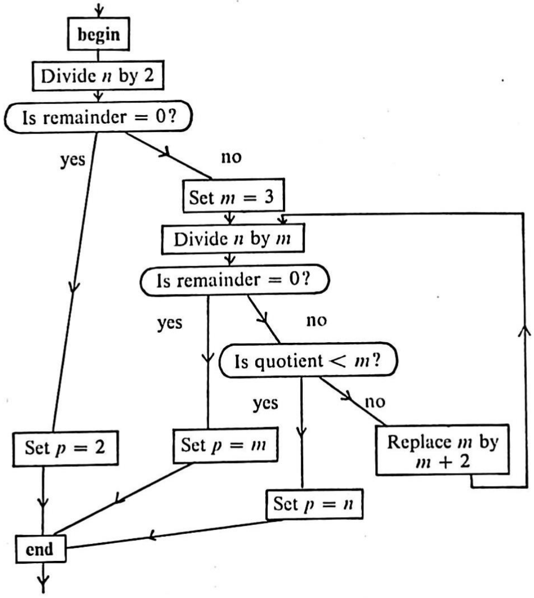


Fig. 2.1. Flow diagram for calculating $p$, the least prime factor of $n$.

*Problem for the reader.* We use non-prime odd numbers as possible factors, yet the process gives the least *prime* factor. Why?

Let us now generalize the problem further still, and suppose that we are asked to produce a table of prime factors for all numbers $n$ in the range $1 \leqslant n \leqslant 1000$. In the flow diagram for this calculation, shown in Fig. 2.3, we no longer have to write out the complete flow diagram for factorizing a particular integer. The box called

*factorize n*

stands for the complete programme of Fig. 2.2. However, *any* correct programme for factorizing an integer may be substituted instead.

Normally we shall start and finish every programme by the boxes

$\boxed{\textbf{begin}}$ and $\boxed{\textbf{end}}$

sometimes adding a comment about the programme (as in Fig. 2.2). A programme that starts with **begin** and finishes with **end** we shall call a *block*. We can always use a block as a part of a larger programme (as in Fig. 2.3), provided that no clashes occur between the names of operands.

We meet the phrases "Replace . . . by . . ." and "Set . . . = . . ." (and really they are the same)

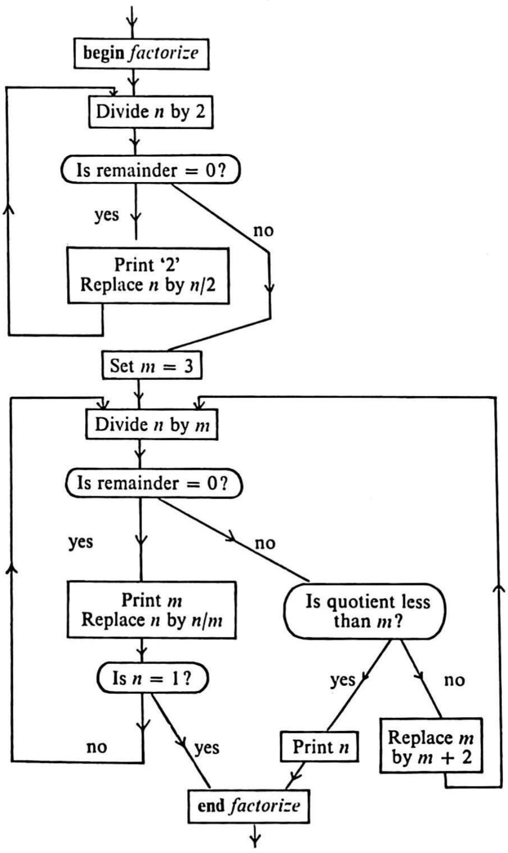


Fig. 2.2. Flow diagram for factorizing an integer.

*Problem for the reader.* Should one take any precautions against (i) $n = 0$, (ii) $n < 0$?

so often that it is convenient to have a common notation for them. We shall use the symbol := to mean "becomes equal to" (or "is replaced by" or "is set equal to").

The three programmes illustrated so far show one of the commonest features of all computer programmes, the repeated cycle of instructions, or *loop*. In Figs. 2.1 and 2.2 we have a loop that is obeyed until a particular criterion is satisfied and it is not known beforehand how many cycles will be necessary. The last example shows a loop where we know in advance how many times we have to go round and we count until we have done the process the right number of times.

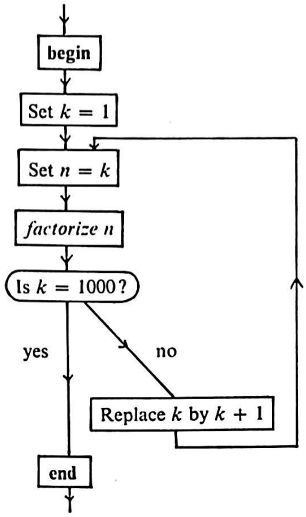


Fig. 2.3. Flow diagram for factorizing all integers between 1 and 1000.

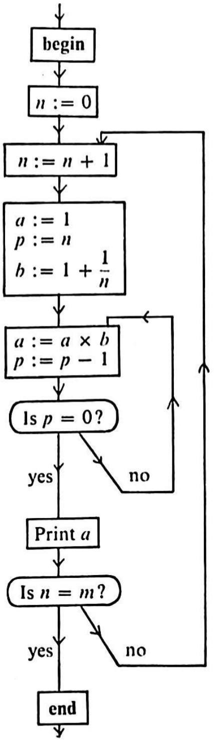


Fig. 2.4. Procedure for calculating $\left(1+\frac{1}{n}\right)^n$.

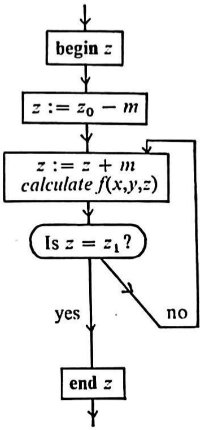


Fig. 2.5. Evaluation of $f(x,y,z)$ for varying values of $z$.

*Problem for the reader.* Our table is for $n$ in the range $1 \leqslant n \leqslant 1000$. Why is it necessary to introduce $k$ in Fig. 2.3?

Often we have a programme where there is a loop inside another loop. To illustrate this, we consider the evaluation of

$$\left(1+\frac{1}{n}\right)^n$$

for $n = 1,2,3,\ldots,m$, without using logarithms. A possible procedure for this calculation is shown in Fig. 2.4. Note that the number of times we go round the inner loop on any occasion is equal to the number of times we have been round the outer loop.

This programme illustrates, incidentally, some good points of technique. The new value of $n$ is

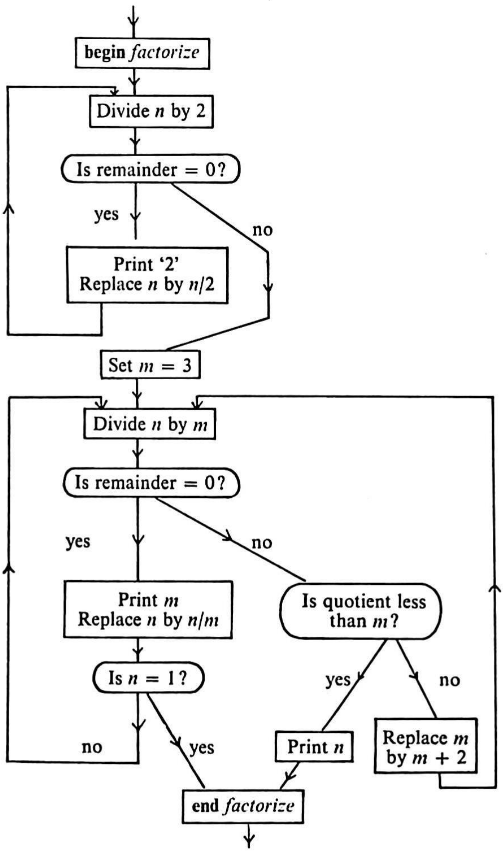


Fig. 2.2. Flow diagram for factorizing an integer.

*Problem for the reader.* Should one take any precautions against (i) $n = 0$, (ii) $n<0$?

so often that it is convenient to have a common notation for them. We shall use the symbol := to mean "becomes equal to" (or "is replaced by" or "is set equal to").

The three programmes illustrated so far show one of the commonest features of all computer programmes, the repeated cycle of instructions, or *loop*. In Figs. 2.1 and 2.2 we have a loop that is obeyed until a particular criterion is satisfied and it is not known beforehand how many cycles will be necessary. The last example shows a loop where we know in advance how many times we have to go round and we count until we have done the process the right number of times.

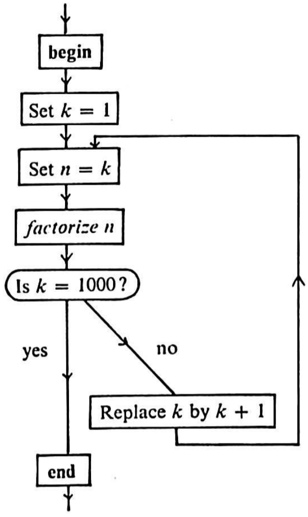


Fig. 2.3. Flow diagram for factorizing all integers between 1 and 1000.

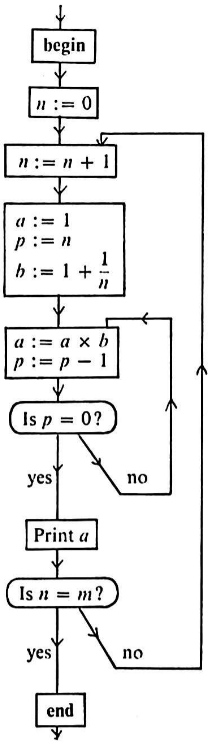


Fig. 2.4. Procedure for calculating $\left(1+\frac{1}{n}\right)^n$.

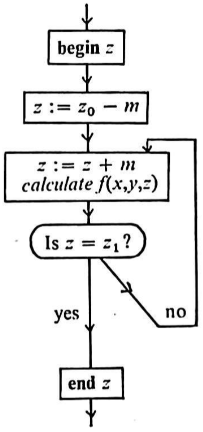


Fig. 2.5. Evaluation of $f(x,y,z)$ for varying values of $z$.

*Problem for the reader.* Our table is for $n$ in the range $1 \leqslant n \leqslant 1000$. Why is it necessary to introduce $k$ in Fig. 2.3?

Often we have a programme where there is a loop inside another loop. To illustrate this, we consider the evaluation of

$$\left(1+\frac{1}{n}\right)^n$$

for $n = 1,2,3,\ldots,m$, without using logarithms. A possible procedure for this calculation is shown in Fig. 2.4. Note that the number of times we go round the inner loop on any occasion is equal to the number of times we have been round the outer loop.

This programme illustrates, incidentally, some good points of technique. The new value of $n$ is

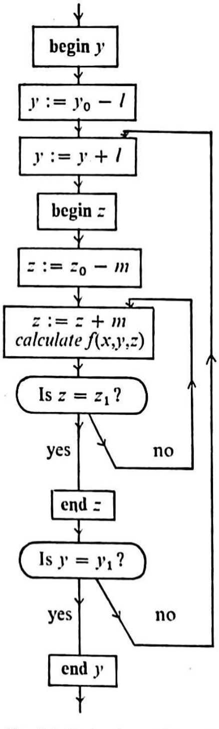


Fig. 2.6. Evaluation of $f(x,y,z)$ for varying values of $y$ and $z$.

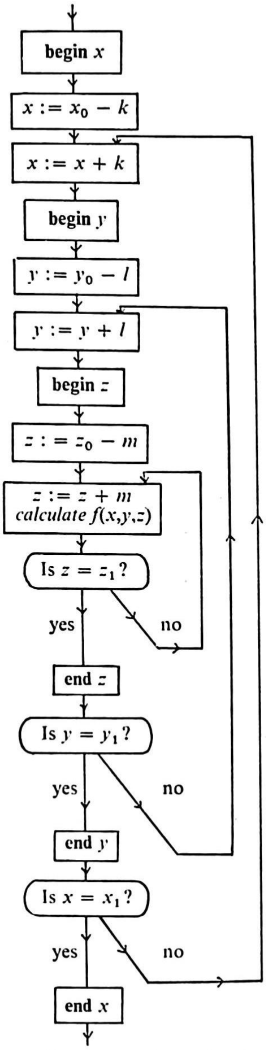


Fig. 2.7. Evaluation of $f(x,y,z)$ for range of values of $x,y,z$.

calculated at the *beginning* of the outer loop; if it were calculated just before the test for the end, this test could no longer be "Is $n = m$?". (What should it be?)

Secondly, by proper setting of the starting value of $a$ (outside the inner loop) we are able to use the same inner loop for all values of $n$. The seemingly more natural choice

$$a := 1+\frac{1}{n}$$

would require special treatment when $n = 1$.

This raises the question of how it is possible to anticipate what the appropriate starting values of operands should be. The answer is simple, even if somewhat unexpected: *start writing the programme in the middle and then work outwards!* The innermost loop of a programme should be written first, as this determines what has to be set outside it.

To illustrate this, we shall consider the problem of tabulating a function $f(x,y,z)$, first for fixed values of $x$ and $y$, with $z$ going in steps of size $m$ from $z_0$ to $z_1$. Fig. 2.5 shows how to do this. Now suppose that we wish to extend our table to take account of varying values of $y$. For the moment $x$ is still fixed, but $y$ is to go from $y_0$ to $y_1$ in steps of $l$, and for every value of $y$ we go over the whole range of $z$. Our programme is shown in Fig. 2.6. Finally, we wish to vary $x$ in steps of $k$ from $x_0$ to $x_1$. The values of $x$ must be changed outside the loop controlled by $y$. This is done in Fig. 2.7.

It should be noticed that if there are several loops, one inside the other, the order in which counters controlling the loops are set has to be the reverse of the order in which the counters are tested. Fig. 2.7 shows this very clearly.

Our next example shows how the operation of finding a square root (which is rarely included among the basic machine operations) can be built up from a suitable combination of simpler steps. We use the fact that if $y$ is any approximation to $\sqrt{a}$, a better approximation* is given by

$$z = \tfrac{1}{2}(y + a/y).$$

We may start with an arbitrary value of $y$ and repeatedly improve the approximation. If we wish to evaluate $\sqrt{a}$ to 8 figures, we continue until

$$|y-z|<(0{\cdot}00000001)\times z$$

as in Fig. 2.8.

Our next example requires that numbers be read into the computer. We shall write

$$x := \textit{input}$$

to read a number. This is to be understood as reading the first number available at the input, setting $x$ equal to this value and leaving the computer ready to read the next number.

A commonly occurring problem in statistics is to estimate the mean $m$, variance $v$, and standard deviation $s$ of a population when only a sample is available. If the members of the sample are the $n$ numbers $x_1,x_2,\ldots,x_n$ the requisite formulae are

$$m = \frac{1}{n}\sum_{i=1}^{n} x_i,$$

$$v = s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i-m)^2.$$

* A proof is given in most books on numerical analysis, such as Redish [3].
If $a = 2$, the sequence could be 1, 1·5, 1·4167, 1·414207, 1·41421356 which is correct to 8 decimal places.

The calculation of $v$ is simplified by noting that

$$\sum_{i=1}^{n}(x_i-m)^2 = \sum_{i=1}^{n}x_i^2-nm^2 = \sum_{i=1}^{n}x_i^2-m\left(\sum_{i=1}^{n}x_i\right).$$

We shall therefore find the sum of the $x$'s and the sum of their squares.

We assume that the data are available in the order $n, x_1, x_2, \ldots, x_n$, i.e. the first number tells us how many $x$'s are to follow. This is convenient as our programme will then work for any number of $x$'s, no matter how many. The flow diagram for our calculation is shown in Fig. 2.9. We use $b$ and $c$ for accumulating the sums $\Sigma x_i$ and $\Sigma x_i^2$. Note that $b$ and $c$ must initially be made equal to zero.

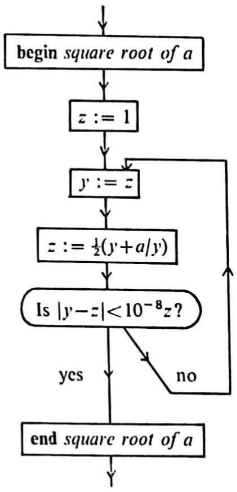


Fig. 2.8. Flow diagram for finding $z = \sqrt{a}$.

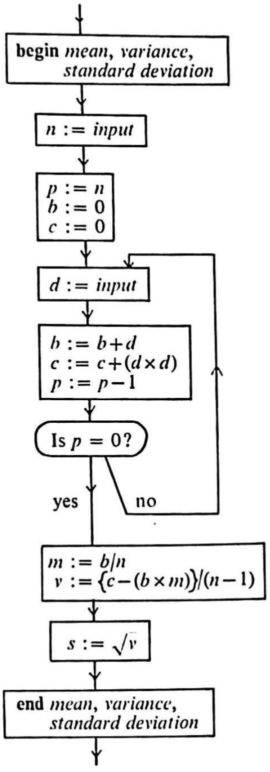


Fig. 2.9. Calculation of mean, variance, standard deviation.

It is interesting to notice that the programme shown above requires storage space for only one value of $x$ at a time. It can therefore be used for samples where the number of $x$'s far exceeds the storage capacity of the computer.

CHAPTER 3

# SUFFICES

In the examples of the previous chapter there was no need to deal with sets of numbers, only with individual numbers. Many problems in mathematics, however, require that we repeat a particular sequence of operations with many numbers. Symbolically this is easily managed by suffices, and computers have facilities for using this powerful technique.

There is a small typographical difficulty. In *writing* mathematics we use symbols like $a_i$, using the spatial layout on the page to carry extra information, to say nothing of italics, bold print, etc. The devices that are used for preparing input to a computer rarely cater for anything more sophisticated than printing on one level. Most of them print capitals only, as upper and lower case are expensive luxuries, and even then the range of symbols will not include much beyond the alphabet and the digits 0 to 9. These considerations do not affect the argument. Our mathematics is just as easily intelligible if we write $a(i)$ instead of $a_i$.*

To operate on the "next member" of the sequence we have to change the appropriate suffix and we do this by an ordinary arithmetic instruction. This immediately suggests a loop of instructions to deal with successive members, the test for leaving the loop being a simple one to see what value has been reached by the suffix. Some examples will make this clearer. The programme is shown in Fig. 3.1.

*Example* 1. Add up the numbers $a_1, a_2, \ldots, a_n$.

*Example* 2. Evaluation of the polynomial

$$p(x) = a_0x^n + a_1x^{n-1} + \ldots + a_{n-1}x + a_n$$

for a given value of $x$. Here we have an example where we enter the loop at a point that is not the natural beginning, as shown in Fig. 3.2.

In organizing a calculation for a computer we must take account of the order in which the data are presented. In the previous example, it is first of all necessary to read the coefficients of the polynomial and place them as $a(0), a(1), \ldots, a(n)$ in the computer's store. The next two programmes (Fig. 3.3) show how the $n+1$ coefficients could be read and then stored in the required order inside the computer.

If we were to use the programme of Fig. 3.3a to read coefficients available in the order $a_n, a_{n-1}, \ldots, a_1, a_0$ then the programme of Fig. 3.2 would evaluate the unwanted polynomial

$$a_nx^n + a_{n-1}x^{n-1} + \ldots + a_1x + a_0.$$

The programmes of Figs. 3.2, 3.3 use the quantity $n$, the degree of the polynomial. How is this to be supplied to the programme? One way, of course, is to write the programme to evaluate polynomials of a given (fixed) degree. This, however, would mean that we should have to have a separate programme for every possible value of $n$. Also, if we had to evaluate polynomials of different degrees in the course of one programme it would be necessary to incorporate several copies of the procedure for evaluating a polynomial. This takes up valuable space. It is more economical to give $n$ as part of the initial data, together with any other parameters that may occur, e.g. the values of $x$ for which we wish to evaluate $p(x)$. The main advantage is the more

* No special meaning is attached to the shape of the brackets.

general applicability of our programme, as the same set of instructions can now be used whatever the value of $n$ (as in the example of Fig. 2.9).

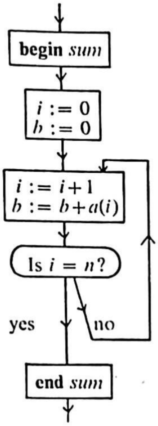


Fig. 3.1. Programme for $\sum_{i=1} a_i$.

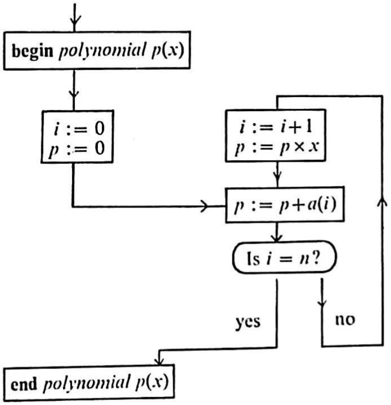


Fig. 3.2. Programme to evaluate a polynomial.

*Problem for the reader.* Check that the method of Fig. 3.2 ("nested multiplication") is more economical in multiplications than a method which first evaluates all the requisite powers of $x$.

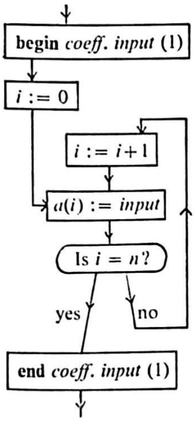


Fig. 3.3(*a*). Input of coefficients available in the order $a_0, a_1, \ldots, a_n$.

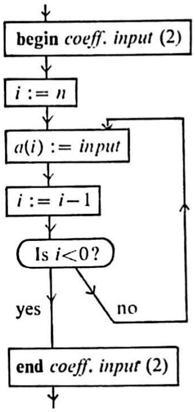


Fig. 3.3(*b*). Input of coefficients available in the order $a_n, a_{n-1}, \ldots, a_1, a_0$.

The next example (Fig. 3.4) is a complete programme for tabulating a polynomial of degree $n$ for the values $x_0, x_0+h, \ldots, x_0+rh$. This programme incorporates the programmes of Figs. 3.3a and Fig. 3.2. The data must be presented in the order

$$n, a_0, a_1, \ldots, a_{n-1}, a_n, x_0, h, r$$

because this is the sequence in which the corresponding input instructions are obeyed.

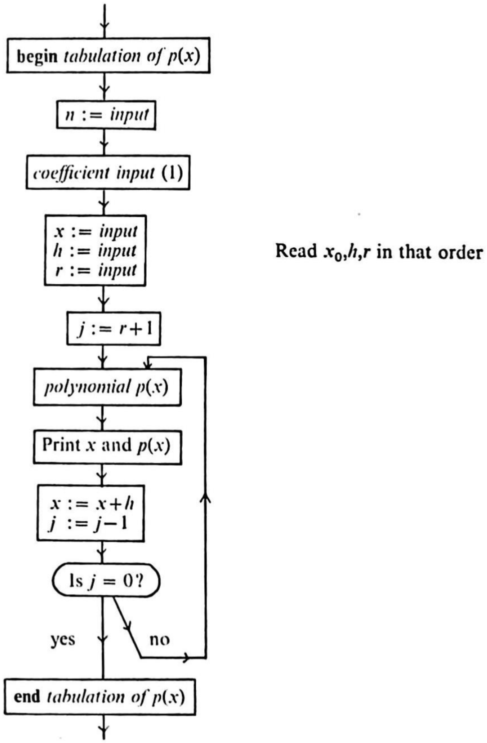


Fig. 3.4. Tabulation of polynomial $p(x)$ for various values of $x$.

In this programme all quantities that are used are presented to the programme as input data. The programme is therefore completely general and can be used for any desired values of $n, x_0, h, r$.

In all the preceding examples we have enclosed all the steps of a programme in boxes. Although this device is helpful to the beginner, it is too clumsy for general use and we shall now abandon it.

Another simplification that we shall make in the written form of our programmes concerns the alternative routes to be followed after a test instruction. The method used so far, namely drawing arrows to the appropriate points, cannot be employed for communicating with a computer; also it is awkward to use in a programme that cannot be written on a single sheet of paper. The simplest alternative is to give a *label* to those instructions to which we wish to refer. This label can then be used in a test instruction, together with appropriate wording, such as "*if . . . then go to . . . else . . .*".* At this stage we shall not make any definite rules about labels but we adopt the convention that they are to be terminated by a colon when they precede an instruction.

The problem of having numbers in the right order in the store is more involved when we have a two-dimensional array, such as the elements of a matrix. The numbers presented to an input device can be read only as a single sequence. We have to decide the order in which the numbers are to be read and how such an array is to be stored as a single sequence (which need not be the same as at input). This is the subject of the next example.

We suppose that the coefficients $a_{ij}$ of the $m \times n$ matrix **A** are available column by column at the input device. For checking purposes every column will be followed by a "column check-sum" which is the sum of the previous $m$ numbers. This serves two purposes, to check that the matrix has been correctly prepared for the computer, and also as a simple check against malfunctioning of the input device itself.

```
             begin matrix input
             m := input
             n := input
             k := 0
             j := 0
column:      j := j+1
             i := 0
             s := 0
read:                    k := k+1
                         i := i+1
                         a(k) := input
                         s := s+a(k)
                         if i ≠ m then go to read else continue
             b := input
             s := s−b
             if s ≠ 0 then go to error else continue
             if j ≠ n then go to column else go to exit
error:       stop (check-sum failure)
exit:        end matrix input
```

Fig. 3.5. Input of an $m \times n$ matrix with check-sums.

For the $2\times2$ matrix $\begin{pmatrix} 11 & 12 \\ 21 & 22 \end{pmatrix}$ the numbers presented to the input are in the sequence 11,21,32,12,22,34; the matrix elements are stored in the order 11,21,12,22. Normally some emergency action would follow after a check-sum failure, such as trying a second time, but this need not concern us here. It is important to realize that it is possible to get a check-sum failure without

* This slightly curious form of wording is chosen to conform with the conventions of ALGOL (described in Part III).

any part of the computer working incorrectly, as rounding errors will generally give a check-sum that does not agree exactly with the sum of the elements. To overcome this difficulty we often put a tolerance on the check-sum and are prepared to accept the computer's input if the number $s-b$ is smaller than this tolerance. This topic is discussed in Chapter 4, where we consider the representation of numbers inside the machine.

The next example concerns finding the largest of a set of positive numbers, $a(1)$, $a(2)$, $a(3)$, . . . , $a(n)$. As this will usually be part of a larger programme, we assume that the $a$'s and $n$ are available in the store.

The method consists of finding which is the largest of the first $k$ numbers and then comparing this with the next possible candidate $a(k+1)$ to determine the largest of the first $k+1$ numbers. We start this process with $k = 2$ and repeat it until all numbers have been examined. The programme uses $m$ to record the position of the largest element at each stage.

```
          begin max. search n
          k := 1
          m := 1
next:     k := k+1
          if a(m)>a(k) then go to test else continue
          m := k
test:     if k ≠ n then go to next else continue
          b := a(m)              (b is largest element)
          end max. search n
```

Fig. 3.6. Programme to find the largest element of a sequence.

*Problems for the reader.* What happens if $n = 1$ and how should one deal with this case? What happens if there are two elements that are equal and largest?

The method of the preceding example gives us the value of the largest element and also its position in the sequence. This is required in the next example, where we sort a sequence of positive numbers into ascending numerical order.

```
          begin sort n
largest:  if n = 1 then go to exit else continue
          max. search n          (Programme of Fig. 3.6)
          a(m) := a(n)
          a(n) := b
          n := n-1
          go to largest
exit:     end sort n
```

Fig. 3.7. Programme for sorting $n$ positive numbers.

As a method of sorting numbers the above is very slow. Its main virtue is that the only storage space used for numbers other than the $a$'s is that for $b$. The number of operations is proportional to $\frac{1}{2}n^2$. There are far more efficient methods available, where the number of operations is proportional to $n \log_2 n$ but they require more storage space and are not so easy to understand. However, the fact that we can use a computer to sort numbers is very important for commercial applications of a computer. The numbers involved could be invoice serial numbers, for instance, which have to be put into their correct order. It is not difficult to adapt the above programme so that it is not only the serial numbers that are moved around in the store, but also the information associated with them.

An interesting possibility arises if the labels are given in a (partly) numerical way, for instance as *stage* 1, *stage* 2, etc. It would be useful to be able to write instructions like

**go to** *stage* (*s*)

where *s* has been set, or even evaluated, somewhere in the programme. This is indeed possible and such a programming device is called a *switch*. The use of a switch is illustrated in the next example.

Suppose that we are given an angle $\theta$ (in degrees) between 0° and 360°, and wish to find *sin* $\theta$, but that the only programme we have for evaluating a sine requires that the angle is between 0° and 90°. We therefore have to reduce $\theta$ to an angle in the first quadrant but have to keep information about the value of $\theta$ so that we can find the correct sign to attach to our result.

```
                      begin any quadrant
                        q := (integer part of θ/90)+1
                      go to quadrant (q)
quadrant 4:             θ := 360−θ
                      go to quadrant (1)
quadrant 3:             θ := θ−180
                      go to quadrant (1)
quadrant 2:             θ := 180−θ
quadrant 1:             f := sin θ
                      go to sign (q)
sign 3: sign 4:         f := −f
sign 1: sign 2:       end any quadrant
```

Fig. 3.8. Example of a switch. *sin θ* stands for any programme that evaluates the sine of an angle in the first quadrant.

A point to notice in this example is that it may be helpful to give two different labels to one instruction. Programmes to evaluate trigonometric functions of angles in any quadrant are available for all computers; the above example gives an indication of the type of calculation that has to be done by such a programme.

Our last example in this chapter gives an important variation of a switch, by showing how it is possible to use one programme as a block at several points in a larger programme although only a single copy of this block is stored.

We return to an earlier example, the factorization of an integer $n$ (Fig. 2.2 on page 8). The process of trying a possible factor $m$ is the same whether $m$ is even or odd, and so we could use the same instructions in the two cases. This would make our programme appear as shown in Fig. 3.9. As can be seen we have had to write out twice (and store twice in the computer) the instructions for testing $m$. The only significant difference is the name of the label we go to if $r \neq 0$. The use of a switch allows us to store only one copy of these instructions and is shown in Fig. 3.10.

A set of instructions that is stored only once, but can be entered from several points and ends with a jump to one of several places, is called a *subroutine*. The indication of where to return to in the main programme is known as a *link*.

If we try to use the polynomial evaluation programme of Fig. 3.2 for different polynomials, we have an apparent difficulty because the coefficients are to be in particular places in the store. As a

```
         begin factorize n
         m := 2
               divide:   q := integer part of n/m
                         r := n - m×q
                         if r ≠ 0 then go to odd else continue
                         print m
                         n := q
                         go to divide
odd:     m := 3
               trial:    q := integer part of n/m
                         r := n - m×q
                         if r ≠ 0 then go to next else continue
                         print m
                         n := q
                         go to trial
next:    if n = 1 then go to exit else continue
         if q<m then go to prime else continue
         m := m+2
         go to trial
prime:   print n
exit:    end factorize n
```

Fig. 3.9. Factorization of an integer (again).

```
         begin factorize n
         m := 2
         k := 1
         go to trial
case 1:  m := 3
         k := 2
         go to trial
case 2:  if n = 1 then go to exit else continue
         if q<m then go to prime else continue
         m := m+2
         go to trial
prime:   print n
         go to exit
trial:                 q := integer part of n/m
                       r := n - m×q
                       if r ≠ 0 then go to case (k) else continue
                       print m
                       n := q
                       go to trial
exit:    end factorize n
```

Fig. 3.10. Factorization of an integer, using a programme switch.

result, this block could not easily be used as a subroutine to evaluate two different polynomials in the course of the same programme. In the different autocodes described in Part II we overcome this kind of difficulty by specifying the position of $a_0$, the leading coefficient. ALGOL, described in Part III, has an elegant method of solving this very common problem, by introducing the concept of a *formal parameter.*

CHAPTER 4

# THE REPRESENTATION OF NUMBERS

In the preceding chapters no mention has been made of the accuracy* we can expect from a computer. Only a finite number of digits can be used to represent a given number and so all calculations can be carried out only to limited, even if high, accuracy. Most machines are designed to use the same number of digits for all numbers and are then said to have a *fixed word-length*.† For fixed word-length machines we can typically expect 10 decimal digits to be used. The fixed number of digits used to represent a number in a fixed word-length computer puts a limit on the range of values the numbers may take. For example, the largest positive integer represented by 10 digits is +9999999999. We say that *overflow* occurs when a number goes outside the permitted range, and numbers appearing in our calculations have to be scaled so that overflow does not occur. It may be extremely difficult to do this scaling adequately and to overcome this difficulty we often use *floating-point arithmetic*. In this system numbers are represented by a certain number of significant digits together with a scaling factor, typically a power of 10, to give the position of the decimal point. For example, $\pi$ could be represented with 10 significant digits in any of the following (equivalent) forms:—

$$\begin{aligned} 3{\cdot}141592654 \times 10^{0} &= 0{\cdot}003141592654 \times 10^{3} \\ &= 31{\cdot}41592654 \times 10^{-1} \\ &= 0{\cdot}3141592654 \times 10^{1}. \end{aligned}$$

Floating-point arithmetic also gives a limited permitted range of numbers but the limit now depends on how many digits are used for the exponent in the scaling factor. Two decimal digits for the exponent would allow numbers as big as $10^{99}$ and as small as $10^{-99}$; such a range is sufficient for most calculations.

Some numbers, however, are known to be integers small enough to be represented exactly by the number of digits used. Examples are numbers used as counters for repeating loops of instructions and the suffices we use to refer to sets of numbers. For such integers there is no need to use any scale factors and we can use *fixed-point arithmetic*. On several machines, especially the more recent models, floating-point arithmetic is achieved by circuitry, and is then comparable in speed with fixed-point arithmetic. A machine equipped only with fixed-point facilities can be programmed to carry out floating-point arithmetic but calculations are then very much slower.

The necessity for using finite word-length affects the accuracy we can achieve. Most numbers cannot be represented exactly by the number of digits allowed and so there will be rounding errors associated with our numbers. Results calculated from rounded numbers will again be in error and so errors are propagated throughout a calculation. This is a problem of numerical analysis, not of computer programming, though of course programmes should be written so that the best numerical methods available are used. The choice of methods is outside the scope of our book.

A consequence of the propagation of errors is that comparisons between numbers will not be exact. For example, if $x$ is to go from 0 to 1 (both of which can be represented exactly) in steps of

* Computational accuracy is meant, not mechanical reliability. Modern computers work reliably for long periods and generally include some circuitry to check that no malfunctioning has occurred.

† Some computers have a *variable word-length*.

$1/12 = 0{\cdot}0833333333$, correct to only 10 decimals, our last value is $0{\cdot}9999999996$, and in a test of the form

**if** $x = 1$ **then . . . else . . .**

we should find that $x$ never takes the exact value 1. This particular difficulty can be overcome in various ways. The best method is to use integers to count exactly how many times the value of $x$ has been increased; another way is to change the test from an equality into an inequality and adjust the limit, e.g.

**if** $x \geqslant 0{\cdot}95$ **then . . . else . . .**

A further consequence of the truncation of our numbers is that check-sums, as in the programme of Fig. 3.5 (on page 17), can in general only be satisfied to within a certain tolerance.

It should be noticed that the order in which we perform operations may have an appreciable effect, as can be seen from the following example: working to nine significant figures,

$$\begin{aligned}(1{\cdot}23456789+123456789)-123456788 &= 123456790-123456788\\ &= 2{\cdot}00000000\end{aligned}$$

but

$$\begin{aligned}1{\cdot}23456789+(123456789-123456788) &= 1{\cdot}23456789+1{\cdot}00000000\\ &= 2{\cdot}23456789,\end{aligned}$$

where in both cases the term in brackets is evaluated first. The computer may not give any indication that accuracy has decreased through loss of significant figures, and so we see that floating-point working has the danger that it may give spurious accuracy.

For design reasons many machines work in binary, that is, in the scale of 2, rather than in the more familiar decimal scale. The conversion from the decimal used outside the machine to the binary used internally, and vice versa, is performed by the machine itself. Everything that has been said earlier about the limitations of representing numbers by only a finite number of decimal digits applies equally to machines that work in binary. The conversion of numbers from one scale to another brings its own errors, which can be regarded as further rounding errors. Fractions that are exact in a finite number of digits in decimal need not be so in binary (or any other scale, for that matter). Thus in decimal $\frac{1}{5} = 0{\cdot}2$, whereas in binary* $\frac{1}{5} = 0{\cdot}0011001100110011\ldots$

We now consider briefly the way information enters and leaves a computer. The most common methods use paper tape or punched cards. The problems, such as they are, are slightly different in the two cases.

The most widespread form of paper tape at the present has *characters* consisting of up to 5 holes, not including a sprocket hole, across its width.† A code with 5 holes to represent different characters allows only 32 different symbols. This is regarded as inadequate and so we normally have two codes, called "Letters" and "Figures", each of 30 characters. The remaining two characters cause no printing but merely change a printer from one code to the other. They are generally called the "Figure Shift" and "Letter Shift" characters. The tape is read serially, character by character. All characters after "Figure Shift" are regarded as being in the "Figures" code until a "Letter Shift" character is encountered; thereafter, the code is "Letters" until we reach a "Figure Shift". The order in which the characters appear on the tape is the same as their order on a printed page. There are some symbols that can be punched on the tape to control the layout of printing. These special symbols are used to put spaces between items ("Space" or *Sp*) and to arrange for printing to start at the beginning ("Carriage Return" or *Cr*) of a new line ("Line Feed" or *Lf*); it is not possible to turn back to a previous line.

* The $k^{th}$ digit after the point has the value $2^{-k}$ so that $\frac{1}{5}$ is $2^{-3} + 2^{-4} + 2^{-7} + 2^{-8} + \ldots$
† Wider tapes, with 7 or 8 holes, are now becoming available; they allow a bigger range of characters.

Paper tape allows a wide variety of printing layout because this is automatically controlled by the symbols appearing on the tape. The speed of printing is not very high, being comparable to a good typing speed. Frequently, therefore, computers do not have such a printer attached directly but only a paper tape punch which can operate several times faster than a printer, and the tape is then printed away from the machine.

Unfortunately there has been no standardization between manufacturers regarding the tape code used on their machines, nor are the extra symbols necessarily the same.

Punched cards contain a number of columns (usually 80) and rows (usually 12) in which holes may be punched. The normal card thus has 960 positions for holes.

The rows are labelled* Y,X,0,1, . . . ,9. A hole in a numbered row has the value of that number, e.g. the 5-row indicates a 5, the other two rows are used for + and −. Letters are coded by having two holes punched in the same column. Again, there is a profusion of codes. Printers worked by punched cards are capable of printing the whole of one line at the same time and are therefore often called *line printers.* All the information on a card is printed together, every card causing the printing of a new line. There is therefore no need to have "Carriage Return" and "Line Feed" symbols. Spaces between numbers are managed mechanically by a control panel on the printer. A different problem arises because of the limited width of a card and the fact that many computers using cards are not capable of using all the columns. This imposes a very rigid format on input and output, compared to the flexibility of paper tape. Numbers have to be in certain specified positions on the card, with a predetermined number of figures, and instructions all have to take the same number of characters. On the other hand, card-operated printers can be extremely fast and are much faster than character-by-character printers.

These differences between cards and paper tape arise primarily from their quite different original uses. Paper tape has been used for a long time in communication networks ("Telex" for example) before it was adopted for use as a computer input/output medium. Punched cards are very common in commercial use and were in fact originally developed for census work. Different columns can be used easily to designate different properties, so that the layout itself carries information.

* This is one system only; there are several others. All agree on 0, 1, . . . , 9, the variations affecting the naming and significance of the other two rows.

CHAPTER 5

# AUTOCODES

The preceding chapters have described in some detail, and almost entirely by examples, how to set about planning a programme. The next problem is to translate such a plan into action, that is, to put the programme into a form acceptable to the machine. In all our programmes so far we have made liberal use of words and symbols, so the question arises: "Can we communicate with a computer in this way?"

If we consider the computer to be just a large piece of expensive circuitry, the answer is certainly "No". As was mentioned in Chapter 1, very few basic arithmetic operations are included in the computer's repertoire. The locations in the store are identified by numerical *addresses* and, similarly, the operations are all numbered and instructions are held entirely in numerical form.* It is perfectly possible to write programmes in this style, often called "machine language" or "machine code", once one knows it well enough. However, a great deal of very detailed clerical work is involved because all internal administration of the programme has to be attended to by the programmer (strictly, by the coder), and it is very easy to make mistakes in this process. For general use by the non-specialist this system is too difficult and often rather frightening.

Quite early in the development of computers it was realized that the machine itself could be made to help in the clerical work of coding. It was found possible to write programmes that made the computer appear to other users as if it had most of the facilities for dealing with programmes written in commonly occurring symbols. The earliest of these programmes were interpretive in that they never made a machine code programme from the symbolic instructions but translated and executed every instruction each time it was encountered. These *interpreters* tend to be slow, as translation is time-consuming when every word has to be looked up in a dictionary every time it occurs. Later, more ambitious programmes called *compilers* became available. These produce a machine language programme from the symbolic instructions without executing any of them until the process of compilation is complete. The resulting programme is in the machine's normal code and does not have to be recompiled subsequently. The efficiency of compiled programmes is in general comparable to the work of an average programmer writing in machine language.

Such programming systems exist for many computers and are usually called *autocodes*. The use of an autocode will certainly save time in writing the programme. Further time is likely to be saved at the stage known as *development*, i.e. testing the programme on the computer to remove errors; these will be easier to locate, and fewer in number, than if the programme had originally been coded in machine language. However, the final programme may take longer to run at the *production* stage. This difference may be small (as on the Ferranti Mercury and the Elliott 803 computers) or comparatively large (as on the Ferranti Pegasus). For programmes that are used on relatively few occasions† this may be less important than the saving in development time.

Any autocode programme consists of a series of instructions in which one number is calculated

* The exact nature of this representation does not concern us, nor is it the same for different machines. However, the idea is not unfamiliar as, for example, all telephone numbers are essentially numerical. It is just that WHI 1212 is easier to remember than 944 1212.

† Programmes that are to be used very frequently are generally written in machine code: often the user has only to supply the data without knowing how the programme works.

from some others which have been calculated earlier. Often, the new number is the result of evaluating a formula and several numbers appear on the right hand side, e.g.

$$x := (-b+\sqrt{b^2-4ac})/2a.$$

For some autocodes it is permissible to write such a formula as a single instruction. If, however, we were to do this evaluation by hand, or with a desk calculator, we would find that at each stage we deal with at most two numbers and the above formula is then worked out in several steps, as

$$\begin{aligned} d &:= b\times b \\ e &:= a\times c \\ e &:= 4\times e \\ d &:= d-e \\ d &:= \sqrt{d} \\ d &:= -b+d \\ e &:= 2\times a \\ x &:= d/e \end{aligned}$$

Any calculation can be broken down into such steps, and the simpler autocodes require instructions like this.

In Part II we present information about three autocodes in common use in Britain, namely those for the Ferranti Pegasus and Sirius, Elliott 803, and Ferranti Mercury computers.

The very profusion of autocodes has led to an attempt to set up an international computer language, ALGOL (for *algo*rithmic *l*anguage), to be used as a method of communication between programmers and also between programmers and machines. The writing of an ALGOL compiler for any particular machine is a big and complicated job and much programming effort is being directed towards the construction of such compilers. It is likely that ALGOL will be the main programming language of many computers. An introduction to ALGOL is given in Part III.

PART II

# AUTOCODES

CHAPTER 6

# THE PEGASUS-SIRIUS AUTOCODE

## 6.1. General Information

Pegasus and Sirius are the names of computers made by Ferranti Ltd., of Manchester. Pegasus is a medium-sized, medium-speed machine with a magnetic drum store of 8192* words capacity, working in fixed-point binary arithmetic, with a word length of 39 binary digits which corresponds to 11 decimal places. Sirius is a more compact machine, working in decimal. The store is normally 4000 words but is extendable to a theoretical maximum of 10000 words, a word holding 10 decimal digits. Both machines use paper tape as their input and output media, using the code given in the Appendix. The autocodes used for the two machines are practically identical and everything in this chapter applies to both unless otherwise stated.

## 6.2. Variables, Indices, Instructions

The autocode is very simple. Results are calculated from one or two numbers previously found, either by one of the simple arithmetic operations $(+, -, \times, /)$, or by some of the more common mathematical functions. Two types of number are recognized by the autocode, namely *variables* and *indices*. The variables are floating-point numbers and are used for doing most of the arithmetic. They are called

$$v0,\ v1,\ v2,\ \ldots$$

Indices are integers that are used for counting and as suffices (in the sense of Part I). They are denoted by

$$n0,\ n1,\ n2,\ \ldots$$

The allowed range of values of variables and indices is given in section 6.12 (page 40). It is possible to use the indices as suffices of variables as in

$$vn1,\ v(1+n2),\ \ldots$$

where the value of the index determines which particular variable is meant. We are restricted to the form $v(1+n2)$ and are not allowed the mathematically equivalent $v(n2+1)$. Similarly the index must always be added, although its value may be negative, i.e. $v(100-n5)$ is forbidden, but $v(200+n6)$ may be used with negative values of $n6$ provided that the suffix is positive. We may even write $v(-1+n2)$, i.e. the constant part may be negative, so long as the suffix is positive.

The index used is often called a *modifier*. Only one index is allowed as a modifier for any variable. It is not possible to use indices to refer to other indices, i.e. $n(1+n2)$ is not allowed.

Arithmetic instructions are written in the form illustrated by

$$v1 = v2+v3$$

or

$$n1 = n1-1,$$

every instruction being written on a new line. The symbol = is used instead of the := of Part I. A variable or index appearing on the right-hand side of an instruction is unchanged unless

* Early models have a smaller drum, with only 5120 words.

it is also to be the result of the operation. It is not permitted to mix variables and indices in an instruction, except as explained below. This rule is most easily remembered by noting that the numbers taking part in the arithmetic called for by an instruction are of the same type. The full list of allowed arithmetic instructions is given in the table in section 6.13.

In instructions involving variables we may replace a variable occurring on the right-hand side by an unsigned number, so that

$$v0 = v1 \times 3{\cdot}14159$$

is allowed. The "=" symbol may, if desired, be followed by a minus sign, to give the negative result. Such a minus sign may not follow another arithmetic symbol, so that $v0 = -v1 \times 3{\cdot}14159$ is in order but $v0 = v1 \times -3{\cdot}14159$ is not.

Rules for instructions involving indices are exactly similar. An index may be replaced by an integer, i.e. a number without a decimal point.

We may change a variable into an index, and vice versa, by the instructions

$$\begin{aligned} v0 &= n1 \\ n0 &= v1 \qquad \text{(nearest integer)} \\ v0 &= n1/n2 \end{aligned}$$

The usual rules apply: we may replace variables by numbers, indices by integers and a minus sign may follow the equals sign. These are the *only* instructions where indices and variables occur as equal partners. The index $n0$ is changed by all input instructions (section 6.6) and so can be used only as a temporary store. On Sirius $n0$ is more rapidly available than the other indices, and so it is advantageous to use it as a modifier or counter wherever possible.

## 6.3. Jumps and Labels

Instructions to which jumps are to be made are given numerical labels, numbered 0,1,2, . . . The first instruction of a programme is automatically given the label 0 and need not be labelled explicitly by the programmer. The labels may be allocated in any order and need not be consecutive. It is not permitted to give the same label to different instructions, nor does it make sense to do so.* A label is written before the instruction to which it refers and is terminated by a right-bracket, ). In Pegasus, but not Sirius, an instruction may be given more than one label if desired.

In Part I we used the words **go to** and **if** in our conditional jumps. These are replaced in this autocode by an arrow and a comma. Jump instructions are written in the form

$$\rightarrow 1$$

read as "go to label 1" (unconditional jump) or

$$\rightarrow 1,\ v1 > v2$$

read as "go to label 1 if $v1 > v2$" (conditional jump). There is no comma in the unconditional jump instruction. In the case of conditional jumps, if the condition is not satisfied the programme continues with the instruction immediately following (in fact, **else** *continue* is understood).

Conditional jumps can be made up with the symbols $=$, $\neq$, $>$ and $\geqslant$, but not $<$ and $\leqslant$ which are not included in the Ferranti teleprinter code. We are only allowed to compare numbers, not expressions, and we may only compare like with like. Minus signs may be included on either side of the condition. A jump instruction may itself be labelled. Examples of permitted jump instructions are

* If Pegasus reads an instruction bearing a label which has already been allocated, the label is attached to the later instruction.

```
   →1, v1>−v2
   →1, n1≠0
3)→1, v1 = v2
1)→1, v(1+n2) = 0.
```

The last of these is an example of a *loop stop*; obviously no further progress will be made in the programme if $v(1+n2) = 0$. A loop stop can be used to indicate that something has gone wrong with a calculation, e.g. that the required operations on the given data would produce nonsense. A full list of allowed jump instructions is given in section 6.13. Jump instructions that would appear reasonable but are not permitted include (among others) $\rightarrow 1$, $v1 > n1$ and $\rightarrow 1$, $v1 > v2+v3$ and $\rightarrow 1$, $v1 = v2/10$.

The destination indicated in a jump instruction may depend on an index, e.g.

```
→n1
→(1+n2), −v1⩾v(5+n7).
```

This form can be used to implement the "programme switch" described in chapter 3.

As an example of a simple sequence of instructions we now give the coded version of the square root procedure of Fig. 2.8. We assume that $a$ is given as $v1$ and we denote the result by $v2$.

```
   v2 = 1
1) v0 = v2
   v2 = v1/v2
   v2 = v0+v2
   v2 = v2/2
   v0 = v0−v2
   v3 = v2×0·00000001
   →2, v0⩾0
   v0 = −v0
2) →1, v0>v3
```

**Fig. 6.1. Coded programme for finding a square root.**

## 6.4 Functions

We frequently require the absolute value of a number, and so the autocode includes a special function MOD to find it, so that we can replace the instruction (in Fig. 6.1)

```
→2, v0⩾0
v0 = −v0
```

by

```
v0 = MOD v0
```

and the label 2 is now unnecessary. In fact, calculation of a square root is so common a requirement that the autocode can find a square root directly and the whole programme of Fig. 6.1 can be replaced by the single instruction

```
v2 = SQRT v1
```

This saves time and space, as the instructions for finding a square root are now machine code instructions, whereas autocode instructions would have to be interpreted into machine code instructions.

Several of the more common functions may be used directly in the autocode. The full list is given in section 6.13. For any function only one variable may be written on the right-hand side. The only function that may be used with indices is MOD, e.g.

$$n1 = \text{MOD}\, n2.$$

The result is, as usual, of the same type as the numbers occurring on the right-hand side. As before, a minus sign is permitted after the equals sign but not elsewhere.

## 6.5 Some Examples of Programmes

The next coded example is the programme of Fig. 2.4, to find $(1+1/n)^n$ without using logarithms. We assume that $n3$ contains the value $m$.

```
   n1 = 0
1) n1 = n1+1
   v1 = 1
   n2 = n1
   v2 = 1/n2
   v2 = 1+v2
2) v1 = v1 × v2
   n2 = n2−1
   →2, n2 ≠ 0
   PRINT v1
   →1, n1 ≠ n3
   STOP
```

Fig. 6.2. Autocode programme for $(1+1/n)^n$ without logarithms.

It should be noted that the output instruction is not quite correct as it stands; the full form of the PRINT instruction is given below. The example has also introduced a new instruction,

STOP.

This does what it says. The machine will halt until the "Run Key" on the console is operated. (Compare the WAIT instruction of the Elliott 803 autocode, chapter 7).

We can simplify and shorten the above example by using the functions LOG and EXP. This allows us to avoid the inner loop of the programme.

```
   n1 = 0
1) n1 = n1+1
   v1 = n1
   v2 = 1/v1
   v2 = 1+v2
   v2 = LOG v2
   v2 = v1 × v2
   v1 = EXP v2
   PRINT v1
   →1, n1 ≠ n3
   STOP
```

Fig. 6.3. Simplified programme for $(1+1/n)^n$ using logarithms.

Our next example of a sequence of coded instructions is the polynomial evaluation programme of Fig. 3.2 (page 15). We shall assume that $x$ is given as $v1$ and the coefficients are given as $v100$, $v101$, $v102, \ldots$ (i.e. $a_0$ as $v100$, $a_1$ as $v101$, etc.), and that $n$, the degree of the polynomial, is given as $n1$. We shall denote the result by $v0$.

```
   n0 = 0
   v0 = 0
   →1
2) n0 = n0+1
   v0 = v0×v1
1) v0 = v0+v(100+n0)
   →2, n0 ≠ n1
   PRINT v1
   PRINT v0
```

Fig. 6.4. Evaluation of a polynomial (1st version).

We could make this polynomial evaluation work with coefficients anywhere in the store by asking that the location of $a_0$ is given by $n2$ (in the example above we would have $n2 = 100$). The programme now appears as

```
   n0 = n2
   n3 = n1+n2
   v0 = 0
   →1
2) n0 = n0+1
   v0 = v0×v1
1) v0 = v0+vn0
   →2, n0 ≠ n3
   PRINT v1
   PRINT v0
```

Fig. 6.5. Evaluation of a polynomial (2nd version).

As a last example before discussing the input and output instructions we give the coded version of the sorting programme of Fig. 3.7 (page 18), which puts $a_1, a_2, \ldots, a_n$ into ascending order. We assume that $n$ is given as $n1$ and that the sequence of $a$'s starts with $vn2$. Our programme can then appear as follows:

```
   →4, n1 = 1
   n0 = n1+n2
   n0 = n0−1
3) n4 = n2
   n3 = n2
1) n4 = n4+1
   →2, vn3>vn4
   n3 = n4
2) →1, n4 ≠ n0
   v0 = vn3
   vn3 = vn4
   vn4 = v0
   n0 = n0−1
   →3, n0 ≠ n2
4) STOP
```

Fig. 6.6. Sort into ascending order.

Notice that $a_n$ is effectively $v(n1+n2-1)$. This programme uses $v0$ as temporary storage. All programmes require some intermediate results and as a general rule, working space should be at one end of the available storage space. This device puts no unnatural restrictions on the amount of space available for data. It is good practice to leave the low-numbered variables as general working space.

## 6.6 Input and Output

Single numbers may be read into the machine by the instructions

$$n1 = \text{TAPE}$$

or (any $n$ or $v$ may be used)

$$v1 = \text{TAPE}$$

No minus sign is allowed after the equals sign. This instruction reads a single number, starting with a sign† and finishing with either a Space symbol or the Carriage Return—Line Feed symbols. The data tape is left ready to read the next number that appears on the tape—exactly our convention of Part I with $:=$ *input*. Decimal points are punched only if necessary. Incorrect forms of punching cause the computer to stop.

It is possible to read more than one number by a single instruction. To read 15 numbers we can write

$$v1 = \text{TAPE } 15$$

which places the numbers in $v1, v2, \ldots, v15$ respectively. Similarly we may write

$$n1 = \text{TAPE } 10$$

to read 10 indices. We may go even further and write

$$v1 = \text{TAPE } n2 \quad \text{or} \quad n1 = \text{TAPE } n2$$

where the number of numbers to be read depends on the value of $n2$. In this case $n2$ must be greater than zero. The form $n1 = \text{TAPE } n2$ is in order as the value of $n2$ is taken before a possible replacement occurs. The form $v1 = \text{TAPE } n2$ is often used when the data tape carries information about how much data there is to follow, as for instance in the sequence

```
n2 = TAPE
v1 = TAPE n2
```

This is much faster, and more elegant, than the equivalent sequence

```
   n2 = TAPE
   n3 = 0
1) v(1+n3) = TAPE
   n3 = n3+1
   →1, n3 ≠ n2
```

There is also a form of the input instruction that deals with cases where the amount of data is not specified in the programme. We write

$$v1 = \text{TAPE*}$$

and numbers are read, and stored in consecutive variables, until the special character L is encountered on the tape. At this point the computer stops reading numbers and proceeds to the next instruction. As we will often want to know how many numbers have been read, this count is automatically placed in $n0$.

† On Sirius, the + sign may be omitted.

All input instructions set *n*0 equal to the number of numbers read by that input instruction whenever it is met, and all instructions will finish reading when they encounter L on the tape. If an L occurs before any numbers have been read, input stops and the value of *n*0 is correctly set to zero.

If the input instruction finds the character Z the tape stops. This is a temporary stop only and the computer can continue with this input instruction when the controls are operated (unlike L which signifies the end of input by this instruction). Z should always be put at the end of a data tape.

Pegasus and Sirius each have two tape readers and it is possible to read data from either. To read from the second reader we use the word TAPEB instead of TAPE in the input instructions. This is the only change required. It is perfectly possible for a programme to use both readers.

Output is achieved by instructions like PRINT *v*1 or PRINT *n*1 where again any variable or index may be used. Actually no printing occurs, as neither Pegasus nor Sirius has a direct printer, but the results are punched on paper tape which can subsequently be printed. Only one result can be punched at a time.

The PRINT instruction as given so far is not complete. We have to give a "style number" which controls the page layout, i.e. where a number is to be printed in relation to the previous result, and also the position of the decimal point and whether we require fixed or floating-point printing. The style number is a four digit number made up of 3 components, *a*,*b*,*c* to give

$$1000a+20b+c \text{ (note the 20)}$$

where *b* is the number of places required before the decimal point (possibly zero), *c* is the number of places required after the decimal point (possibly zero) and *a* is 1,2,3 or 4 according to the following table:

| | New Line | Same Line |
|---|---|---|
| Floating-Point | $a = 1$ | $a = 2$ |
| Fixed-Point | $a = 3$ | $a = 4$ |

"New Line" printing is preceded by Carriage Return—Line Feed (*CrLf*) whereas "Same Line" has two Space symbols (*SpSp*) before the number. On Pegasus signs are always punched, but Sirius punches *Sp* instead of + signs. Fixed-point printing should be used when the magnitudes occurring can be estimated. In fixed-point, if too many figures are required before the decimal point, spaces are printed instead; if too few figures are allowed before the point the style is automatically changed to the floating-point style with the same *b* and *c* (and the page layout is spoiled but this is a minor matter). No decimal point is punched if $c = 0$, but if $b = 0$ a zero is punched before the decimal point. All output is correctly rounded. For indices only two style numbers are relevant, 3000 and 4000; in spite of their curious appearance they cause fixed-point printing always allowing four digits and a sign. The style number may be given as an index, so that instructions like

PRINT *v*1, *n*2

are permitted.

Fixed-point output may be read directly on a subsequent occasion as a data tape for another programme. It should be noticed, however, that the PRINT instruction precedes a number by *CrLf* or *SpSp*, whereas the input instructions require such symbols as terminating characters. The last number punched therefore has no terminating character; this has to be supplied separately, either on tape-editing equipment or by using the special symbols X and S described below. Floating-point output cannot be read by an = TAPE instruction.

As an example of a programme involving both input and output, we now give the autocode version of the statistical programme of Fig. 2.9.

```
   STOP
   n1 = TAPE
   n2 = n1
   v1 = 0
   v2 = 0
1) v3 = TAPE
   v1 = v1+v3
   v3 = v3×v3
   v2 = v2+v3
   n2 = n2−1
   →1, n2 ≠ 0
   v3 = n1
   v4 = v1/v3
   v5 = v1×v4
   v2 = v2−v5
   v3 = v3−1
   v2 = v2/v3
   v3 = SQRT v3
   PRINT n1, 3000
   PRINT v4, 1026
   PRINT v2, 2026
   PRINT v3, 2026
   →0
```

Fig. 6.7. Autocode programme for mean, variance and standard deviation.

The style number of the PRINT instructions ensures that the number of terms is printed on a line by itself and that the mean, variance, and standard deviation appear on the next line, in floating-point form, with one figure before and six following the decimal point. For a general programme the floating-point style is essential as there is no way of estimating the magnitude of the results.

The STOP instruction has been deliberately placed at the beginning of our programme. While the programme tape is being read into the computer, it is not possible to place the data tape in the reader. The STOP instruction allows us a pause during which we can insert the tape in the reader. (Compare the WAIT instruction of Elliott 803 autocode, chapter 7). It is a good habit to start a programme with the instruction STOP.

The way of finishing the programme may appear somewhat odd. However, it does what is required as the programme duly comes to a halt on the STOP instruction (labelled 0). Moreover, we are now ready to deal with the next batch of data.

A common requirement is the introduction of extra gaps, either horizontally or vertically, into the layout of printed results, and the autocode caters for this. By placing an X for Carriage Return-Line Feed, or an S for Space, in front of an arithmetic instruction (i.e. any instruction that "calculates" and produces a result), we get additional control over the appearance of our printing.

Only one X or S may be attached to one instruction and if this instruction happens to be labelled it is advisable to write the letter after the right bracket that terminates the label.

Often we wish to inspect intermediate results of a calculation, especially while the programme is still being developed and there are suspected errors in it. Such *optional printing* is available and is obtained by writing XP or SP in front of an arithmetic instruction. For variables we then get floating-point printing with 9 decimal places, either on a new line or on the same line as the previous printing, i.e. XP and SP correspond to PRINT style numbers 1009 and 2009 respectively. For indices the printing is the usual fixed-point index printing. It is possible to suppress XP and SP printing from the control switches on the computer console, and for this reason it should never be used for general output. We may not have more than one of X, S, XP and SP attached to any instruction.

## 6.7 Beginning and Ending a Programme, Bracketed Interludes

When an autocode programme is to be read into the computer we have to tell the autocode where the programme starts and where it ends. At the start of all our programmes we write the symbols

J1.0 (for Pegasus)

or

J$\nu$1 (for Sirius)

After reading these introductory symbols* the instructions of the programme are read into the computer. At this stage they are being stored but not yet obeyed.

In Pegasus autocode it is possible to enclose a sequence of autocode instructions in brackets. Such a sequence is called a *bracketed interlude*. When the closing bracket, ), is read the computer starts obeying the first instruction of the interlude and continues with the other instructions of the interlude. One of these can be a jump to a labelled instruction outside the interlude in which case the interlude is left. This gives us

$$(\rightarrow 0)$$

as a possible way of entering our programme. This interlude consists of a single instruction which jumps to label 0. It is the most commonly used way of entering a programme.

If the interlude does not end with a jump the computer will continue to read instructions after the interlude has been obeyed. The new instructions are stored in the place previously occupied by the interlude and all trace of this is lost. This gives us the reason for actually wanting an interlude in the first place; an interlude is a way of having a piece of programme obeyed once, leaving some parameters set to particular values to be used by the main programme, and not taking up valuable space after its work has been finished.

The Sirius autocode recognizes (→0) but otherwise the interlude facility cannot be used. The instructions of an interlude are not lost if no further instructions are read. Thus the interlude

$$(\rightarrow 0)$$

not only enters our programme but we also have the instruction

$$\rightarrow 0$$

as the last instruction of our programme, so that it will automatically return to the instruction labelled 0.

* They are in the computer's machine code and direct it to start at the beginning of the autocode compiler, which then starts reading instructions.

## 6.8 Headings

### (i) Name

It is possible to give a programme a *name*. This is punched at the beginning of the programme,* and must be preceded by the letter N. The end of the name is specified by punching "blank tape", at least two consecutive figure-shift characters. When the programme is read into the computer the name is copied on to the output tape but the name itself is not stored. Data may also have a name, preceded by the letter N and terminated by blank tape as above.

Headings and descriptions may be copied on to the output tape by this facility. On Pegasus this is the only way (in autocode) of copying texts. It is possible to use an input instruction to copy headings without reading any numbers, by giving "data" consisting of the name, which is copied, and then the character L to signify that input is to end. As all input instructions destroy the original contents of *n*0 the way causing least loss of information is

$n0 =$ TAPE or $n0 =$ TAPEB

### (ii) TEXT (Sirius only)

The Sirius autocode allows texts to be stored as part of the programme. At a point in the programme where we wish to punch a text during operation of the programme we write the word TEXT followed by the symbols we wish to print. For example,

```
TEXT
DATA INCONSISTENT
```

would cause the words DATA INCONSISTENT to be printed when the programme is obeyed, but not when the programme is read. A text is limited to a maximum of 50 characters. It is terminated by two consecutive figure shift characters.

### (iii) Date (Pegasus only)

The *date* is normally stored in Pegasus. It may be punched on the output tape, to give a record of when a programme was run, by putting the character D at the very beginning of the programme tape, normally before the name and certainly before the J1.0. D may not appear on a data tape.

## 6.9 Programme Alterations (Pegasus only)

It is, unfortunately, only too easy to make mistakes in writing programmes. Corrections can be made by editing the programme tape but sometimes only a few instructions need to be corrected. The Pegasus autocode allows this, without the necessity of repunching the whole programme. After the programme has been read into the store a tape with corrections can be fed in. We have to tell the computer which instructions are to be changed. This is done by saying where the instruction is relative to a labelled instruction. For example, 3, 5 refers to the 5th instruction *after* that labelled 3 and to change it we write

```
ALTER 3,5
New instruction
```

Such a tape is called *an ALTER sequence* and must be headed by the symbols

J1.2

To alter the instruction labelled 3 we could write either

ALTER 3,0 or more simply ALTER 3

* I.e. before J1.0 or J$v$1.

Every ALTER affects only one instruction, and to change two consecutive instructions we have to write e.g.

```
ALTER 3,5
New instruction
ALTER 3,6
New instruction
```

Only one J1.2 is necessary, or indeed allowed, at the start of the ALTER sequence and we finish by an interlude to enter the programme as before.

*Example* 1.

```
J1.2
ALTER 3,5
v3 = v5×v(1+n2)
(→0)
```

changes an instruction and then enters the programme at label 0.

*Example* 2.

```
   J1.2
   ALTER 3,1
3) v5 = v5×10
   (→0)
```

moves the label 3 from one instruction to another.† The instruction itself may be changed as well, of course.

The ALTER facility is not available on Sirius.

## 6.10 Approximate Equality

In Chapter 4 we mentioned the fact that because of rounding errors numbers will not be represented exactly inside the machine. As a consequence it is most unlikely that tests for equality between two computed numbers will ever be satisfied. To overcome this difficulty the autocode includes tests for approximate equality of variables to a specified number of significant digits. The instructions are written like

$$\rightarrow 1,\ v2 =^{*} v3 \text{ and } \rightarrow 1,\ v2 \neq^{*} v3$$

where the number of significant digits to which the test is applied is given in *n*0. For Pegasus this is given as a number of binary digits, $0 \leq n0 \leq 28$;‡ for Sirius *n*0 is the number of decimal digits, $0 \leq n0 \leq 8$. It is essential that the numbers occurring in these test instructions are non-zero.

If there is any danger of either number in such a comparison being exactly zero, it is necessary to add a non-zero constant to both sides and then make the comparison. For example, to make a jump if *v*1 is approximately equal to zero, we could use

```
v1 = v1+1
→1, v1 =* 1
```

to obtain a valid comparison.

## 6.11 Other Facilities

We have described most of the commonly used features of the Pegasus/Sirius autocode. For other facilities, which include how to read further instructions, scaling of numbers at input and combin-

† See section 6.3 regarding two instructions with the same label.
‡ To convert this to decimal digits multiply by 0·3; for example 20 binary digits correspond to 6 decimal digits.

ing autocode with machine code, the reader should consult the Ferranti publications dealing with Pegasus and Sirius autocodes.

## 6.12 Allowed Ranges for Numbers and Allocations of Store

| | *Pegasus* | *Sirius* |
|---|---|---|
| Indices | $-8191 \leqslant n \leqslant 8191$ | $-5\times10^9 \leqslant n \leqslant 5\times10^9$ |
| Variables | $10^{-77} \leqslant \lvert v\rvert \leqslant 10^{76}$ or $v = 0$ | $10^{-50} \leqslant \lvert v\rvert \leqslant 10^{50}$ or $v = 0$ |

On Pegasus the range quoted for variables is the decimal equivalent of the range used in binary.

| | *Pegasus* | *Sirius* |
|---|---|---|
| Labels | 102, numbered 0, . . . , 101 | As for Pegasus, but space is needed only for those labels actually used. |
| Indices | 28, numbered $n0$, . . . , $n27$ | 28, numbered $n0$, . . . , $n27$ |
| Variables | 1380, numbered $v0$, . . . , $v1379$ | This depends on the compiled programme. The autocode prints out the amount of space available. |
| Instructions | 594 (210 on small drum machines) | |

## 6.13 Allowed Arithmetic and Jump Instructions

*Simple Arithmetic*

| | |
|---|---|
| $v1 = v2$ | $n1 = n2$ |
| $v1 = v2+v3$ | $n1 = n2+n3$ |
| $v1 = v2-v3$ | $n1 = n2-n3$ |
| $v1 = v2\times v3$ | $n1 = n2\times n3$ |
| $v1 = v2\ /\ v3$ | $n1 = n2\ /\ n3$ |

$n1 = n2 * n3$ (remainder of $n2/n3$)
$n1 = v2$ (nearest integer)
$v1 = n2$
$v1 = n2\ /\ n3$

*Functions*

$v1 =$ MOD $v2$
$v1 =$ INT $v2$ (integer part)
$v1 =$ FRAC $v2$ (fraction part)
$v1 =$ SQRT $v2$

$$\left.\begin{array}{l} v1 = \mathrm{SIN}\ v2 \\ v1 = \mathrm{COS}\ v2 \\ v1 = \mathrm{TAN}\ v2 \\ v1 = \mathrm{CSC}\ v2 \\ v1 = \mathrm{SEC}\ v2 \\ v1 = \mathrm{COT}\ v2 \end{array}\right\}\text{(argument in radians)}$$

| | |
|---|---|
| $v1$ = ARCSIN $v2$ | (in range $-\frac{1}{2}\pi \leqslant x < \frac{1}{2}\pi$) |
| $v1$ = ARCCOS $v2$ | (in range $0 \leqslant x < \pi$) |
| $v1$ = ARCTAN $v2$ | (in range $-\frac{1}{2}\pi \leqslant x < \frac{1}{2}\pi$) |
| $v1$ = LOG $v2$ | (natural logarithm) |
| $v1$ = EXP $v2$ | ($e^x$) |
| $v1$ = EXPM $v2$ | ($e^{-x}$) |
| $n1$ = MOD $n2$ | |

Notes: INT $v2 \leqslant v2$, FRAC $v2 \geqslant 0$

*Jump Instructions*

$\rightarrow 1$ (unconditional jump)

| | |
|---|---|
| $\rightarrow 1,\ \pm v2 \geqslant \pm v3$ | $\rightarrow 1,\ \pm v2 > \pm v3$ |
| $\rightarrow 1,\ \pm v2 = \pm v3$ | $\rightarrow 1,\ \pm v2 \neq \pm v3$ |
| $\rightarrow 1,\ \pm n2 \geqslant \pm n3$ | $\rightarrow 1,\ \pm n2 > \pm n3$ |
| $\rightarrow 1,\ \pm n2 = \pm n3$ | $\rightarrow 1,\ \pm n2 \neq \pm n3$ |

$\rightarrow 1,\ \pm v2 =^{*} \pm v3$
$\rightarrow 1,\ \pm v2 =^{*} \pm v3$
[Agreement to $n0$ significant binary digits (Pegasus) or decimal digits (Sirius)]

Further information can be found in reference [**8**].

CHAPTER 7

# THE ELLIOTT 803 AUTOCODE

The basic Elliott 803 computer has a store of 4096 words on magnetic cores, but this is extendable to 8192 words; each word has 39 binary digits. Five-hole paper tape is used for input and output, but punched card input and output and magnetic film backing store can also be provided.

An automatic floating-point unit is available as an optional extra, and this will make most autocode programmes run considerably faster; the programmes themselves are written in precisely the same form in either case, but the translating programme (compiler) used will be such as to take advantage of the floating-point unit if it has been fitted.

## 7.1 Some Introductory Examples

We shall start by giving the autocode instructions needed to evaluate $\left(1+\frac{1}{n}\right)^n$ for $n = 1, 2, \ldots, m$, following the method shown in the flow diagram Fig. 2.4 of page 9:

```
   N = 0
1) N = N+1
   A = 1
   P = N
   B = 1/N
   B = 1+B
2) A = A*B
   P = P-1
   JUMP IF P>0@2
   LINE
   PRINT A
   JUMP IF N<M@1
```

Fig. 7.1. Evaluation of $\left(1+\frac{1}{n}\right)^n$ for $n = 1, 2, \ldots, m$.

It will be seen that in the Elliott autocode we use the single symbol = instead of :=, and that the multiplication sign is represented by an asterisk (viz. the instruction labelled 2 above). Each autocode instruction is written on a line of its own, and those that are destinations of *jump* instructions are labelled by means of a *reference number* followed by a right-bracket. The instruction LINE ensures that the next number to be printed starts a new line.

The arithmetic instructions of the Elliott autocode are limited to having not more than two numbers (operands) to the right of the = sign, and for this reason we had to write the two instructions

```
B = 1/N
B = 1+B
```

for the operation $b := 1+\frac{1}{n}$.

The instructions given above can however be considerably simplified by using the special facilities which the autocode provides for organizing loops, as follows:

```
CYCLE N = 1:1:M
A = 1
B = 1/N
B = 1+B
CYCLE P = N:-1:1
A = A*B
REPEAT P
LINE
PRINT A
REPEAT N
```

Fig. 7.2. Evaluation of $\left(1+\frac{1}{n}\right)^n$ for $n = 1, 2, \ldots, m$. (2nd version).

It is important to realise that to each CYCLE instruction there must be one and only one corresponding REPEAT instruction to indicate the end of the loop. The CYCLE instruction specifies the initial value, the stepping value, and the final value of the *controlled variable*, and the corresponding REPEAT instruction must again contain the name of this variable. Our example (Fig. 7.2) shows a cycle within a cycle, and the reader should note how the REPEAT instructions refer to the controlled variables in the reverse order, i.e.

$$\begin{tikzcd}
\text{CYCLE N} \arrow[r, phantom, "\ldots\ldots"] & \text{CYCLE P} \arrow[r, phantom, "\ldots\ldots"] & \text{REPEAT P} \arrow[l, bend right] \arrow[r, phantom, "\ldots\ldots"] & \text{REPEAT N} \arrow[lll, bend right]
\end{tikzcd}$$

The two sets of instructions given so far are essentially the same as regards the arithmetic operations to be carried out by the computer. However, for very large values of M the repeated multiplications would be rather time-consuming, and it would be faster to make use of logarithms. The repertoire of the autocode includes a set of basic functions (a full list is given in section 7.8), and in particular it can evaluate (natural) logarithms and exponentials. We could therefore write the procedure as follows:

```
CYCLE N = 1:1:M
B = 1/N
B = 1+B
B = LOG B
B = N*B
A = EXP B
LINE
PRINT A
REPEAT N
```

Fig. 7.3. Evaluation of $\left(1+\frac{1}{n}\right)^n$ using logarithmic function.

## 7.2 Integers and Floating-point Variables

The autocode treats numbers as being either in *integer form* or in *floating-point form*. This distinction is fundamental to the internal working of the computer, and has already been discussed in chapter 4. Any integer $n$ in the range

$$-274\,877\,906\,944 \leqslant n \leqslant 274\,877\,906\,943$$

can be represented in integer form. Numbers in floating-point form, on the other hand, can have magnitudes up to $\frac{1}{2} \times 10^{77}$, and they need not be whole numbers. They may be considered as being numbers like $0{\cdot}274\,877\,907 \times 10^{12}$ and are represented to an accuracy of between 8 and 9 significant decimals.* Numbers of magnitude less than about $\frac{1}{2} \times 10^{-77}$ are replaced by zero.

Numbers written on the programme sheet in numerical form are called *constants*. They should be written in ordinary decimal notation but should not normally have more than eleven digits altogether. Numbers in floating-point form may be written as constants thus:

5 or 5·0    99·9    −42·3    ·125    ·000124    −·125

For constants in integer form, decimal points are *not* permitted even if followed by 0's.

Numbers other than constants (as defined above) arise as a result of the operations carried out by the computer, and such numbers are called *variables*; it should be noted that this includes numbers read into the computer from data tapes by means of the READ instructions. Variables are represented by letters of the alphabet, just as in ordinary mathematical notation. However, only capital letters should be used since the teleprinter equipment which transcribes the programme has no lower case. We shall continue to use the word "number" as meaning either a constant or a variable.

The Elliott autocode allows the user complete freedom as to the choice of letters for his variables, and any variable may have a suffix attached to it (see section 7.3). The *setting instructions* SETS and SETV are used to declare which variables represent numbers in integer form and which represent floating-point variables in any particular programme.

Thus the setting instruction

SETV NAPBM

would indicate that all the variables used in the procedure of Fig. 7.1 (or of Fig. 7.2) are in floating-point form; in fact, this was assumed in writing these procedures, as we shall now explain.

The variables and constants occurring in any arithmetic instruction must *all be of the same form*, i.e. they must either all be floating-point, or else all be in integer form. *Division of numbers in integer form is not permitted*, on the grounds that it could give rise to a non-integral result. It is clear therefore that the instruction

$$B = 1/N$$

implies that both B and N must be in floating-point form and the instruction $P = N$ then forces P into the same form. A constant like 1 or 2 will automatically be interpreted as a floating-point constant if it is part of an instruction involving floating-point variables.

It may be noted that it is perfectly permissible to use floating-point variables for numbers which actually take only integral values, as has been done for P, N and M above. The question now arises as to whether this could introduce rounding errors. All integers up to $2^{29}$ (which is just over $\frac{1}{2} \times 10^{9}$) will be accurately represented in floating-point form and any arithmetic operations with such integers will give an exact result provided the result is itself an integer in the range $-2^{29}$ to $+2^{29}$.

To convert a variable I from integer form to floating-point form we use the instruction

A = STAND I

which is said to *standardize* I. It is important to realize that in this instruction the variables A and I are of an entirely different form, even though they both represent the same numerical value. We are now in a position to re-write the procedure of Fig. 7.2 so as to have N, P, and M in integer form, by simply replacing the instruction $B = 1/N$ by the two instructions

* The representation of these numbers inside the computer gives an accuracy of 28 significant binary digits, with a possible error in the 29th digit.

```
B = STAND N
B = 1/B
```

The only other instruction which allows us to change from one form of variable to the other is

```
I = INT A
```

which, when A is positive, puts I equal to the integral part of A (defined as the greatest integer not exceeding A). The autocode departs from the usual mathematical definition in the case of negative values of A, for which INT A gives "minus the integral part of the modulus of A". The variable I in the instruction I = INT A may be taken to be of either integer or floating-point form, according as to whether it is declared under SETS or SETV.

## 7.3 Suffices

The use of suffices was discussed in chapter 3, and as an example we now give the autocode instructions corresponding to Fig. 3.1 for adding the sequence of numbers $a_1, a_2, \ldots, a_n$.

```
B = 0
CYCLE I = 1:1:N
B = B+A(I)
REPEAT I
```

The autocode will also accept AI as an alternative form of writing A(I), and when the suffix is zero it may be omitted altogether, so that *A(0) and A0 denote the same variable as A*. The various forms of suffices are shown in the table below, in which the numbers 7 and 8 may be replaced by any other positive integer constants, and in which I and J denote variables in integer form (but *without* any further suffix attached to them).

| | | | | |
|---|---|---|---|---|
| A(0) | or | A0 | or | A |
| A(7) | or | A7 | | |
| A(I) | or | AI | | |
| A(I+7) | or | A(7+I) | | |
| A(I−7) | | | | |
| — | — | — | | |
| A(7−I) | | | | |
| A(I+J) | and | A(I−J) | | |
| A(7I) | | | | |
| A(7I+8) | or | A(8+7I) | | |
| A(7I−8) | and | A(8−7I) | | |
| A(7I+J) | or | A(J+7I) | | |
| A(7I−J) | and | A(J−7I) | | |

Fig. 7.4. Table of allowed forms of suffices.

The forms of suffix shown in the first five lines of this table are known as *simple suffices*, and those in the first two lines are *numerical suffices*; we shall need to refer to this classification in connection with some restrictions on the use of suffices (e.g. section 7.10).

Variables occurring in a suffix must be in integer form, and suffices should not be allowed to take negative values. The reader should note that the notation for multiplication inside a suffix, as in A(7I) or A(7I+J), differs from the notation in an arithmetic instruction such as K = 7*I, where the asterisk must be used. Further, the product of two variables, such as I*J, is *not* permitted as a suffix.

As another example, we give the autocode instructions for evaluating the polynomial

$$p = a_0x^n+a_1x^{n-1}+\ldots+a_{n-1}x+a_n$$

by the method of nested multiplication discussed in connection with Fig. 3.2 on page 15. We assume that the variables X, A, A1, . . . A(N), and N have the values of $x$, $a_0$, $a_1$, . . . , $a_n$, and $n$ respectively, and that N is in integer form (and not less than 1):

```
P = A
CYCLE I = 1:1:N
P = P*X
P = P+AI
REPEAT I
```

## 7.4 Setting Instructions, START and STOP

During the process of "translation" the autocode compiler must assign storage locations to all the variables, and to do this it needs to know the range of suffices that will be required with each of the letters used. This information must be supplied at the beginning of a programme in the *setting instructions*; these also define which variables are to be in integer form and which in floating-point form.

Thus the setting instructions for the procedure for adding the sequence of numbers $a_1, a_2, \ldots, a_n$ (cf. page 45) might be

```
SETS IN
SETV A(200)B
```

which indicates that the variables I and N following SETS are in integer form, that the variables A and B following SETV are in floating-point form, and that 201 storage locations are to be reserved for the sequence

A, A1, A2, . . . , A200

all of which will be in floating-point form. Note that the setting instructions have to give the maximum numerical value taken by each suffix, and that this value (if it is not zero) must be put in brackets. Errors due to allowing suffices to become larger than indicated in the setting instructions will not be detected by the autocode. If a programme with the above setting instructions were to call for B2, this would be interpreted as A1, while A201 would be interpreted as N. This will cause the programme to behave in unexpected ways.

There is also a setting instruction SETF which lists the basic functions used in the programme (see section 7.8), and finally there is the SETR instruction which gives the maximum reference number attached to an instruction. The SETR instruction must always be given as the last of the setting instructions.

At the end of any complete autocode programme there must be something to tell the compiler that the process of translation is complete. This is done by means of the word START, followed by the reference number of the instruction at which the programme is to be entered when the translated programme is "run". Note that the reference number 0 is not allowed.

Many autocode programmes end with

```
STOP
START 1
```

The instruction STOP will be translated into one which, during the running of the translated programme, will stop (i.e. end) the computation and give a high pitched note on the loudspeaker. START 1 is not translated as an instruction of the programme.

## 7.5 Input

Most programmes are designed so as to work with parameters or other numerical data supplied on a separate *data tape*. The instruction READ A causes the next *number* on the tape in the tape-reader to be read and its value to be assigned to the variable A. Such a number will consist of a sequence of decimal digits (with or without a decimal point) terminated *either* by carriage return and line feed *or* by two or more space symbols. These alternative forms for terminating numbers make it possible to have data tapes printed out either in the form of a single column or as several tabulated columns. Single spaces are ignored by the READ instruction, so that we may arrange digits in groups for ease of reading, as in 1 000 000.

The READ instruction provides three further facilities:

(i) *label:** In order to identify the data being read, we may put a *label* on a data tape in the form of a set of (punched) characters which are copied directly on to the output tape as the data tape is read. Such a label must be preceded by an = sign and terminated by the character *bl*.† When an instruction like READ A encounters the = sign, it will cause all characters up to the next *bl* to be copied on to the output tape and will then go on to read the next number on the data tape and make A equal to this number.

(ii) *trigger:* It is often convenient to have a device to indicate the end of a sequence of numbers on a data tape, and to cause the programme to jump to another instruction. This may be done by placing a *trigger*, consisting of an integer constant followed by a left-bracket, on the data tape. When a READ instruction encounters a trigger such as 5(, it causes a jump to the instruction with reference number 5.

(iii) *stop:* When a READ instruction encounters a right-bracket on a data tape it causes the computer to stop in the same way as a STOP instruction.

*Conventions for punching numbers on data tapes.* Negative numbers are preceded by a minus sign, but in the case of positive numbers the plus sign is optional. Not more than eleven‡ decimal digits may be punched for each number, and the end of a number must be indicated either by the characters *Cr Lf* or by *Sp Sp*, as explained at the beginning of this section. As an alternative to the ordinary decimal notation we may also punch the number represented by $a \times 10^p$ in the form $a/p$, so that 0·00000001 could be punched as ·1/−7 or as 1/−8. Note that floating-point numbers may be put in this form only on a data tape; on a programme sheet the oblique stroke would denote division.

Whereas the READ instruction reads numbers together with their terminating characters, there is also an instruction INPUT I which will read a single character from the tape and set the integer I equal to the numerical value of this character as defined in the Elliott tape code (see Appendix).

## 7.6 Output

The standard output instructions cause characters to be punched on the output tape, and this tape may subsequently be printed out by a teleprinter while the computer is doing another job. However, we shall find it convenient to *describe* the output instructions in terms of what is finally printed by the teleprinter.

A PRINT instruction is used to print the numerical value of a variable, and this instruction also specifies how many decimal places are to be printed. In all cases negative numbers are preceded by a minus sign, whereas for positive numbers a space is printed instead of a plus. Non-significant leading zeros are also replaced by spaces, unless they come after a decimal point. All numbers are

* In the Elliott autocode specification, "label" does not mean "reference number".
† Blank tape, i.e. no holes, which in Elliott code is not the figure-shift character.
‡ But integers up to 274 877 906 943 are accepted.

followed by two spaces, so that output and input tapes are compatible, enabling us to use an output tape directly as the data tape for another programme.

For variables in floating-point form there are three different kinds of style available, as illustrated by the following instructions. In each case we shall show how a number equal to $-\pi$ would be printed. Round-off on the last decimal place printed is automatic.

PRINT A,8 prints A with eight digits, the decimal point being placed in the appropriate position, as in −3·1415927.

PRINT A,2:6 prints A with two digits in front of and six digits after the decimal point, as in − 3·141593.

PRINT A,5/ prints A as a fraction to five decimal places (starting with a decimal point), followed by an oblique stroke and a decimal exponent (given to two digits), as in −·31416/ 01.

For variables in integer form there is only one kind of style, as illustrated by PRINT N,5 which will print N as five decimal digits; if necessary, spaces will be printed in front of the number to make up for missing digits, thus enabling us to tabulate results in columns.

The number of decimal places to be printed may also be controlled by the programme itself in instructions of the form

```
PRINT A,I
PRINT A,I:J
PRINT A,I/
```

in which I and J are variables in integer form.

It may happen that the number to be printed is too large to be represented in the specified style; e.g. the instruction PRINT A,2:1 is not appropriate for printing the number 103·4. In such cases the printing will move to a new line on the page and print a question mark followed by the output corresponding to PRINT A,9/ or, in the case of variables in integer form, that corresponding to PRINT A,12.

If no style is specified, the instruction PRINT A will adopt the style used for the variable of the same form (i.e. integer or floating-point) last printed. If no such variable has been printed the style adopted is PRINT A,4 for integer form, or PRINT A,9/ for floating-point form.

The following instructions are used to control the layout of results on the printed page (which allows up to 68 characters per line):

| | |
|---|---|
| LINE | causes the printer to begin a new line, |
| LINES N | causes the printer to begin N lines further on, |
| SPACES N | causes the printer to move N spaces to the right, |

where N may be either a constant or a variable in integer form, but not less than one.

It is useful to incorporate a title to identify the programme, and subtitles to act as headings for particular results or columns of results. This can be done by the instruction TITLE, followed by a space and then the heading to be printed, and terminated by the symbol *bl* for "blank". At run-time this causes everything between the space and the blank to be punched on to the output tape, with the character for figure shift added at the end.

Finally there is the instruction OUTPUT N to punch just one character, namely the one whose "numerical value" in the tape code (see Appendix) corresponds to the value of N, where N may be a constant or a variable in integer form having at most a numerical suffix. Printing is in terms of figure shift characters unless a letter shift has been punched by means of OUTPUT 31. Such a letter shift would need to be cancelled by means of OUTPUT 27 before the next PRINT instruction is obeyed, unless a TITLE instruction intervenes. To run out blank tape we use OUTPUT 0 in a cycle.

## 7.7 A Complete Programme

We shall now give a programme for estimating the mean and standard deviation of a set of numbers given on a data tape. The method is similar to that discussed in connection with Fig. 2.9 on page 12, but we do not now require the number *n* for the size of the sample to be given on the data tape. Instead, we assume that the set of numbers to be read is terminated by the trigger

4(

which is punched at the end of the data tape, and we shall count the numbers as we read them in.

```
   SETS N
   SETV ABCP
   SETF SQRT
   SETR 4
1) N = 0
   B = 0
   C = 0
2) READ A
   B = B+A
   A = A*A
   C = C+A
   N = N+1
   JUMP @2
4) LINE
   TITLE N = bl
   PRINT N,4
   P = STAND N
   A = B/P
   LINE
   TITLE MEAN = bl
   PRINT A,6
   A = A*B
   A = C-A
   P = P-1
   A = A/P
   A = SQRT A
   LINE
   TITLE DEVIATION = bl
   PRINT A,6
   WAIT
   JUMP @1
   START 1
```

Fig. 7.5. Mean and standard deviation.

When all the numbers making up the sample have been read, the trigger 4(, which terminates the data, causes a jump to the instruction with reference number 4, and the programme then calculates and prints the sample size, mean, and standard deviation.

The instruction WAIT causes the computer to stop and emit an audible hoot until a certain button is pressed on the keyboard; subsequently the programme continues with the next instruction, which in our example is JUMP @1. The purpose of this is to give us time to place a new data tape in the tape reader and to carry out the same calculation on a further sample. By using the label facility (see section 7.5) on the data tapes, we can arrange for the name of each sample to be printed out just ahead of the results calculated from it.

As autocode programmes are not self-starting, it is not necessary to provide a WAIT instruction for putting in the first data tape. To start the programme, the operator has to press certain buttons on the keyboard. In the same way, whenever the computer has come to a STOP or WAIT, the programme may be re-entered manually at any instruction which has a reference number.

## 7.8 Summary of Arithmetic and Function Instructions

Basically there are only five arithmetic operations in the autocode, as shown in the following instructions:

| | | |
|---|---|---|
| | A = B | A = −B |
| | A = B+C | A = −B+C |
| | A = B−C | A = −B−C |
| | A = B*C | A = −B*C |
| floating-point only: | A = B/C | A = −B/C |

The expressions on the right of the = sign never involve more than two numbers, but one or both of these may be a constant. The instructions will make the variable on the left of the = sign take the value of the expression on the right. The variables and constants in such an instruction must all be of the same form, i.e. they must either all be in floating-point or else all be in integer form. Division of numbers in integer form is not allowed. The variables may have any kind of suffix, and for this reason the operation of multiplication is denoted by an asterisk in an arithmetic instruction, e.g. A = B*C.

The autocode includes facilities for evaluating functions by means of the following instructions:

| | |
|---|---|
| A = LOG X | natural logarithm to the base *e* |
| A = EXP X | exponential |
| A = SQRT X | square root |
| A = SIN X<br>A = COS X<br>A = TAN X | where X must be in units of 180°, so that to find the sine of an angle of D degrees we write:<br>X = D/180<br>A = SIN X |
| A = ARCTAN X | evaluates A in units of 180° |
| B = INT X | integral part as defined in section 7.2 |
| A = FRAC X | fractional part, equal to "X minus INT X" |
| B = MOD C | modulus (absolute value) |
| A = STAND I | standardization of an integer as defined in section 7.2 |

In this table of function instructions, A and X must both be in floating-point form; the variable B in B = INT X and in B = MOD C may be of either integer or floating-point form, but in B = MOD C we must have B and C both of the same form. The variables may have any kind of suffix, but the arguments (X, C, or I) *cannot* be replaced by an arithmetic expression, not even by −X. However, the forms

A = −LOG X

and so on are permitted, and the arguments may be replaced by constants (positive or negative).

The functions used should normally be listed under SETF, but the abbreviation TRIG should be used to cover any or all of SIN, COS, and TAN. The functions MOD and STAND need not be included in the SETF list.

If, when the programme is run, the argument X in A = SQRT X or in A = LOG X has a negative *value*, then the computer will give an error indication in the form of a continuous output of the symbol 5 or 2 on the output punch. Other error indications will be given when the numbers to be calculated lie outside the range represented in the computer, or when a division by zero is attempted.

## 7.9 Jump Instructions and Subroutines

The simplest jump instructions are unconditional, such as

```
JUMP @3   or   JUMP @K
```

which means jump to the instruction whose reference number is 3 (or is equal to the *current value* of K). In the instruction JUMP @K the variable K must be of integer form, and is *not* allowed to have a suffix; this instruction may be used as a switch, as explained in chapter 3.

The autocode provides a great variety of conditional jump instructions. Thus

```
JUMP IF A = B @3
```

has the effect of a jump if A = B, and is ignored if A ≠ B. We also have the opposite instruction

```
JUMP UNLESS A = B @3
```

which jumps if A ≠ B and is ignored if A = B.

In these jump instructions the condition A = B looks formally the same as an arithmetic instruction. In fact the autocode allows jump instructions with conditions in the form of any of the arithmetic instructions or function instructions listed in the previous section; note that this implies that the equal sign in the condition must be preceded by just a single variable (*not* a constant). However, jump instructions test for equality without actually changing the variable on the left side. Tests for equality should not be used on floating-point variables, since these are subject to rounding errors as explained in chapter 4.

The autocode also allows tests for inequality with the symbols < or > which may be substituted for the = sign in any of the forms discussed. However, the symbols ⩽, ≠, ⩾ are not available in the autocode, and it is for this reason that the instruction JUMP UNLESS is sometimes useful. Thus we can write

```
JUMP UNLESS A>B+C @3
```

to have the effect of jump if A⩽B+C.

The following are just a few examples of permissible jump instructions:

```
JUMP IF A>SQRT B @4
JUMP IF J<MOD I @8
JUMP UNLESS J = 3*I @K
JUMP IF A(7I+J)<-BJ-C(7I) @K
```

Where the destination of the jump is specified by a variable, like K, this must be in integer form and is *not* allowed to have a suffix.

The reference numbers used as the destinations of jumps may be any unsigned integers other than zero, and they may appear in any sequence in the programme. The largest such integer used must be stated in SETR, but it is not necessary to use all the integers up to this value as reference numbers in the programme.

A *subroutine* is a sequence of instructions which may be used at several points of a programme, but is stored only once. It is often convenient to write a programme in the form of a *main programme* which "calls in" one or more subroutines to do specific jobs. The use of a subroutine involves two jumps, one to jump from the main programme to the beginning of the subroutine, and the other to jump from the end of the subroutine back to the point where the main programme is to continue. For the first of these jumps the autocode instruction

SUBR K

is used, where K may be a constant or a variable in integer form, but must not have a suffix. This will cause a jump to the subroutine which starts with the instruction whose reference number corresponds to the *value* of K. The last instruction to be obeyed in such a subroutine must be followed by the instruction

EXIT

which will cause a jump back to the instruction immediately after the last SUBR instruction to have been obeyed.

```
   SETS I
   SETV NMQR
   SETF FRAC
   SETR 6
1) TITLE FACTORIZE bl
   CYCLE I = 4:1:500
   LINES 2
   PRINT I, 3
   TITLE = bl
   N = STAND I
   M = 2
   SUBR 5
   M = 3
2) SUBR 5
   JUMP IF N = 1 @4
   JUMP IF Q<M @3
   M = M+2
   JUMP @2
3) PRINT N,3:0
4) REPEAT I
   STOP
5) Q = N/M
   R = FRAC Q
   JUMP IF R>0 @6
   PRINT M,3:0
   N = Q
   JUMP IF N>1 @5
6) EXIT
   START 1
```

Fig. 7.6. Factor table for integers up to 500.

A subroutine must always be left via an EXIT instruction; if it has several branches, these may be terminated by separate EXIT instructions. The autocode allows subroutines within subroutines up to six in depth. If this depth is exceeded, an error indication will be given when the programme is run.

The use of a subroutine is illustrated by the programme for factorizing integers in Fig. 7.6, which uses the method discussed in connection with Fig. 3.10 on page 20.

## 7.10 Cycle Instructions

The most general form of a "Type 1" cycle instruction is

```
CYCLE A = B:C:D
.
.
.
REPEAT A
```

where the *controlled variable* A may have any kind of suffix, but the variables B, C, D on the right may have simple suffices only. Any of the variables B, C, D may of course be replaced by constants, or by $-B$, $-C$, or $-D$.

Some examples of the use of such cycle instructions were given in section 7.1. The precise definition of these CYCLE instructions depends on whether the controlled variable A is in integer or in floating-point form.

If A is in integer form, then of course B, C, and D must also be in integer form and such that $(D-B)/C$ is a non-negative integer. The loop (i.e. the instructions between the CYCLE and the REPEAT) will be performed $1+(D-B)/C$ times, corresponding to the sequence of values $A = B$, $B+C$, $B+2C, \ldots, D$, and the programme then continues with the instruction immediately after the REPEAT. If the condition on $(D-B)/C$ is not satisfied the cycle continues indefinitely.

Similarly, if A is in floating-point form, then B, C, and D must also be in floating-point form, and the intention of the programme will be to perform the loop for the sequence of values

$$A = B,\ B+C,\ B+2C, \ldots \text{ as far as } D.$$

However, owing to round-off errors in floating-point arithmetic, the sequence of values of A calculated in the computer might not have any term that is exactly equal to D. For this reason the CYCLE instruction will identify the term in the sequence nearest to D and, on reaching this term, will replace it by the exact value of D.

To perform a loop for the N values of the arithmetic progression

$$A = B,\ B+C,\ B+2C, \ldots, B+(N-1)C$$

we may use the VARY instruction

```
VARY A = B:C:N
.
.
.
REPEAT A
```

in which N must always be a positive integer (not zero), either a constant or a variable in integer form. Because the VARY instruction does not need to make any "end-adjustment", it will be somewhat faster than the corresponding CYCLE instruction when A is in floating-point form.

It is not advisable to change the values of A, C, or D inside a loop; for details of the effects of doing this the reader should consult the manufacturers' specification [7].

If the values to be taken by the controlled variable do not form an arithmetic progression, we may use a "Type 2" cycle instruction in which we give an explicit list of the values to be taken by the controlled variable, e.g.

```
CYCLE A = 11, 13, 17, 19, 23, 29
      .
      .
      .
REPEAT A
```

We may have cycles within cycles to a combined depth of five, a VARY instruction counting as a cycle. If the autocode programme is compiled on to tape (two-pass) instead of into the store (load-and-go), then the "Type 1" CYCLE instructions are exempt from this restriction, but the combined depth of "Type 2" CYCLE and of VARY instructions must not exceed five.

## 7.11 Checking Facilities

The purpose of writing an autocode programme is to run it on the computer. This is done in two stages. In the first stage, called *translation*, the autocode programme is translated automatically into an equivalent machine-code programme, which may either be stored ready to be obeyed (*load-and-go*) or punched out on tape to be fed into the computer subsequently in stage two (*two-pass*). During the first stage the computer reads the autocode programme one instruction at a time, and translation will stop if the computer detects any clerical error or inconsistency in the autocode programme. Such errors must then be corrected and the whole process of translation started again.

In stage two the translated programme is *run*, i.e. the computer executes the instructions of this programme, and this may show up further errors. To help in detecting such errors the autocode provides two further facilities, called *check* and *trace* respectively.

The instruction CHECK A provides an optional print facility, which will print the value of A if the "B-button" on the keyboard is down both in translation and in running. The printing will be in the style of PRINT A,9/ for floating-point variables and of PRINT A,12 for variables in integer form. To distinguish this optional printing, each value is printed at the beginning of a new line, preceded by an asterisk.

The trace facility is also controlled by the setting of a certain button on the keyboard, and may be used to study the sequence in which groups of instructions in the programme are actually being obeyed. It will print out the reference number of any numbered instruction whenever such an instruction is obeyed, each reference number being printed on a new line and followed by a right-bracket. In the case of cycles, the reference number (if any) of the CYCLE or VARY instruction will be printed only when entering it for the first loop, while the reference number (if any) of the REPEAT instruction will be printed at the end of each loop performed.

## Conclusion

The account given in this chapter is based on the specification of the Mark 3 Autocode published in July 1962. This autocode also contains instructions for controlling a second input/output channel and a magnetic film backing store which may be fitted as optional extras to the 803 computer. There are also facilities for incorporating blocks of machine code in an autocode programme. For details of these additional instructions and facilities we refer the reader to the manufacturers' specification [7].

Finally it should be pointed out that there is a Library of standard programmes, some of which may be incorporated in an autocode programme as subroutines, while others are complete programmes in themselves.

CHAPTER 8

# MERCURY AUTOCODE

**Introduction**

Mercury is a computer designed at Manchester University and built by Ferranti Ltd. It was one of the first British computers to have floating-point arithmetic built into its arithmetic unit, and it was given a larger immediate-access store than earlier machines. Thus it became the first British computer for which an efficient autocode compiler could be written. Most Mercury programmes are written in autocode, and it is normally reckoned that a programme written in Mercury autocode will take less than twice the time of the most efficient programme in machine language for the same job.

The original autocode compiler for Mercury was written by a group at Manchester University, led by Dr. R. A. Brooker. Since then a number of computer centres have made their own additions to the compiler, with the unfortunate result that there is no standard Mercury autocode. The version described in this chapter, known as CHLF 3,* is used on a number of Mercury computers, and will also be available on Atlas.

In Mercury autocode, calculations can be carried out on two types of numbers called *variables* and *indices*. Variables are normally used, as they can take any value up to $6\times10^{76}$ in modulus, and their accuracy can be 9 significant (decimal) figures. The value of an index, on the other hand, can only be a small integer in the range from $-512$ to $+511$. Indices are therefore used almost exclusively for counting.

**8.1 Indices**

There are 24 indices in the computing store of Mercury. Their names are

|  |  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|
| I | J | K | L | M | N | O | P | Q | R | S | T |
| I′ | J′ | K′ | L′ | M′ | N′ | O′ | P′ | Q′ | R′ | S′ | T′ |

An index† never has a suffix (section 8.2).

Typical instructions carrying out arithmetic on indices are

```
I = J
K = L-2
R = S+T-MN+13
M = SPQR+ST-2N
N = N+1
```

Such instructions replace the value of the index on the left of the = sign by the expression on the right: they do not change the values of the indices appearing on the right. It will be seen that = is the symbol for "becomes", replacing the combination := used in Part I.

*The initials stand for C.E.R.N. Geneva, A.E.R.E. Harwell, London University, and R.A.E. Farnborough: the 3 refers to the number of magnetic drums.

† A useful mnemonic is IT. Letters in this range denote indices: outside this range they denote variables.

The name *index expression* is given to the right-hand side of an equation which sets the value of an index. Certain restrictions apply to it.

(1) Multiplication signs are omitted.
(2) No division is possible and the division sign / is forbidden.
(3) Brackets and powers are not allowed.
(4) Variables must not be used.
(5) Constants may appear, but they must be integers.
(6) Any number of terms may appear on the right-hand side so long as the teleprinter can print it on one line (a limit of 68 printed characters).

Arithmetic with indices is exact. It is for this reason that the use of division, variables and non-integral constants is not allowed.*

The result of an index calculation must lie between $-512$ and $+511$. If it does not, the computer gives a wrong result within this range, differing from the correct result by a multiple of 1024. Thus if $N = 30$, the instruction $M = NN$ sets $M = 900-1024 = -124$.

## 8.2 Variables

Arithmetic with variables is always rounded.† The method of rounding is a little unusual, and has the unfortunate consequence that arithmetic with exact numbers gives an inexact result. For instance, the sum of 4 and 7 turns out to be $11+2^{-25}$, and the product of 4 and 7 becomes $28+3\times 2^{-24}$. However, most calculations are carried out with numbers which are stored to the full accuracy of the machine and have already been rounded. Results derived from them will also need to be rounded, and the method used does this successfully.

509 variables are provided in the computing store‡ of Mercury, the 29 *special variables* and 480 *main variables*. The special variables, which are always available to the programmer, are denoted by

$$\begin{array}{ccccccccccccccc} A & B & C & D & E & F & G & H & U & V & W & X & Y & Z & \pi \\ A' & B' & C' & D' & E' & F' & G' & H' & U' & V' & W' & X' & Y' & Z' & \end{array}$$

The machine sets the value of $\pi$ to be 3·14159265 initially: it can be changed by programme, but it will return to this value each time a change of chapter (section 8.12) occurs. There is no variable $\pi'$.

The names of the main variables consist of a letter and a suffix, e.g. $A_{12}$. The letter can be any one of the fifteen letters for variables, A B C D E F G H U V W X Y Z or $\pi$; the suffix is normally written on the same line as the letter, e.g. A12, because the teleprinters used in preparing programmes cannot print suffices below the line. The names of main variables have to be declared at the head of a programme by the *main-variable directives* described in section 8.14.

In the written programme the suffix of a main variable may include an index. Suffices are limited to the following forms:

| | |
|---|---|
| Integer | e.g. A12, |
| Index | e.g. AI, |
| (Index ± integer) | e.g. A(I+12) and A(I−12). |

* In Mercury division never gives an exact result, so an equation like $T = 6/R$ could not be relied upon to set $T = 3$ if $R = 2$.

† But see page 59 for the use of the ≈ symbol.

‡ The magnetic drums provide storage for a large number of auxiliary variables, described in section 8.13.

C

Brackets must enclose the compound suffix in the last case, and the index must precede the integer. It will be seen that the suffix cannot have the complexity allowed for an index expression: even A(I+J) and A(−I+12) are forbidden.

The effect of an instruction using a main variable such as A(I+12) depends on the current value of the index I on each occasion it is obeyed. If I = 10 when the instruction

$$A(I+7) = BI+D(I-7)$$

is obeyed, the effect is the same as

$$A17 = B10+D3.$$

Practical applications of this technique are described in section 8.6.

Special and main variables are separate entities, so that X and X0, for instance, are different variables. But the effect of including a variable in any instruction is the same whether a special or main variable is used. The special variables are normally used for individual numbers, and the main variables for sets of numbers. If, for example, a table of some function had to be calculated for a range of values of the argument, the argument could be stored as a special variable, perhaps X, and the function values in a set of main variables, such as F0, F1, etc.

Typical instructions which set variables are

```
X = X0
C = D−E+J          (even though J is an index)
F = GH−3·519W(K+27)+13/X12
B17 = −7·2IASW3/Y(J+2)+3·751JJXX'X0/I+T−9
X = X+H
```

Only the variable on the left of the = sign in any instruction is changed. The name *general expression* will be given to the right-hand side of an instruction setting a variable. Certain restrictions apply to it.

(1) Multiplication signs are omitted. In each product of several items, the sequence should be number, indices, variables. Otherwise ambiguities could arise because suffices are written on the same line as the variable to which they are attached. The convention in Mercury autocode is that a number or index following a main-variable letter is a suffix to that letter; thus A12Y means (A12) × Y, not A × 12 × Y.*

(2) The division sign / is allowed, but only one item, whether a constant, index or variable, may follow it. If a complicated denominator is needed, it should be calculated in an earlier instruction, or else the ψDIVIDE instruction (see section 8.3) should be used.

(3) Powers are not allowed and the use of brackets is reserved for compound suffices of main variables. Thus $X = W(Y+Z^3)$ must be written out as X = WY+WZZZ.

(4) Constants may appear in any term on the right-hand side. (Their use as the denominator in a division should be avoided: it is quicker to multiply by the reciprocal.)

(5) Indices may appear as numbers in a general expression (whereas variables may not appear in an index expression).

(6) The size of a general expression is limited by the length of one line on the teleprinter, 68 printed characters.

* ALGOL avoids this confusion by insisting on multiplication signs and putting all suffices in square brackets, e.g. A[12] × Y.

*Example.* Given $a$ and $x$, calculate $y = \frac{1}{2}\left(x+\frac{a}{x}\right)$.

One instruction is sufficient:

```
Y = 0·5X+0·5A/X.
```

Note that we cannot write A/2X on account of rule (2) above.

*Example.* Calculate $y = (1+0{\cdot}2507213x+0{\cdot}0292732x^2+0{\cdot}0038278x^3)^{-4}$.
(This can be used as an approximation to $e^{-x}$.)

```
Y = 0·0038278X+0·0292732
Y = YX+0·2507213
Y = YX+1
Y = YY
Y = YY
Y = 1/Y
```

Fig. 8.1.

The programme of Fig. 8.1 shows three ways in which greater efficiency in calculations can be obtained.

(i) The apparently shorter programme

```
Y = 0·0038278XXX+0·0292372XX+0·2507213X+1
Y = YYYY
Y = 1/Y
```

takes longer to obey and needs more space in the store of the computer to hold it, largely because it carries out 9 multiplications instead of 5.

(ii) Division is a slow process—the instruction Y = 1/Y takes about 3 milliseconds to carry out, as long as the previous five instructions in Fig. 8.1—and so the number of divisions should be reduced to a minimum.

(iii) In formulae like Y = YX+1 or Y = Y/X+B, which add products (or quotients) and single items, it is better to place the single items at the end, because the compiler then provides a more efficient translation.

*Exact Arithmetic with Variables*

The values of variables cannot be stored exactly unless they are integers, and arithmetic with variables is normally rounded. However, if the sign $\approx$ replaces = in any instruction which evaluates a general expression, unrounded arithmetic will be used. This facility enables us to obtain exact results for addition, subtraction and multiplication (but not division) of variables which represent whole numbers exactly.

## 8.3 Functions

So far we have broken down calculations into series of simple arithmetical instructions. It would be inconvenient to have to do this if the expressions to be calculated involved functions such as exponentials, sines or square roots. The functions provided in Mercury autocode for setting a variable are shown in Table 8.1.

| | | |
|---|---|---|
| $Y = \psi$ SQRT(V) | $+\sqrt{V}$, provided $V \geqslant 0$ | Note 1. |
| $Y = \psi$ LOG(V) | $\log_e V$, provided $V > 0$ | Note 1. |
| $Y = \psi$ EXP(V) | exp V, provided $V < 177$. | Note 1. |
| $Y = \psi$ SIN(V) | sin V. | |
| $Y = \psi$ COS(V) | cos V. | |
| $Y = \psi$ TAN(V) | tan V. | |
| $Y = \psi$ MOD(V) | $\lvert V \rvert$, i.e. $+V$ if $V \geqslant 0$, $-V$ if $V < 0$. | |
| $Y = \psi$ SIGN(V) | $+1$ if $V \geqslant 0$, $-1$ if $V < 0$. | |
| $Y = \psi$ INTPT(V) | [V]; i.e. nearest integer $\leqslant$ V. | Note 2. |
| $Y = \psi$ FRPT(V) | $V - [V]$ | Note 2. |
| $Y = \psi$ DIVIDE(U,V) | $\dfrac{U}{V}$ | Note 3. |
| $Y = \psi$ ARCTAN(U,V) | $\tan^{-1}\left(\dfrac{V}{U}\right)$ | Note 4. |
| $Y = \psi$ RADIUS(U,V) | $+\sqrt{(U^2+V^2)}$ | |
| $Y = \psi$ POLY(V)AI,N | $A_I + A_{I+1}V + A_{I+2}V^2 + \ldots + A_{I+N}V^N$ | Note 5. |
| $Y = \psi$ PARITY(S) | $+1$ if S is even, $-1$ if S is odd. | Note 6. |

Y can be replaced by any variable.
U,V can be replaced by any general expression.

Table 8.1. Basic functions in Mercury autocode.

*Notes.*

1. See section 8.14 for the action taken if an impossible value is given to V.
2. If $V = +2{\cdot}6$, $\psi$ INTPT(V) $= +2$, $\psi$ FRPT(V) $= +0{\cdot}6$
   If $V = -2{\cdot}6$, $\psi$ INTPT(V) $= -3$, $\psi$ FRPT(V) $= +0{\cdot}4$
   If $V = +3$, $\psi$ INTPT(V) $= +3$, $\psi$ FRPT(V) $= 0$.
3. $\psi$ DIVIDE can be used where the denominator is a general expression, whereas the division symbol / can have a denominator of one item only.
4. The argument is $\dfrac{V}{U}$, not $\dfrac{U}{V}$. The result Y lies in the quadrant for which sin Y, cos Y have the same signs as U, V respectively: $-\pi < Y \leqslant +\pi$. ($\psi$ ARCTAN is arranged to give the correct argument of the complex number $U+iV$ and $\psi$ RADIUS its modulus.)
5. $Y = \psi$ POLY(V)AI,N evaluates the polynomial of degree N whose coefficients are $A_I$ (the constant term), $A_{I+1}, A_{I+2}, \ldots A_{I+N}$ (coefficient of the highest power), for the argument V. N can be replaced by any index or integer, $A_I$ by any main variable: compound suffices, e.g. B(I+3), are allowed.
6. S can be replaced by any index expression.

It is important to reproduce the spelling and punctuation of Table 8.1 exactly.* Any mistake will give trouble to the autocode compiler, which may refuse to translate the instruction or, worse still, translate it incorrectly.

An equation to set the value of a variable can have either a function or a general expression on the right-hand side, but not both. The programme to calculate $y = a \cos x + b \sin x$ must be

* Except that spaces may be inserted anywhere, e.g. $Y = \psi$ ARC TAN (U,V).

```
Y = ψ COS(X)
Z = ψ SIN(X)
Y = AY+BZ
```

*Example.* Calculate $y = \frac{2}{\sqrt{3}} \tan^{-1}\left(\frac{1+2\sqrt{x}}{\sqrt{3}}\right) - \frac{1}{3} \log(1+\sqrt{x}+x)$.

```
Z = ψ SQRT(X)
Y = ψ LOG(1+Z+X)
Z = ψ ARCTAN(1·732050808,2Z+1)
Y = 1·1547005384Z−0·3333333333Y
```

Note that this programme introduces only the one variable Z as "working space". This illustrates how the amount of working space used in any calculation may be kept down by using the place for the final result to hold some of the intermediate results as well, and also by replacing one intermediate result by another, as soon as the first result is no longer required.

There is one function which sets the value of an index from a variable. This is

```
I = ψ INTPT(V)
```

where I can be replaced by any index, V by any general expression. The need for this arises because variables cannot be used in index expressions. If, for example, the value of the variable π3 is known to be an integer, and some loop of instructions is to be obeyed 2π3 times, the instruction

```
N = ψ INTPT(2π3)
```

enables us to set the index N to this value for purposes of counting.

Two functions ψ MAX and ψ MIN, which find the largest and smallest of a set of numbers, are described in section 8.6.

## 8.4 Labels and Jumps

In Part I we explained the need for jumps in our sequences of instructions and for labels marking the instructions which jumps lead to. In Mercury autocode, instructions can have a *label* attached by having a number (up to 127*) and a right-bracket symbol written before the instruction; e.g. to attach label 21 to an instruction, we write

```
21) X0 = X0+H.
```

The labels may appear in any order in the programme; if ten labels are used it is not necessary that they should be 1, 2, 3, . . . 10 in this sequence, nor indeed that the first ten numbers should be used. Obviously we must not affix the same label to two different instructions, but one instruction can have several different labels.

To jump unconditionally to the instruction labelled 21 we write

```
JUMP 21
```

(A right-bracket must not be written after the label number.) If the jump is to be made only if a condition is satisfied, we write a comma after the label number, followed by the condition: e.g.

```
JUMP 21, A⩾B0.
```

There are a number of restrictions on the condition that may be written in such an instruction.

(1) The comparison sign may be $\geqslant$, $>$, $=$ or $\neq$. ($<$ and $\leqslant$ signs are not provided in the teleprinter code.)

(2) Only a single item may be written on each side of the comparison sign; arithmetic expressions are not allowed.

* Labels 0 and 100 have special uses; see section 8.14.

(3) The comparison should always be between like items, variable with variable (or constant), index with index (or integer).

Thus, of the following conditional jump instructions, the first five are permissible and the last three are not.

```
JUMP 1,   A12⩾B3
JUMP 23,  C = X(I+5)
JUMP 45,  2·718>V7
JUMP 79,  I ≠ J
JUMP 15,  T' = 27
.................................
JUMP 37,  W⩾K            }
JUMP 60,  K>3·5          }  not allowed.
JUMP 99,  W0+W1 = 12·3   }
```

The second instruction is allowed because X(I+5) is a single variable with a compound suffix (see section 8.2).

It is dangerous to expect two variables to be exactly equal, because of the rounding errors incurred in calculating them, as is explained in chapter 4. For practical purposes, therefore, only the ⩾ sign* should be used in comparing variables.

*Example.* Calculate the square root of A.

The flow diagram of Fig. 2.8 is coded as follows:

```
   Z = 1
7) Y = Z
   Z = 0·5Z+0·5A/Z
   Y = ψ MOD(Y−Z)
   E = 0·00000001Z
   JUMP 7, Y⩾E
```

*Example.* Calculate $\left(1+\frac{1}{n}\right)^n$ for $n = 1,2,3 \ldots m$ without using logarithms.

We assume that the value of *m* is already known.

```
   N = 0
1) N = N+1
   A = 1
   B = 1+1/N
   P = N
2) A = AB
   P = P−1
   JUMP 2, P ≠ 0
   PRINT(A)1,8        (Explained in section 8.10)
   JUMP 1, N ≠ M
```

Fig. 8.2. Calculation of $\left(1+\frac{1}{n}\right)^n$—first programme.

## 8.5 Cycles

In the outer loop of the above example, N increases in steps of 1 until it reaches the value M; in the inner loop P decreases from N in steps of 1 until it reaches the value 1. This demonstrates the very common type of loop in which an index increases or decreases in equal steps.

* There is no significant difference between ⩾ and > in this context, but the former is translated more efficiently.

Mercury autocode provides a simple way of coding such loops. The instruction

I = P(Q)R  or  I = P(−Q)R

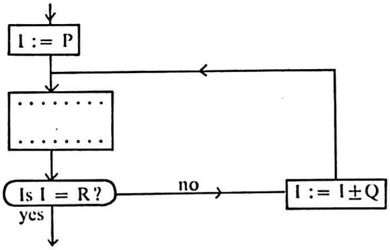
is placed at the beginning of the loop, and

REPEAT

is written at the end. Then the programme

```
..........
I = P(±Q)R
.......... } Instructions in the loop
.......... } which do not change I.
REPEAT
..........
```

has the effect of the following flow diagram:



*Example.* Calculate $\left(1+\frac{1}{n}\right)^n$ for $n = 1,2,3 \ldots m$, using the cycle-setting and REPEAT instructions.

The inner loop of this programme is

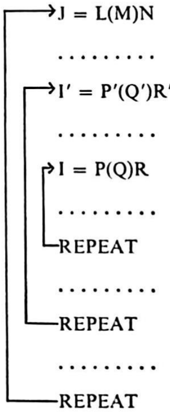
```
P = N(−1)1
A = AB
REPEAT
```

The full programme is therefore as shown in Fig. 8.3.

```
N = 1(1)M                          ┐
A = 1                              │
B = 1+1/N                          │
P = N(−1)1   ┐  Inner              │  Outer
A = AB       │  loop               │  loop
REPEAT       ┘                     │
PRINT (A) 1.8                      │
REPEAT                             ┘
```

Fig. 8.3. Calculation of $\left(1+\frac{1}{n}\right)^n$—second programme.

It will be seen that the cycle-setting instruction does not need a label and the REPEAT does not specify which cycle-setting instruction it returns to. This might be thought ambiguous in the case of a cycle within a cycle. However no difficulty arises so long as the programme has been coded correctly, starting with the innermost loop, then the loop which encloses it, then the loop which encloses that one, and so on. The final appearance of the programme is shown in Fig. 8.4.



```
J = L(M)N
.........
I' = P'(Q')R'
.........
I = P(Q)R
.........
REPEAT
.........
REPEAT
.........
REPEAT
```

Fig. 8.4. Cycles within cycles.

The connections between cycle-settings and REPEATs are shown by arrows in Fig. 8.4. It will be seen that there must be an equal number of cycle-setting and REPEAT instructions and each REPEAT is connected to the last "unattached" cycle-setting instruction. It is this latter rule that the compiler programme uses when translating the autocode programme into machine language and so it is unnecessary to provide a label to link cycle-setting and REPEAT instructions.

There are some points to note about the instruction $I = P(\pm Q)R$ and its associated REPEAT.

(1) I must be an index; P, Q and R may be indices or integers. Variables are not allowed.
(2) If Q or R are indices, they must be different from I.
(3) Only Q may be preceded by a + or − sign.
(4) Negative integers may not be written in place of P or R, but indices written in these positions may have negative values.
(5) Cycles may be used within cycles: a "nest" of eight cycles is allowed.
(6) Where one cycle lies within another, different indices must be used for the counters of the cycles.
(7) There must be an equal number of cycle-setting and REPEAT instructions.
(8) A jump to the instruction $I = P(\pm Q)R$ sets $I = P$ and starts the cycle.
(9) A jump to the instruction REPEAT makes the computer test whether $I = R$; if not, I is increased by $\pm Q$ and the cycle repeated.

Suppose now that J is to increase from $-1$ to $(N-1)$ in steps of 1. We cannot write $J = -1(1)N-1$, because P may not be replaced by a negative integer, nor may any index formula such as $(N-1)$ be used for R. So the instructions must be

```
K = N-1
J = -1
J = J(1)K
```

Similarly, if S is to decrease from MN to 0 in steps of N, we write

```
S = MN
S = S(-N)0
```

*Example.* Print the roots of $x^2-2bx+r = 0$ for $r = 0,1,2,\ldots n$.

The roots are $x = b\pm\sqrt{b^2-r}$. For $r\leqslant b^2$ they will be real, for $r>b^2$ they will be a complex conjugate pair. We shall print real and imaginary parts of the complex pair on the same line, but a pair of real roots will be printed on successive lines.*

In Fig. 8.5 we construct the programme without using a cycle-setting instruction.

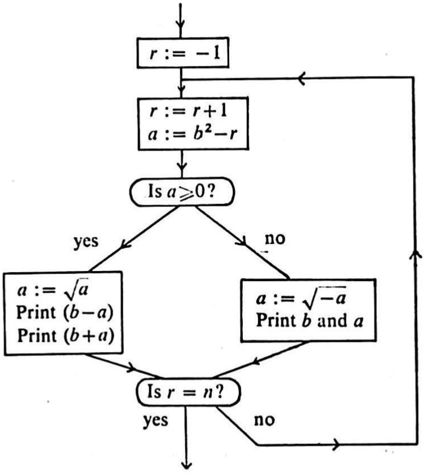


Fig. 8.5. Flow diagram for solution of $x^2 - 2bx + r = 0$.

The corresponding Mercury autocode programme, using a cycle-setting instruction, is:

```
   C = BB
   R = 0(1)N
   NEWLINE
   A = C-R
   JUMP 1, A≥0
   A = ψ SQRT(-A)
   PRINT(B)0,8
   PRINT(A)0,8
   JUMP 2
1) A = ψ SQRT(A)
   PRINT(B-A)0,8
   NEWLINE
   PRINT(B+A)0,8
2) REPEAT
```

Fig. 8.6. Programme for solution of $x^2 - 2bx + r = 0$.

* The instruction NEWLINE arranges that the next number printed starts a new line, see section 8.10.

The instruction JUMP 2 in Fig. 8.6 must *not* be replaced by REPEAT. If it were, there would be more REPEATs than cycle-settings in any programme of which this might form part, and the REPEAT labelled 2) would be matched with some other cycle-setting instruction than the one intended.

This example shows that, where a conditional jump inside a cycle chooses between two courses of action, the two branches must meet again at the REPEAT, if not before.

## 8.6 Sets of Numbers

A set of numbers will always be stored as main variables. If, for example, B3,B4, . . . B16,B17 is a set of 15 numbers, we can place their sum in C by the instructions:

```
C = 0
J = 3(1)17
C = C+BJ
REPEAT
```

By this use of suffices we need only one addition instruction in a loop obeyed 15 times, rather than 15 separate addition instructions.

A flow diagram for the evaluation of $y = a_0x^n+a_1x^{n-1}+ \ldots +a_{n-1}x+a_n$ was given in Fig. 3.2 (page 15). It is a simple matter to write the corresponding programme in Mercury autocode.*

```
Y = 0
I = 0(1)N
Y = YX+AI
REPEAT
```

The use of compound suffices is not restricted to the right-hand side of an equation: a variable may have a compound suffix wherever it occurs, e.g. in a comparison, or on the left-hand side of an equation.

*Example.* Find the largest of the set of numbers A1†, . . . AN. (See Fig. 3.6.)

```
    M = 1
    K = 2(1)N
    JUMP 12,AM≥AK
    M = K
12) REPEAT
    B = AM
```

Fig. 8.7. Largest element of a sequence.

However, Mercury autocode provides a function ψ MAX to replace this programme. The result of the instruction M = ψ MAX(A0,I,J) is the position of the largest number in the set AI,A(I+1), . . . AJ, not the number itself. A similar instruction, M = ψ MIN(A0,I,J), finds the smallest in the set. In both these instruction A0 must be the name of a main variable with suffix zero; I and J may be indices or integers; M must be an index. In practice we should use ψ MAX in preference to the programme of Fig. 8.7.

*Example.* Sort the numbers A1,A2, . . . AN into ascending order, using the above programme to find the largest element. (See Fig. 3.7.)

* In practice we would use the function ψ POLY, but this requires the constant term to have the smallest suffix.

† Sets of main variables always include a variable with suffix 0; it is wasteful not to use A0, but Fig. 3.6 does not include it.

```
    JUMP 99,N = 1
    N = N(-1)2
    M = 1
    K = 2(1)N
    JUMP 12, AM≥AK
    M = K
12) REPEAT
    B = AM
    AM = AN
    AN = B
    REPEAT
99) ......
```

Fig. 8.8. Sort into ascending sequence.

Mercury autocode does not make provision for a two-dimensional array of numbers as such, since a variable cannot be given a pair of suffices. Instead, the array has to be stored as a single sequence of numbers, and the position of a particular element has to be calculated from an index expression.

*Example.* The $m \times n$ matrix **A** is punched on tape, column by column, with column check-sums. Read this matrix and store it column by column in A1,A2, . . . , A($mn$), using the flow-diagram of Fig. 3.5.

We assume that $m$ and $n$ are already known, and that check-sums should be accurate to 7 decimal places: the programme will jump to label 77) when an error occurs. We shall use the instruction READ(X) which reads one number and stores it as X.

```
K = 1
J = 1(1)N
A = 0           For column sum
I = 1(1)M
READ(AK)
A = A+AK
K = K+1
REPEAT
READ(B)         Column-sum on tape
B = MOD(A-B)
JUMP 77,B≥0·0000001
REPEAT
```

Fig. 8.9. Matrix read and stored column by column.

It is not necessary always to store the data in the same sequence as that in which they are presented on the data tape. For example, the same $m \times n$ matrix **A** could be stored with the elements of each row in consecutive variables, even though it would still have to be read off the data tape column by column. In this case it would be necessary to calculate the position of $a_{ij}$ in the row-by-row array from the index formula

$$K = NI+J-N.$$

## 8.7 Switches

A switch chooses one of several courses of action in a single jump instruction. To do this in Mercury autocode, we need the concept of a *label store.*

In each block* of a Mercury autocode programme, some instructions are labelled. When this programme is translated into machine language, a list is made of these labels and of the positions of the corresponding instructions in the translated programme. The instruction

```
I) = L)
```

takes the value of L, looks up the corresponding label in this list, and temporarily changes index I into a *label store* by placing in it the position of label L in the translated programme. (I can be replaced by the name of any index, L by any index or integer.)

When this instruction has been obeyed, the instruction

```
JUMP (I)
```

jumps to the label whose position is held in label store I.

*Example*. Given an angle *a* (in degrees) between 0° and 360°, find sin *a* using the programme of Fig. 3.8. In this programme we assume that the function ψ SIN gives the sine of an angle in radians between 0 and $\pi/2$ only.

```
     Q = ψ INTPT (0·0111111111A+1)   Quadrant number
     I) = Q)                         Set label store I
     JUMP(I)                         Switch:
  4) A = 360−A                            fourth quadrant
     JUMP 1
  3) A = A−180                            third quadrant
     JUMP 1
  2) A = 180−A                            second quadrant
  1) X = ψ SIN (0·01745329252A)           first quadrant
     Q = Q+4
     I) = Q)
     JUMP(I)                         Switch:
7)8) X = −X                               third and fourth quadrant
5)6) HALT                                 first and second quadrant
```

Fig. 8.10. Reduction to first quadrant.

## 8.8 Hoot, Halt, and End

The instruction HOOT plays a certain note on the loudspeaker for one second before the next autocode instruction is obeyed. The instructions HOOT 1, HOOT 2, . . . HOOT 8, play the respective notes of one octave of a major scale, each note for one second. HOOT 8 gives the highest note (the same note as HOOT).

The instruction HALT makes the computer wait until the operator presses a key on the console; then the next autocode instruction is obeyed. An intermittent note is played on the loudspeaker to attract the operator's attention.

The instruction END terminates the execution of any autocode programme; it causes a high note to be played on the loudspeaker. It differs from HALT in that it is impossible to continue a programme after END has been obeyed.

* A chapter (section 8.12) or a routine (section 8.11).

## 8.9 Input

The normal medium for input and output of numbers on Mercury is 5-hole punched paper tape. Numbers such as −12·34 are punched on a data tape character by character; they must be followed by a *terminator*, either *CrLf* or *SpSp*. (A single *Sp* in any number is ignored in Mercury autocode; this allows the printing of numbers like +1 00000 00000 in a more legible form.) Negative numbers are preceded by a minus sign, but in the case of positive numbers the plus sign is optional.

Numbers to be stored as variables may also be punched in the floating-point form *a,p* for $a \times 10^p$; thus −0·00000001 could be punched as −0·1,−7 or −1,−8. Numbers to be stored as indices must be punched in fixed-point form without a decimal point.

Numbers which appear as constants in the programme must be punched in fixed-point form only. They do not require a terminator.

Every tape, whether programme or data, should end with the character →. The machine halts on reading this; then, when the operator presses a key, it continues reading the tape. By using this character we allow time for the operator to change tapes in the tape reader.

The READ instruction is used for the input of numbers to the computing store. In the form READ(X) it reads the first available number on the data tape and sets the variable X to have that value, leaving the data tape in position for the next number to be read. (X can be replaced by any variable.) In similar fashion, the instruction READ(I) sets the index I to the value of the number read.

A sequence of numbers may be read by a cycle of instructions: for example,

```
I = 1(1)N
READ (X(I−1))
REPEAT
```

reads N numbers and stores them as the set of main variables X0 to X(N−1).

*Example.* Find the mean, variance and standard deviation of a set of *n* numbers. (See Fig. 2.9, page 12.)

```
1) READ(N)
   B = 0
   C = 0
   P = 1(1)N
   READ(D)
   B = B+D
   C = DD+C
   REPEAT
   U = B/N
   V = ψ DIVIDE (−BU+C,N−1)
   W = ψ SQRT(V)
   NEWLINE
   PRINT (U) 0,8
   PRINT (V) 0,8
   PRINT (W) 0,8
   JUMP 1
```

Fig. 8.11. Mean, variance and standard deviation.

Programmes should usually be designed so that, when calculations with one set of data are complete, a jump to an earlier part of the programme makes the machine read and process another set of data. In the programme of Fig. 8.11 the single instruction JUMP 1 is sufficient. No HALT instruction is required, even if the sets of numbers are on different tapes, so long as the symbol → is punched at the end of each tape. It must also be punched at the end of the programme tape.

Unfortunately there is no facility in Mercury autocode for reading a sequence of numbers with a marker to indicate the end of the sequence, comparable to the v0 = TAPE* instruction in the Pegasus-Sirius autocode or the trigger facility of the Elliott 803 autocode. If a general programme is to be designed to read a sequence whose length is not known at the time of programming, one alternative is to specify that the sequence must be preceded on the data tape by an integer whose value is the number of numbers in the sequence. (This method was used in the programme of Fig. 8.11.)

A better alternative, available when the numbers lie in a certain range, is to end the sequence by a number outside the range; for example, a set of positive numbers could be ended by a negative number. The advantage of this alternative is that the numbers can be counted by the machine, thus eliminating a possible source of human error.

Mercury can have several tape readers connected to it. These input *channels* are numbered 1,2, etc. and the programme can select channel 2, for example, by the instruction

CHANNEL 2 R.

Once this has been obeyed all READ instructions* read from channel 2 until a different CHANNEL *n* R instruction is obeyed. When input takes place before a channel is selected, channel 1 is used: in particular, the programme tape is read through channel 1.

## 8.10 Output

The output instructions cause characters to be punched on the output tape, and this tape may subsequently be printed away from the computer.† However, we shall find it convenient to *describe* the output instructions in terms of what is finally printed by the teleprinter.

The instruction

PRINT (V) M,N

is used to print the numerical value of the general expression V, and it also specifies the style of printing to be used. In all cases numbers are rounded before being printed. Non-significant zeros in front of the decimal point are replaced by spaces, except that a single digit 0 is printed when the integer part of the number is zero. Negative numbers are preceded by a minus sign immediately before the first digit, whereas for positive numbers a space is printed instead of a plus sign. All numbers are followed by two spaces, so that output and input tapes are compatible and we can use an output tape directly as the input tape for another programme.

If M $\neq$ 0, the number is printed in *fixed-point* form, with M digits in front of the decimal point and N digits after it. However, if the number to be printed is too large for the style specified, all the digits of the integer part and N digits of the fractional part are printed; since extra characters are printed, the layout on the printed page is disturbed. (If more than 15 integer digits are needed, floating-point printing is used instead.)

If M = 0, the number is printed in *floating-point* form, with one digit in front of the decimal point and N digits after it.

As examples, we show in Fig. 8.12 the printing caused by various PRINT instructions. (The special variable $\pi$ normally has the value 3·14159265.)

* And the $\psi$10(A,U) instruction described in section 8.13.

† Most Mercury computers do not have a printer attached directly to them.

```
PRINT (−π) 1,4        −3.1416
PRINT (π) 3,5           3.14159
PRINT (π/100) 3,8       0.03141593
PRINT (−1000π) 2,4  −3141.5927
PRINT (−π) 0,4        −3.1416,  0
PRINT (π/100) 0,6      3.141593, −2
PRINT (0) 0,3          0.000,  0
```

Fig. 8.12. Examples of printing.

*Layout in Printing*

As only 68 characters can be printed on one line, it is essential to arrange for the numbers to be printed in a suitable layout. Two instructions are provided for this purpose:

NEWLINE and SPACE

The former punches the characters *CrLf* on the output; the latter punches one *Sp* character, and so provides an additional gap between numbers printed on the same line. Another way of obtaining a wide gap before a fixed-point number is to set too high a value for M; e.g. if it is known that $|X| < 100$, the instruction PRINT (X) 5,7 will always print three spaces in front of X.

*Example.* Print a table of $r$, $r^2$, $\sqrt{r}$, $\frac{1}{r}$, $\frac{1}{\sqrt{r}}$ for $r = 1(1)24$. For a particular value of $r$ the five functions are to be printed on the same line, with a blank line after every fifth line of figures.

We use the index S to count the number of lines printed since the previous blank line. As we cannot assume that the printer is ready to print at the beginning of a line when the programme is entered,* a NEWLINE instruction must be obeyed before the first PRINT.

```
   S = 0
   R = 1(1)24
   NEWLINE
   PRINT(R)2,0
   PRINT(RR)3,0
   X = ψ SQRT(R)
   PRINT(X)1,6
   PRINT(1/R)1,6
   PRINT(X/R)1,6
   S = S+1
   JUMP 1, S ≠ 5
   NEWLINE
   S = 0
1) REPEAT
   END
```

Fig. 8.13. Print table of functions with layout.

*Optional printing*

While a programme is being tested it is often useful to print some intermediate results which would not be needed if the programme were functioning correctly. These results help to show which parts of the programme are free from errors.

* In just the same way, we must not assume that S is zero initially.

If a ? is written after an arithmetic instruction, the normal effect is that the result of the calculation is printed, just as if a pair of extra instructions had been added,

```
NEWLINE
PRINT(X)0,5
```

for a variable, and

```
NEWLINE
PRINT(I)3,0
```

in the case of an index. This printing is called *optional* because it will be suppressed unless hand-switch 4 on the console is horizontal, both during translation and at the time of running.

*Captions*

Tables printed by the output instructions can be given headings. If a single heading is needed before any of the numbers are printed, the TITLE directive (described in section 8.14) should be used; but the CAPTION instruction is necessary if texts are to be interspersed with numerical results. Both TITLE and CAPTION are facilities for copying a set of characters on to the output tape.

In the written form of the programme, the word CAPTION is placed on a line by itself and the following line is stored as the text of the caption. (The *Cr* and *Lf* symbols at each end of the line are not included in the text.)

The text is printed each time the CAPTION instruction is obeyed and then the computer proceeds to the instruction written on the line after the text. Thus the programme

```
I = 1(1)3
NEWLINE
CAPTION
CASE NUMBER
PRINT(I)1,0
NEWLINE
.........
REPEAT
```

will print the texts

```
CASE NUMBER 1
CASE NUMBER 2
CASE NUMBER 3
```

on separate lines.

*Output Channels*

Mercury can have several punches connected to it. The output *channels* are numbered 1,2, etc., and the programme can select channel number 5, for example, by the instruction

```
CHANNEL 5 P
```

Once this has been obeyed all output instructions* punch on channel 5 until a different

* Including the ψ8(A,U,V,M,N) instruction described in section 8.13.

CHANNEL *n* P instruction is obeyed. When output takes place before a channel is selected, channel 1 is used.

## 8.11 Subroutines

In many calculations we find that one particular operation, such as the taking of a cube root, recurs frequently. When the calculation is written as a programme (often called a routine), a *subroutine* will be required for the recurrent operation. We shall normally write this subroutine and test it for correctness before incorporating it in the main programme.

We take as an example the programme of Fig. 8.14 which sets X equal to the cube root of A.

```
50) X = ψ MOD (A)
    JUMP 51, X = 0
    X = ψ LOG (X)
    X = 0·3333333333X
    X = ψ EXP (X)
    Y = ψ SIGN (A)
51) X = XY
```

Fig. 8.14. Subroutine to set $x = \sqrt[3]{a}$.

The simplest method of incorporating this subroutine is to write it out each time a cube root is required, but this would waste storage space if several copies were required. It is therefore better to place only one copy of the subroutine in the main programme. We then require some kind of switch at the end of the subroutine to tell the machine what point in the programme to return to. This switch is provided by making RETURN the last instruction obeyed by the subroutine. To enter the subroutine we must now use the instruction

JUMPDOWN 50.

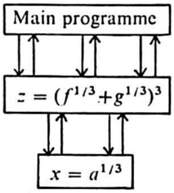
When the machine obeys the JUMPDOWN instruction it jumps unconditionally to label 50, but first it records the position of JUMPDOWN in the programme. When RETURN is obeyed, the machine looks up this record and jumps to the instruction following JUMPDOWN.

Thus, having placed the instruction RETURN at the end of the subroutine shown in Fig. 8.14, we can calculate $(f^{1/3}+g^{1/3})^3$ as follows:

```
10) A = F
    JUMPDOWN 50
    Z = X              f^{1/3}
    A = G
    JUMPDOWN 50
    Z = X+Z            f^{1/3}+g^{1/3}
    Z = ZZZ
```

Fig. 8.15. Programme to calculate $z = (f^{1/3}+g^{1/3})^3$.

The calculation of $z = (f^{1/3}+g^{1/3})^3$ may itself be a recurrent part of a larger programme. In this case it can also be used as a subroutine (once RETURN has been added as its final instruction), being entered by an instruction JUMPDOWN 10. The calculation of a cube root is now a sub-sub-routine of the main programme:



Each time a JUMPDOWN is obeyed, the machine also records the level of the subroutine to be entered. On obeying a RETURN the machine jumps back to the next higher level of subroutine and so the JUMPDOWN which led down from it is cancelled. There may be as many as six uncancelled JUMPDOWNs in force simultaneously.

The matching of a JUMPDOWN and its associated RETURN when the programme is *obeyed* is thus analagous to the matching of cycle-setting and REPEAT instructions. The reader should notice that cycle-settings and REPEATs are matched in the written programme in exactly the same way as in execution, but this is not true of JUMPDOWN and RETURN. A complete programme will obey equal numbers of JUMPDOWN and RETURN instructions but the number of written instructions of the two types need not be the same.

We may also use JUMPDOWN(I), where I is any label store whose value has been previously set by an I) = L) instruction (see section 8.7). This JUMPDOWN instruction records its position in the usual way, then jumps as if it were the JUMP(I) instruction.

*Routines*

In Mercury autocode a subroutine may be made a numbered *routine*. To give a subroutine the number 641, for example, we head it by

ROUTINE 641

and terminate it by a pair of asterisks.

Each such routine is a self-contained section of programme; for example, it must have an equal number of cycle-setting and REPEAT instructions. It also has its own set of labels. This enables us to use an existing routine in writing a new programme without having to ensure that different label numbers are used in the routine and the main programme.

If the cube-root subroutine of Fig. 8.14 is now called routine 641, we cannot enter it by the instruction JUMPDOWN 50, because this would refer to label 50 of the main programme, not label 50 of the routine. So we use

JUMPDOWN (R 641)

to enter routine 641 at its first instruction, or, to enter the routine at its label 15,

JUMPDOWN (R 641/15).

If the programme to calculate $z = (f^{1/3}+g^{1/3})^3$ is called routine 640, its final version will be as shown in Fig. 8.16.

```
ROUTINE 640
A = F
JUMPDOWN (R 641)
Z = X
A = G
JUMPDOWN (R 641)
Z = X+Z
Z = ZZZ
RETURN
**
```

Fig. 8.16. Routine to calculate $z = (f^{1/3}+g^{1/3})^3$.

The use of routines on the programme tape is explained on page 81.

Two instructions previously introduced behave in unexpected ways when used inside a routine. The instruction I) = 15) will set label store I to hold the position of label 15 in the routine, but the instruction I) = Q) will use the value of index Q to find the corresponding label *in the main chapter*, no matter how many JUMPDOWNs are in force. Secondly, CAPTION should be avoided.

## 8.12 Chapters

Until this point we have assumed that Mercury has only one type of store, and that this store is sufficiently large to hold any programme and all the variables it uses. In fact, Mercury has a small *computing store* of magnetic cores and a much larger *backing store* of magnetic drums.

Access to individual instructions or numbers held in the computing store is almost instantaneous, but a much longer time is required to obtain information from the backing store. As the names imply, the computing store is used to hold the current sets of instructions and numbers and the backing store provides a reserve of storage space from which information is transferred to the computing store in large blocks.

A Mercury autocode programme is divided into blocks called *chapters* which are numbered 0,1,2, etc. The maximum number of autocode instructions which can be held in one chapter is usually about 80, but this varies with the complexity of the instructions used.* Programmes which are too large for a single chapter often fall naturally into sections, and then separate chapters can be used for the various sections. Just one chapter is held in the computing store at any time.

The chapters of a programme each have their own sets of labels, numbered from 0 to 127. By use of the main-variable directives (section 8.14) they can also give different names to the main variables, but more frequently the same main variables are used in every chapter.

When Mercury is just about to enter an autocode programme, chapter 0 will be in the computing store and the backing store will hold the entire programme (including a copy of chapter 0). After some of the instructions of chapter 0 have been obeyed, a different chapter will have to be fetched into the computing store and entered. If chapter 1 is to be entered at label 15, the instruction will be

ACROSS 15/1†

* A directive PSA described in section 8.14 gives information about the size of the chapter.
† Notice that the instruction entering routine 1 at label 15 is JUMPDOWN (R1/15).

The instructions of chapter 1 will now be obeyed until another chapter-changing instruction is reached. This could enter a further chapter or return to chapter 0. In the same way, it is always possible to leave a chapter temporarily for another and return to it later.

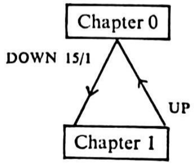
A chapter can also be used as a subroutine to a main chapter, in a way precisely similar to the use of subroutines described in the previous section. In this case the chapter is entered by an instruction

DOWN 15/1

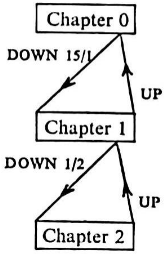
(instead of the ACROSS 15/1 used earlier), and the instruction

UP

makes the machine return to the instruction in the main chapter immediately following the DOWN. The programme can be illustrated by the following diagram:

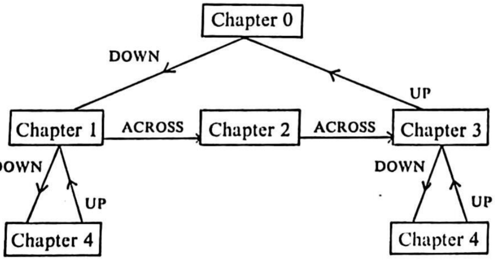


A chapter entered by DOWN is called a *sub-chapter*. It may have its own sub-chapter, which will be a sub-sub-chapter of the main chapter, as is shown in the following diagram:



The method is exactly analogous to that of JUMPDOWN and RETURN. DOWN makes a record of the current level and the position of the DOWN instruction, while UP cancels the last undeleted DOWN. We may go down as far as sub-sub-sub-chapter level.

ACROSS instructions may be obeyed between a DOWN and the UP which cancels it. No record is made of the position of any ACROSS, which is equivalent to a jump to another chapter and does not affect the chapter level in any way. Complicated arrangements of chapters such as the following are possible:



Here the DOWN in chapter 0 leads to chapter 1, but it is an UP in chapter 3 which eventually re-enters chapter 0 at the instruction following the DOWN.

The number of chapters allowed depends on the size of the Mercury being used. On a machine with 3 magnetic drums, chapters may be numbered up to 27.

The three chapter-changing instructions do not change any numbers in the store, except that the value of $\pi$ returns to 3·14159265 at each change of chapter.

## 8.13 Auxiliary Variables

Numbers held in the backing store are called *auxiliary variables*. The capacity of this store depends not only on the number of drums fitted to the computer, but also on the size of the particular programme. When a programme with highest chapter number $c$ is run on a Mercury with three drums, $13824-512c$ auxiliary variables can be stored.

The positions in the backing store are numbered from 0 to 13823, and all instructions using auxiliary variables refer to them by their *addresses* in the backing store. The phrase "auxiliary variable 1234" is often used as a shorthand for "the auxiliary variable at address 1234": its use does not imply that the *value* of that auxiliary variable is $+1234$.

If the address of a particular auxiliary variable is to be calculated by programme, exact arithmetic with variables (see page 59) will be required, since the range of values for an index is too small. All the instructions handling auxiliary variables allow us to specify a particular one by making the value of a variable in the computing store equal to the required address. (The primed special variables are often used for this purpose.)

For example, we may need to store an array of $m^2$ numbers as auxiliary variables starting at address 1024 and a second array immediately after the first. The instruction

$$Z' \approx MM+1024$$

sets special variable Z′ to hold the address of the initial auxiliary variable of the second array. Any instruction to handle this array can now use Z′ to find the starting point. We may therefore refer to the auxiliary variable whose address is the value of the special variable Z′, or, more succinctly, to the "auxiliary variable at address Z′".

Numbers stored as auxiliary variables cannot be used for calculations until they have been placed in the computing store. The two instructions

ψ6(V′)X0,I

and

ψ7(V′)X0,I

D

each written on a line by itself, are used to transfer between the two stores a set of I numbers, starting with variable X0 in the computing store and with the auxiliary variable at address V′ in the backing store. ψ6 transfers numbers into the computing store from the backing store, ψ7 in the opposite direction. In these two instructions V′ may be any general expression with an integral value; X0 may be replaced by the name of any main variable* (with a compound suffix if desired); I can be any index or integer. Thus ψ6(U′V′+N+1500) A(I+3), M is permitted.

By way of example we show the programme to read a data tape containing *m* and *n* followed by two arrays of $m^2$ and *mn* numbers (each assumed to have not more than 480 numbers). These numbers will be stored as consecutive auxiliary variables, starting at the auxiliary variable whose address is given in Z′ (assumed to have been set previously).

```
READ (M)
READ (N)
J = MM
I = 1(1)J
READ (A(I−1))
REPEAT
ψ7(Z′)A0,J
K = MN
I = 1(1)K
READ (A(I−1))
REPEAT
ψ7(Z′+J)A0,K
```

Fig. 8.17. Read and store two arrays (1).

A more efficient programme for this problem would use the instruction

ψ 10(A,U)

which reads U numbers from paper tape and places them in the auxiliary variables starting at address A. Both A and U may be integers or variables with integral values.

The programme for this problem then becomes:

```
READ (M)
READ (N)
X′ ≈ MM
ψ 10(Z′,X′)
Y′ ≈ Z′+X′
X′ ≈ MN
ψ 10(Y′,X′)
```

Fig. 8.18. Read and store two arrays (2).

In this programme the arrays are not restricted to 480 numbers each.

There is also an instruction for punching numbers in the auxiliary variables directly on to the output tape, namely

ψ8(A,U,V,M,N).

The set of UV numbers starting at address A is punched with M digits in front of the point and

* Or, when I = 1, by a special variable.

N digits following it. M and N, which may be integers or indices, have exactly the same effect as in the PRINT(V)M,N instruction: in particular, M = 0 specifies floating-point numbers. A, U and V may be integers or variables with integral values. When the output tape is printed, the numbers will be printed in a single column divided into V blocks of U numbers, with a blank line after each block.

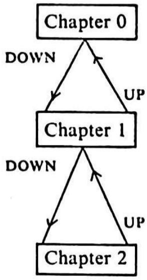
*PRESERVE and RESTORE*

When a change of chapter is made, it is sometimes valuable to be able to preserve the current values of the main and special variables and the indices. Then, when the sub-chapter is later replaced by the main chapter, these values can be restored. This facility is most frequently necessary when the two chapters give different names to the main variables.

The instruction PRESERVE (usually placed just before a DOWN instruction) places in the backing store the current values of the 480 main variables, the 29 special variables, and the 12 indices I to T (but not the primed indices I′ to T′). This does not affect the values of the auxiliary variables. The instruction RESTORE, normally obeyed shortly after the return to the main chapter by an UP instruction, restores these values to the computing store, leaving the primed indices unchanged. Both PRESERVE and RESTORE set $\pi$ equal to 3·14159265.

The reader will notice that, if the sub-chapter leaves its results in the computing store, they will be lost when the RESTORE instruction is obeyed. They must therefore be stored as auxiliary variables before RESTORE is obeyed.

Three *dumps* are used by the PRESERVE and RESTORE instructions; they are called the main dump, sub-chapter dump and sub-sub-chapter dump. PRESERVE and RESTORE each use the dump corresponding to the level of their own chapter. Thus a programme of the form



can have a PRESERVE and a matching RESTORE in both chapter 0 and chapter 1.

**8.14 Directives**

When the complete programme has been written, it is punched on paper tape. This tape is then fed into the computer, where the *autocode compiler* programme reads it, translates the individual instructions into sequences of machine instructions, compiles the chapters (one at a time) and stores them in the backing store. Finally, when the entire programme has been read and compiled, chapter 0 is brought into the computing store and entered at its first instruction.

It will be seen that no instruction of the user's programme is obeyed until the entire programme tape has been read.* Only then can instructions be obeyed, and only then can data tapes be read.

However, the programme tape must include some information to help the compiler in its translation; for instance, the compiler must be told where each chapter begins and ends. The name *directive* is given to such information. Because directives tell the compiler how to do its work, they are obeyed as soon as they are read from the programme tape.

Each chapter must begin with the directive

CHAPTER *c*

where *c* is the chapter number, and end with the directive

CLOSE.

After reading the CLOSE directive, the compiler fills in a number of cross-references within the chapter and then places the complete chapter in the backing store. Unless the chapter was chapter 0, the compiler continues to read the programme tape; but after chapter 0 the compiler brings that chapter into the computing store and enters it at its first instruction. This is the only way in which a programme can be entered.

If main vąriables are used in any chapter, their names must be declared before any instruction refers to them. So *main-variable directives* normally follow the CHAPTER directive. They take two forms.

The first kind are directives such as

$$X \rightarrow 57$$

which declares a set of 58 main variables, X0, X1, . . . X57. Several sets of main variables may be used in one chapter; thus we may have

$$\begin{gathered} X \rightarrow 57 \\ A \rightarrow 100 \\ \pi \rightarrow 6 \end{gathered}$$

at the head of a chapter. These directives give the names X0,X1, . . . X57, A0,A1, . . . A100, $\pi 0$, . . . $\pi 6$ to the first 166 main variables. If, during execution, the programme calls for main variable A101, it will pick up $\pi 0$. (Suffices may become negative or too large during execution without any error being detected.)

The second kind of main-variable directive is

VARIABLES *c*

where *c* is the number of a chapter earlier on the programme tape. This copies the main-variable settings of chapter *c* into the new chapter. Directives of the first kind may be used after this directive, but they should not precede it.

Not more than 480 main variables may be declared in any one chapter. The 29 special variables are never declared.

Two directives which may be used at any point on the programme tape are PSA (standing for Print Space Available) and TITLE.

On reading PSA, the compiler immediately prints out the chapter number and the amount of unfilled space. The second number is, in fact, the number of instructions in machine code which can still be inserted in the chapter; it starts at 832 and decreases steadily.

When the compiler reads TITLE, it copies the following line on to the output tape immediately. TITLE should be distinguished from the instruction CAPTION, for the text following a TITLE is

* The instruction END, for example, does not make the compiler stop reading tape.

not stored and so can be punched once only. If a text of several lines is to be copied, a succession of TITLE directives will be required, since TITLE copies a single line.

Routines appear on the tape preceded by the directive

ROUTINE *n*

and followed by the

**

directive. No matter how many chapters call for routine *n* by using JUMPDOWN (R *n*) instructions, only one copy of the routine is placed on the programme tape. The place for routines is the head of the tape, before the first CHAPTER directive. Nevertheless, they do not require any main-variable directives. This is because a routine, unlike other sections of the programme, is not translated immediately: instead, it is stored temporarily, and the translation takes place at the CLOSE of each chapter which calls for it. Only one copy of any routine is placed in a chapter, no matter how many instructions jump down to it.

For similar reasons, a PSA directive in a routine is not obeyed as soon as it is read; it is stored with the routine and it is obeyed each time the routine is compiled into a chapter. The TITLE directive must not be used in a routine, but quicky-settings (see below) are permitted.

Some of the functions listed in Table 8.1 (page 60) take a long time to evaluate because the subroutines which calculate them are held in the backing store; each time the function is needed, its subroutine is transferred into the computing store. No function takes more than 6 msecs. to evaluate, yet up to 17 msecs. will be added because of the transfer.

By means of a *quicky-setting directive* such as

$\psi$ SQRT

placed just before the CLOSE of a chapter, the programmer can arrange to have the subroutine for the corresponding function included in the chapter. It will be transferred to the computing store when the chapter is entered and no further transfers are required when the function is called for. Such a function is then called a *quicky*.

The quickies required in any chapter should be listed in order of preference with the most frequently used function first. The compiler does not insert the quickies into a chapter until it has translated all the instructions and compiled the routines; then it inserts only those quickies for which there is sufficient space.

Table 8.2 gives the complete list of functions which can be declared as quickies. If one of the pair $\psi$ COS and $\psi$ SIN is declared, the other also becomes a quicky: the same is true for the pair $\psi$ RADIUS and $\psi$ SQRT.

| | | |
|---|---|---|
| $\psi$ ARCTAN | | |
| $\psi$ COS | and | $\psi$ SIN |
| $\psi$ EXP | | |
| $\psi$ LOG | | |
| $\psi$ RADIUS | and | $\psi$ SQRT |
| $\psi$ TAN | | |

Table 8.2. Quicky-setting directives.

Functions which do not appear in this table are always translated into a set of instructions in the chapter, and so do not need to be made quickies.

*Comments* may be included on a programme tape. If a line begins with a pair of > symbols, the compiler reads it but does no translation.

An example of a complete programme tape is shown in Fig. 8.19, where only the directives are given.

```
TITLE
EXAMPLE OF COMPLETE PROGRAMME
TITLE
BHDS/29.2.63

 ROUTINE 640
 .  .  .
 PSA
 **
 ROUTINE 641
 .  .  .
 PSA
 **
 >> ROUTINES SHOULD PRECEDE CHAPTERS

 CHAPTER 1
 X→57
 A→100
 π→6
 .  .  .
 PSA
 .  .  .
 PSA
 ψ EXP
 ψ COS
 CLOSE

 CHAPTER 2
 VARIABLES 1
 H→14
 .  .  .
 ψ LOG
 CLOSE

  >> CHAPTER 0 MUST COME LAST
  >> THE PROGRAMME TAPE SHOULD END WITH AN ARROW
  CHAPTER 0
  VARIABLES 2
  .  .  .
  CLOSE
  →
```

Fig. 8.19. Directives in a complete programme.

The compiler routine can detect the more obvious *faults* in a programme. The majority of faults are detected as soon as they are translated:* examples are the use of main variables not declared by a main-variable directive, and the second occurrence of a label in one chapter. Some faults

* Faults in a routine are not detected until the routine is compiled into a chapter.

cannot be detected until the CLOSE of the chapter in which they occur; an example is a JUMP instruction leading to a non-existent label. On detecting a fault the compiler prints FAULT *n** and any information which may help in the diagnosis of the error. Although the programme cannot now be run, the compiler continues reading the programme tape so that other grammatical faults are discovered. When the CLOSE of chapter 0 is reached, the compiler prints NO GO and does not enter the programme.

Faults can also be detected during the execution of a programme. Examples are:

(1) negative argument for $\psi$ SQRT or $\psi$ LOG;
(2) argument greater than +177 for $\psi$ EXP (so that $e^x > 2^{255}$);
(3) division by zero;
(4) a spurious character on a data tape.

Two courses of action are possible in such circumstances. If there is a label 100 in the current chapter (label 0 for the spurious input character) the programme jumps to this label, and so the programmer can arrange for whatever action he requires. If no such label exists, the machine prints the fault number and the chapter number, followed by the values of the indices and special variables. The machine then stops and execution of the programme cannot be continued.

If a fault is detected while a *routine* is being obeyed, it is still label 100 or 0 of the *chapter* which is looked for.

The compiler does not find every fault which offends the grammar of autocode and it cannot detect any mistakes in the logic of the programme, such as the overwriting of results which will be required later, or conditional jump instructions with the wrong condition. The programmer must test the programme on data for which the results are already known, so that all such errors can be eliminated. They are likely to be numerous.

## 8.15 Other facilities

This chapter does not describe all the instructions and directives provided in Mercury autocode. Some of the more important facilities which have been omitted are:

(1) a set of 20 functions, $\psi 11$ to $\psi 30$, for matrix operations;
(2) the instruction INTSTEP for integrating a set of differential equations;
(3) calculations with complex numbers;
(4) calculations with double-length numbers, for extra precision;
(5) the generation of pseudo-random numbers;
(6) storage for more than 24 indices;
(7) tables of numbers, integers and labels;
(8) the use of a sub-programme of several chapters;
(9) list processing;
(10) the incorporation of instructions in machine language into an autocode programme;
(11) the instruction RMP which reads more programme;
(12) the input instruction SUNVICLOGGER.

A description of the Mercury autocode language is given in reference [9]. Some facilities of CHLF3 autocode, in particular the use of routines, are not described in this publication. The only full descriptions are privately printed and are not obtainable outside computer establishments.

* A short list of fault numbers is given in the "Mercury Autocode Manual" [9].

PART III

# ALGOL

CHAPTER 9

# ALGOL

**Introduction**

We have seen in Part I that before we can put a calculation on an automatic computer, we need a precise description of the numerical processes to be carried out. Such descriptions are known as algorithms. Each of the autocodes described in Part II provides a notation for algorithms which can be understood directly by a particular machine. However, the various autocodes are sufficiently different to cause difficulties in communication between groups of programmers or users working with different machines. For this reason there is now considerable interest in a universal computing language known as ALGOL, which stands for ALGOrithmic Language. This language is the result of the work of several international conferences and committees, culminating in the "Report on the Algorithmic Language ALGOL 60" which was published early in 1960. A revised version [14] of this report was adopted in 1962.

Algol is a language for describing numerical processes so that they can be understood both by human beings and by machines. For a machine to understand Algol it must be provided with an Algol *compiler* or *translator*, i.e. a special programme which enables the machine to translate any Algol programme into its own code. Such translators have been written for and are being used on a number of computers in Europe and America, and in particular for the Elliott 803 and 503, the English Electric KDF 9, and the Ferranti Pegasus in Britain. Algol translators are expected to be available for most large computers in future.

**9.1 Statements and Identifiers**

An Algol programme consists essentially of a sequence of instructions, called *statements*. These statements are written in a notation similar to that adopted in Part I, except that instead of relying on the layout of instructions on separate lines we now use a *semicolon* to indicate the end* of an instruction.

Algol permits great variety in the notation for variables. A variable is denoted by an *identifier*, which may be any combination of letters or of letters and decimal digits, provided it always starts with a letter.

To illustrate what an Algol programme looks like, we give in Fig. 9.1 a straightforward transcription of the programme of Fig. 2.9 for estimating the mean, variance and standard deviation.

Instead of the symbols $p, b, c, m, v, s$ used in Fig. 2.9 we now adopt the rather more descriptive identifiers *count*, *sum*, *sumsq*, *mean*, *variance*, *standev*. The symbol := is the standard Algol notation for "becomes", as in Part I. Any statement using this symbol is called an *assignment statement*.

Algol also recognizes multiple assignment statements, e.g. instead of $a := 0$; $b := 0$; we may write $a := b := 0$.

It will be seen that certain basic symbols of Algol, like the words **begin**, **if**, **then**, **go to**, **end**, are

* Strictly speaking the semicolon is only needed to separate different statements and "declarations" (to be defined later) from each other. However, in this chapter we put a semicolon also at the end of each programme, because it will be needed when incorporating the programme in a larger programme.

always printed in boldface; these words should be underlined in manuscript or typed programmes.

```
begin integer n, count; real sum, sumsq, x, mean, variance, standev;
  n := input; count := n; sum := 0; sumsq := 0;
repeat: x := input;
  sum := sum+x; sumsq := sumsq+x ↑ 2;
  count := count−1; if count ≠ 0 then go to repeat;
  mean := sum/n;
  variance := (sumsq−sum × mean)/(n−1);
  standev := sqrt(variance);
  print(mean); print(variance); print(standev);
end mean, variance, standev;
```

Fig. 9.1.

In the first line, immediately after **begin**, we have the so-called *type declarations*, stating which of the identifiers are of type **integer** (taking integer values only) and which are of type **real** (not restricted to integer values). These declarations enable the compiler to allocate storage locations and to determine whether to use integer or floating-point arithmetic for the identifiers concerned.

In line 5 of the programme we have an example of a *conditional* statement of the form **if** . . . **then** . . .; this is governed by the rule that the (unconditional) statement immediately following **then** is ignored unless the condition in the *if* clause is satisfied. However, by enclosing a sequence of statements in the so-called *statement brackets* **begin** and **end** we can obtain a *compound statement* which may be governed by a single *if* clause, e.g.

**if** . . . **then begin** . . .; . . .; . . . **end**;

Conditional statements may also take the form

**if** . . . **then** . . . **else** . . .;

Because this is a single statement, **else** must *not* be preceded by a semicolon. The statement following **else** may itself be another conditional statement; some examples of this will be given in section 9.7.

In our example the statement governed by the *if* clause is a simple jump instruction, known as a *go to statement*.* The identifier *repeat* is here used as a *label*, and this label followed by a colon must be written in front of the instruction to which we want to jump.

We can include explanatory notes in an Algol programme by placing the word **comment** immediately after a semicolon or immediately after **begin**; this word **comment** and everything up to and including the next following semicolon will then be ignored by the translator. Similarly everything between the basic word **end** up to (but excluding) the first following semicolon or **end** or **else** will be ignored. The last line of our programme illustrates the use of this device. The existence of this convention explains why, in general, we must place a semicolon after **end**.

## 9.2 Arithmetic expressions and basic functions

The programme given in the previous section illustrates the use of the operators +, −, ×, /, ↑ for addition, subtraction, multiplication, division, and raising to a power ("exponentiation"). Thus $x \uparrow 2$ stands for $x^2$, and this notation enables us to write formulae involving powers on a single level. Algol also provides the operator ÷ for finding the integer quotient of two integers.†

* Some compilers use **goto** as a single word.

† If $a$ and $b$ are integers of the same sign, then $a \div b$ represents the integral part of the quotient $a/b$; for integers of opposite signs we obtain $-(a \div (-b))$.

The order in which arithmetic operations occurring in one expression are to be carried out is governed by the following rules of precedence. The highest precedence is given to the operator $\uparrow$, which binds numbers more closely than any other; next come the operators for multiplication and division, which are on the second level of precedence; finally we have the operators for addition and subtraction. Thus

$$sumsq+x\uparrow 2 \quad \text{gives} \quad sumsq+(x\uparrow 2)$$

and

$$sumsq-sum\times mean \quad \text{gives} \quad sumsq-(sum\times mean).$$

Where the above rules do not determine the order of precedence, the operations will be carried out from left to right.* In cases where we want the operations to be carried out in a different order, round brackets should be used as in ordinary mathematical notation, and we may use brackets within brackets to any depth.

Algol allows numbers (and variables) of type real and of type integer to occur together in the same arithmetic expression. Furthermore, if an arithmetic expression of one type is to be assigned to a variable of the opposite type, the value of the expression will be automatically converted to the appropriate type. Conversion from real to integer type is equivalent to rounding to the nearest integer. Variables defined by a multiple assignment statement must all be of the same type.

Algol provides nine basic mathematical functions; in each case the argument, which may be an arithmetic expression, is placed in round brackets after the name of the function. We have:

| | |
|---|---|
| *sqrt* (E) | positive square root of E |
| *sin* (E) | (where E is in radians) |
| *cos* (E) | (where E is in radians) |
| *arctan* (E) | (principal value between $-\frac{1}{2}\pi$ and $+\frac{1}{2}\pi$) |
| *ln* (E) | natural logarithm of E |
| *exp* (E) | exponential of E |
| *abs* (E) | absolute value (modulus) of E |
| *sign* (E) | $+1$, 0, or $-1$, according as E is positive, zero, or negative. |
| *entier* (E) | largest integer not greater than E.† |

These functions may themselves be used as operands in arithmetic expressions in the same way as variables or constants. The nine identifiers *sqrt, sin, cos, arctan, ln, exp, abs, sign, entier* must not be used for any other purpose.

Algol does not define any input or output operations, because these may depend on the particular input and output mechanisms available on a given machine. Unfortunately different conventions are being adopted for different translators. In this chapter we shall use *input* as defined in Part I (page 11) to read the next number from an input device, and we shall use *print* (E) for putting the value of the expression E on to an output device. As regards non-numerical information, Algol defines a *string* as being a sequence of characters enclosed in the *string quotes* ‘ and ’, but no operations for handling such strings are defined.

The special symbol $_{10}$ followed by an integer (signed or unsigned) is used to denote powers of ten. Thus $_{10}-8$ stands for $10^{-8}$, and we may also attach such a power of ten to a decimal number as in $2{\cdot}5_{10}-8$ which is the same as ·000000025. However, we must use a multiplication sign when multiplying a *variable* by a power of ten, as in $a\times{}_{10}+8$ or $a\times{}_{10}8$.

* Thus $a/b\times c$ means $(a/b)\times c$, and $a/b/c$ means $(a/b)/c$.

† *entier:* ici on parle français.

## 9.3 for statements

Our example programme of section 9.1 contains a loop in which we count down from *n* to 1. Algol provides special facilities for handling loops by means of *for statements*, and this enables us to write the programme for mean, variance and standard deviation as in Fig. 9.2.

```
begin integer n, count; real sum, sumsq, x, mean, variance, standev;
  n := input; sum := sumsq := 0;
  for count := 1 step 1 until n do
    begin x := input;
      sum := sum+x; sumsq := sumsq+x↑2
    end;
  mean := sum/n;
  variance := (sumsq−sum×mean)/(n−1);
  standev := sqrt(variance);
  print(mean); print(variance); print(standev)
end mean, variance, standev;
```

Fig. 9.2.

It will be noticed that by using the *for* statement we no longer need to label the beginning of the loop, but on the other hand we must now indicate just which instructions are comprised in this loop. Where such a loop consists of more than one simple statement, we make it into a single compound statement by introducing the statement brackets **begin** and **end**, as in the example above. The careful reader will notice that the semicolon may be omitted in front of **end**. The reason for putting a semicolon after **end** was given at the end of section 9.1.

When the *controlled variable* (e.g. *count* in Fig. 9.2) is given a value outside the range defined by the **step** . . . **until** . . . element, we say that the "*for* list is exhausted", and the statement following **do** will not be executed for such a value of the controlled variable.* Thus, if in

**for** $i$ := 1 **step** 2 **until** $m$ **do** $S$

$m$ has the value 6, the statement $S$ will be executed for $i = 1,3,5$; whereas, if $m$ has the value $-1$, $S$ is not executed at all. In cases where the controlled variable is of type real, rounding errors may cause its intended final value to fall just outside the range defined by **until**. It may therefore be necessary to adjust the **until** element, as has been done in

**for** $h$ := 0 **step** 0·1 **until** 1·05 **do** . . . ;

if we put "**until** 1·0 **do**", there is the danger that the result of ten additions of 0·1 might exceed 1·0 owing to rounding errors, in which case 0·9 would be the last value of $h$ to be used.

Two other types of *for* list are available, as in:

**for** $k$ := 1, 2, 4, 8, 16, 32, 64 **do**. . .

and

**for** $k$ := 1, $2\times k$ **while** $k<100$ **do** . . .

These two lists are actually equivalent; the last *for* list is considered exhausted as soon as the controlled variable $k$ is assigned a value for which the condition specified by **while** does not hold. This form of *for* list is particularly useful in iterative processes where we may want to repeat the

* After exhaustion of the *for* list the programme proceeds to the next instruction, and the value of the controlled variable is not defined. Compiler writers are left free to organize the test for exhaustion to suit their convenience.

loop until some expression becomes sufficiently small. We shall illustrate this in Fig. 9.3 by giving an Algol version of the programme shown in Fig. 2.8 (page 12) for finding $\sqrt{a}$:

```
begin real z;
  root := 1·0;
  for z := (root+a/root)/2 while abs(z−root)>z× ₁₀−8 do root := z
end;
```

Fig. 9.3.

Notice that, since the controlled variable *z* is considered as being undefined on exhaustion of the *for* list, we take the final value of *root* to be our approximation to $\sqrt{a}$. In this respect the present algorithm differs slightly from that of Fig. 2.8. In practice we should of course use the basic function *sqrt*(*a*) to find $\sqrt{a}$.

## 9.4 Subscripts and arrays

In chapter 3 we discussed the importance of the suffix notation for dealing with sets of numbers. In Algol such a set is called an *array*; its elements are called *subscripted variables* and are referred to by writing the suffix (i.e. *subscript*) in square brackets after the identifier of the array. Thus the array *a* in the example of Fig. 3.6 has the elements *a*[1], *a*[2], . . . , *a*[*n*]. To enable the translator to allocate the right number of storage locations, it is necessary to give *subscript bounds* when declaring an array. This is done by one of the declarations

**real array** *a*[1:*n*] or **integer array** *a*[1:*n*]

according as the elements of the array are of type real or of type integer; the declaration **real array** may be abbreviated to **array**. Note the use of the colon to separate the upper from the lower suffix bound in the array declaration.

We can now write the procedure of Fig. 3.6 (page 18) for finding the largest element of the array *a*.

```
begin integer k;
  m := 1;
  for k := 2 step 1 until n do
    if a[k]>a[m] then m := k;
  max := a[m]
end;
```

Fig. 9.4.

We have assumed that all variables other than the "local variable" *k* have been defined outside this block. Note that Algol resolves the difficulty of the special case of $n = 1$, because in this case $k = 2$ is already outside the range of the *for* list, so that the programme proceeds to execute the instruction *max* := *a*[1] which is correct.

Usually subscripts are variables of type integer. However, Algol allows the use of any arithmetic expression as a subscript and will evaluate this to the nearest integer. Further, the subscript bounds in an array declaration need not necessarily be constants. Algol allows the use of *dynamic arrays* in which the subscript bounds are variables or expressions evaluated at the time of execution of the programme.

Algol also allows arrays with more than one subscript. Thus a two-dimensional array representing an $m \times n$ matrix **A** would have its elements denoted by *a*[*i*,*j*] and would be declared as **array** *a*[1:*m*, 1:*n*]. In Fig. 9.5 we use this notation to re-write the programme of Fig. 3.5 (page 17) for reading the elements of a matrix column by column together with column check-sums, failure of this check leading to a statement labelled *error*. By going over to a two-dimensional notation we shall no longer need the suffix *k* which was used in Fig. 3.5 for putting the matrix elements in a one-dimensional sequence. We now assume that *m* and *n* have already been defined and so need not be read in.

```
begin integer i, j;  real sum, b;
  for j := 1 step 1 until n do
    begin sum := 0;  for i := 1 step 1 until m do
      begin b := input; sum := sum+b;  a[i,j] := b end;
      b := input;
      if abs(sum-b)>₁₀-7 then go to error
    end
end;
```

Fig. 9.5.

The reason for introducing the temporary variable *b* in the fourth line of Fig. 9.5 is to avoid having to write *a*[*i,j*] twice. In this way the computer will only have to calculate the address of the storage location of *a*[*i,j*] once for each element of the matrix. Even so, the use of a two-dimensional array is likely to take more computer time than the straightforward approach of Fig. 3.5.

## 9.5 Blocks, declarations and labels

Any identifier, other than a formal parameter or a label, must be listed in a *declaration* before it can be used. Such declarations can only be given immediately after the symbol **begin** which introduces a compound statement, and they are then valid only within this compound statement, i.e. up to the symbol **end** which terminates it. A compound statement which contains one or more declarations is called a *block*.

The division of a programme into blocks is one of the features of Algol. The variables that are declared at the head of a block are called *local variables* of that particular block, and the values of any such variables are lost when we leave this block (either by going through **end**, or by means of a *go to* statement).

Algol allows blocks within blocks to any depth.*

If an identifier *I* of an outer block is declared again in an inner block, then the translator will regard the *I* of the inner block as a distinct variable which temporarily suppresses the meaning that *I* had in the outer block; but on returning from the inner block to the outer block *I* will resume its former meaning and the value it last had in the outer block. To test his understanding of this point, the reader should convince himself that the effect of the example programme of Fig. 9.6 is to print the number 13.

```
begin integer a, b, c;
  a := 1;  b := 2;
  begin integer a, d;
    a := 3;  d := 4;  b := a×d;
  end;
  c := a+b;  print(c)
end;
```

Fig. 9.6. Problem for the reader.

* It is customary to arrange the programme so that each **end** appears either vertically below or horizontally to the right of the **begin** corresponding to it, and to indent an inner block relative to its embracing block. Although the meaning of a programme does not depend on its layout on the page, this does help the human reader to understand and check the programme, and we have followed this convention throughout this chapter.

The rules for local variables make it possible to build up a programme from blocks written independently by different programmers, each of whom is free to choose any identifiers he likes for the local variables of his block. Any clash of names will be automatically resolved by the translator.

In addition to its local variables, an inner block may also use *non-local* variables which are declared in some outer (embracing) block; such variables are sometimes called *global* variables of the inner block, because they preserve their meaning and their value in going from the outer to the inner block and vice versa. Any variable whose value is calculated in one block in order to be used in another block must of necessity be a global variable. Thus the block of programme for finding $\sqrt{a}$ (see Fig. 9.3) does not contain a declaration for the variable *root* because this will be used to communicate the result of the calculation to another block.

It will now be seen that each block introduces its own level of nomenclature, and that the meaning of the identifiers used within a block can only be fully understood after reading the declarations at the head of that block. For this reason the only way in which a block may be entered from outside is through the symbol **begin** and the declarations that follow it.

Although Algol does not require labels to be declared explicitly, the appearance of an identifier (or an unsigned integer) followed by a colon in front of a statement defines a label, and this label is valid only in the block in which it has been so defined. If we regard such definition of a label as being equivalent to a declaration, then the rules governing the use of labels are exactly analogous to the rules governing local and global variables. It follows that a label defined in an inner block is inaccessible to a *go to* statement outside this block.*

By prefixing the symbol **own** to the type declaration † of variables or arrays we ensure that, on leaving the block, the *values* of such *own variables* or *own arrays* are preserved for use on subsequent re-entry to the block. Such variables are still local to the block in which they are declared in so far as they are inaccessible outside this block.

## 9.6 Procedures

One often has to use the same sequence of instructions at several places in a programme, or in several programmes. Such sequences of instructions are sometimes referred to as subroutines, but in Algol they have been given the special name *procedure*.

Any procedure which is used in a given block must be defined by a *procedure declaration* at the head of this block, or at the head of an embracing block. Such a procedure declaration consists of two parts, namely the *procedure heading* and the *procedure body*. Fig. 9.7 shows the declaration of the procedure *coeffin* for reading a sequence of coefficients from an input device:

```
procedure coeffin(n, a);  integer n;  array a;
  begin integer i;
    for i := 0 step 1 until n do a[i] := input
  end;
```

Fig. 9.7.

Here the first line gives the procedure heading, and the block defined by **begin** and **end** is the procedure body.

* An additional restriction, not related to block structure but necessary to avoid ambiguities, is that a *go to* statement outside a *for* statement may not lead to the inside of the *for* statement.

† **own real, own integer, own real array**

The procedure heading starts with the identifier *coeffin* which names this procedure, followed by the list of *formal parameters* in brackets; the nature of these formal parameters is then indicated by a *specification*. Although Algol regards such a specification as optional, it is in fact required by all existing translators to enable them to produce an efficient programme. It should be noted, however, that such a specification of the formal parameters does not have the effect of a declaration,* and that in the case of an array the specification does not give subscript bounds.

The procedure declaration serves merely to *define* a procedure for future reference within the same block. We may then cause the procedure to be executed by means of a *procedure call* (or *procedure statement*), which consists of the name of the procedure followed by a list of *actual parameters* in brackets. Thus the statement *coeffin*($n$, $a$) will cause the procedure body given above to be executed. However, the actual parameters need not have the same names as the formal parameters. Thus the statement *coeffin*(5, $b$) may be used to read 6 numbers and assign their values to the elements $b[0]$, $b[1]$, . . . , $b[5]$ of an array $b$, provided of course that this array has been declared either in the block in which the procedure call occurs or in an embracing block.

Another kind of procedure is analogous to the use of the basic functions in that it produces just a single value which can be put directly into an arithmetic expression by writing the name of the procedure with its actual parameters in brackets. Such *function procedures* are declared as either **real procedure** or **integer procedure**, as the case may be, and such a declaration distinguishes them from the previous kind of procedure. Thus the procedure (discussed in Fig. 3.2 on page 15) for evaluating the polynomial

$$a_0x^n+a_1x^{n-1}+ \dots +a_{n-1}x+a_n$$

could be defined by the declaration of Fig. 9.8.

```
real procedure polynomial(x,n,a);  real x;  integer n;  array a;
  begin integer i;  real p;  p := a[0];
    for i := 1 step 1 until n do p := p×x+a[i];
    polynomial := p
  end;
```

Fig. 9.8. Declaration of function procedure *polynomial.*

Note that the identifier of the procedure must always appear (*without* its parameters) as the left part of an assignment statement in the procedure body so as to define the value of such a function procedure.†

Suppose now that we wanted to find the difference between the values of this polynomial for $x = 2$ and for $x = 1$. This may be done by the Algol statement

*difference* := *polynomial*(2·0, $n$, $a$)−*polynomial*(1·0, $n$, $a$);

In order to show how procedures are incorporated in a complete programme, we shall now give a programme for tabulating the values taken by the two polynomials

$$p_0x^m+p_1x^{m-1}+ \dots +p_{m-1}x+p_m$$

and

$$b_0x^5+b_1x^4+ \dots +b_4x+b_5$$

* In fact a formal parameter may be specified as being a **label** or a **string**, although these basic words are not allowed in declarations.

† However, if this identifier were to appear on the right-hand side of an assignment statement, it would be interpreted as an attempt to execute this procedure (see page 96). This explains why we have introduced the local variable $p$ in Fig. 9.8.

for $x = x_0, x_0+h, x_0+2h, \ldots, x_0+rh$. The procedure declarations will be precisely those given above; the programme itself follows the lines of Fig. 3.4 (page 16), except that we now tabulate two polynomials side by side. The reader should note the use of the function procedure *polynomial* as the argument of the print function.

The data tape for this programme would require a sequence of numbers representing respectively:

$$m, p_0, p_1, \ldots, p_m, b_0, b_1, \ldots, b_5, x_0, h, r.$$

```
begin integer m;  m := input;  comment this outer block serves to define the value of the dynamic
subscript bound  m  which is used in the next (inner) block;
  begin array p[0:m], b[0:5];  real x, h;  integer r, j;
    procedure coeffin(n,a);  integer n;  array a;
      begin integer i;
        for i := 0 step 1 until n do a[i] := input
      end;
    real procedure polynomial(x,n,a);  real x;  integer n;  array a;
      begin integer i;  real p;  p := a[0];
        for i := 1 step 1 until n do p := p×x+a[i];
        polynomial := p
      end of declarations;
    coeffin(m,p);  coeffin(5,b);
    x := input;  h := input;  r := input;
    for j := 0 step 1 until r do
      begin newline;*  print(x);
        print(polynomial(x,m,p));
        print(polynomial(x,5,b));
        x := x+h
      end
  end
end;
```

Fig. 9.9. Tabulation of two polynomials.

Our programme illustrates that the actual parameters used in a procedure call need not have the same names as the corresponding formal parameters in the procedure declaration. The procedure call will cause the actual parameters to be substituted for the formal parameters during execution of the procedure body. It will be clear that each of the actual parameters must be of the same sort as the corresponding formal parameter in the procedure declaration.†

For a formal parameter specified as real, the corresponding actual parameter may be a constant,

* Since Algol does not define a notation for output operations, we have arbitrarily introduced a procedure *newline* (assumed to be available without declaration) which has the effect of starting a new line on the printer.

† To separate from each other any two parameters given in brackets after the name of a procedure we may either use a comma (as in the examples above) or the *parameter delimiter*

)*aa . . . a*:(

where *aa . . . a* denotes any combination of letters. Such a parameter delimiter is purely explanatory, and may be used to indicate the role of the parameter that follows it; e.g. instead of *polynomial* (*x*,*n*,*a*) we could write:

*polynomial* (*x*) *degree*: (*n*) *coefficients*: (*a*)

It will be seen that in each case parameters are enclosed in matching brackets.

a variable, an arithmetic expression, or a function or function procedure with its argument (or parameters) in brackets. Let us consider the procedure call:

*polynomial*(*cos*(*y*), 8, *a*);

In executing this call, the function *cos*(*y*) would be substituted for *x* in the procedure body, and this means that the function *cos*(*y*) would be calculated 8 times in the execution of the call *polynomial*(*cos*(*y*), 8, *a*), even though *y* may have the same value each time. To avoid these repeated evaluations of the same function, Algol allows us to indicate that a parameter does not change its value during execution of the procedure, by putting the corresponding formal parameter in a *value list* in the procedure heading. Such parameters are then said to be *called by value*. This means that, in executing the procedure, the formal parameter is initially assigned the *value* of the corresponding actual parameter at the beginning of the procedure call, and there is then no further reference to this actual parameter. A formal parameter included in the value list is effectively treated as a local variable of the procedure body. By contrast, any parameter *not* included in the value list is said to be *called by name*, which means that the rule for evaluating the actual parameter is referred to on each occurrence of the formal parameter in the procedure body.

The value list, if any, consists of the symbol **value** followed by the list of formal parameters which are to be called by value; all such parameters must also be specified. The value list is placed in the procedure heading immediately after the name of the procedure, e.g.

**real procedure** *polynomial*(*x*,*n*,*a*); **value** *x*; **real** *x*;

An interesting feature of Algol is that it can define a function *recursively*. Consider again the evaluation of the polynomial

$$p_n = a_0x^n+a_1x^{n-1}+ \ldots +a_{n-1}x+a_n.$$

The procedure given in Fig. 9.8 is equivalent to the following sequence of steps:

$$\begin{aligned} p_0 &= a_0 \\ p_1 &= p_0x+a_1 \\ p_2 &= p_1x+a_2 \\ &\cdot\ \cdot\ \cdot \\ p_n &= p_{n-1}x+a_n. \end{aligned}$$

We may regard the two formulae

$$\begin{aligned} p_n &= p_{n-1}x+a_n \quad (\text{for } n\geqslant 1) \\ p_0 &= a_0 \end{aligned}$$

as giving a *recursive definition* of $p_n$, and express this in Algol as the *recursive procedure* of Fig. 9.10. Note that the procedure body now consists of a single statement, so that the usual statement brackets **begin** and **end** can be omitted.

```
real procedure polycur(x,n,a); value x,n; real x; integer n; array a;
  if n = 0 then polycur := a[0]
       else polycur := polycur(x,n−1,a)×x+a[n];
```

Fig. 9.10. Recursive procedure for evaluating polynomial.

The procedure is called recursive because the procedure body contains a call for itself, namely *polycur*(*x*,*n*−1,*a*). At each such call the value of the second parameter is reduced by unity, until eventually the condition *n* = 0 is satisfied. The recursive procedure *polycur*(*x*,*n*,*a*) will give the same result as the procedure *polynomial*(*x*,*n*,*a*) of Fig. 9.8. Recursive procedures are usually inefficient as regards computing time, but they do provide the programmer with a useful tool in certain applications.

## 9.7 Conditional Statements and Switches

The example of section 9.1 contained a conditional statement of the form **if** . . . **then** . . . ; to illustrate the other kind of conditional statement available in Algol, we shall now give a programme for factorizing an integer. After reading the integer from an input device and testing that it is positive, the programme continues exactly as shown in Fig. 2.2 of Part I. By referring to the flow diagram given on page 8, the reader should be able to interpret the construction

**if** . . . **then** . . . **else if** . . . **then** . . . **else** . . .

which constitutes the main part of our Algol programme:

```
begin integer n, q, m;  n := input;  if n⩽0 then stop;*
even:   q := n÷2;
        if n−2×q = 0 then
           begin print(2);  n := q;  go to even end;
        m := 3;
odd:    q := n÷m;
        if n−m×q = 0 then begin print(m);  n := q;
                                  if n ≠ 1 then go to odd
                          end
           else if q<m then print(n)
                   else begin m := m+2;  go to odd end
end;
```

Fig. 9.11. Factorize.

Our next example is somewhat artificial. It supposes that we have a compiler which does not fully implement the basic function *sin*(*x*), but provides instead a function *sinq*(*x*) for which the argument *x* must be an angle in the first quadrant and must be expressed in degrees. The following conditional statement then serves to make the variable *f* equal to the value of the sine of an angle *d* degrees, where $0 \leqslant d < 360$:

```
  if d⩽90 then f := sinq(d)
     else if d⩽180 then f := sinq(180−d)
          else if d⩽270 then f := −sinq(d−180)
               else f := −sinq(360−d);
```

Fig. 9.12.

This example will also serve us to explain the use of a *switch*. We can construct an array whose elements are *labels* by means of a *switch declaration*, e.g.

**switch** *quadrant* := *first*, *second*, *third*, *last*;

the instruction **go to** *quadrant*[*q*] then acts as a switch by leading directly to the statement which carries the $q^{th}$ label of the switch list. We may now write our programme as follows:

* We are assuming the availability of a procedure *stop*.

```
begin integer q;
      switch quadrant := first, second, third, last;
            q := entier(d/90)+1;  go to quadrant[q];
first:      f := sinq(d);  go to finish;
second:     f := sinq(180−d);  go to finish;
third:      f := −sinq(d−180);  go to finish;
last:       f := −sinq(360−d);
finish: ;*
end;
```

Fig. 9.13. Example of a switch.

Algol provides *conditional expressions* as well as conditional statements. An example of a conditional expression is the right-hand side of the assignment statement

$$h := \textbf{if } t<0 \textbf{ then } 0 \textbf{ else } 1$$

which has the same effect as the conditional statement

$$\textbf{if } t<0 \textbf{ then } h := 0 \textbf{ else } h := 1;$$

it is important to realize that a conditional expression must contain both **then** and **else**, as otherwise it would not be fully defined. If the expression following **then** is itself conditional, this should be enclosed in *round* brackets to avoid possible ambiguities as to the meaning of **else**.

The procedure body of Fig. 9.10 may now be written as

$$polycur := \textbf{if } n = 0 \textbf{ then } a[0] \textbf{ else } polycur\ (x,n-1,a)\times x+a[n];$$

Conditional expressions may also be used as parameters of a procedure call (or as the argument of a function).

## 9.8 Logical Operators and Boolean Variables

We shall now consider the structure of the conditions used in *if* clauses. Such a condition is either satisfied or it is not satisfied, and accordingly we say that it has one or other of the two *logical values* **true** and **false**. The simplest form of condition is a relation between two arithmetic expressions, and Algol recognizes the six *relational operators* $<$, $\leqslant$, $=$, $\geqslant$, $>$, $\neq$. Compound conditions may be formed as combinations of such simple relations. Thus the compound condition $0\leqslant x\leqslant 1$ is satisfied provided *both* the simple conditions

$$0\leqslant x \quad \textit{and} \quad x\leqslant 1$$

are satisfied. The word *and* in the previous line may be regarded as a logical operator connecting two conditions, and as such it is denoted by the symbol $\wedge$. The condition that $x$ should lie in the range $0\leqslant x\leqslant 1$ may therefore be expressed as

$$(0\leqslant x)\wedge(x\leqslant 1)$$

or, since arithmetic relations take precedence over logical operators, we may write it without brackets as

$$0\leqslant x\wedge x\leqslant 1.$$

The condition that $x$ should lie *outside* this range may be stated in the form that *either*

$$x<0 \quad \textit{or} \quad 1<x$$

should be satisfied. The word *or* is also used as a logical operator in this sense, and is then denoted

* Algol allows a label to be placed in front of an empty statement, called a *dummy statement*.

by the symbol $\vee$. This enables us to write the condition that $x$ should lie *outside* the range $0 \leqslant x \leqslant 1$ as

$$x < 0 \vee 1 < x.$$

Yet another logical operator is one that turns any condition into its converse. It is denoted by the symbol $\neg$ (pronounced *not*). By placing this in front of our last compound condition we have an alternative form for the condition $0 \leqslant x \leqslant 1$, namely

$$\neg(x < 0 \vee 1 < x).$$

Algol recognizes the five logical operators $\neg, \wedge, \vee, \supset, \equiv$ as defined by the *truth table* of Fig. 9.14; they are called *Boolean* operators.*

A condition may be regarded as a *Boolean expression,* and the truth-value of such an expression may be assigned to a *Boolean variable,* just as the numerical value of an arithmetic expression may be assigned to a real variable. Boolean variables must be declared as **Boolean**, and Algol also allows the declarations **Boolean array** and **Boolean procedure**. An example of a Boolean assignment statement is

$$B := abs(x) < 1$$

which is equivalent to

**if** $abs(x) < 1$ **then** $B :=$ **true else** $B :=$ **false**

Note that if the value of $x$ is changed *after* execution of this statement, the value of $B$ would not be affected.

Two Boolean variables $B1$ and $B2$ may be compounded by means of the logical operators, and the values taken by the resulting expressions are given in the following *truth table*:

| | B1 | false | false | true | true |
|---|---|---|---|---|---|
| | B2 | false | true | false | true |
| not: | $\neg$ B1 | true | true | false | false |
| and: | B1 $\wedge$ B2 | false | false | false | true |
| or: | B1 $\vee$ B2 | false | true | true | true |
| implies: | B1 $\supset$ B2 | true | true | false | true |
| equivalent: | B1 $\equiv$ B2 | true | false | false | true |

Fig. 9.14. Truth-table for logical operations.

We shall now give a procedure (based on the method described on page 18 of Part I) for sorting a set of positive numbers $a[1], a[2], \ldots, a[n]$ into *either* ascending *or* descending order of magnitude, according as the value corresponding to the formal parameter $U$ is **true** or **false**.

```
procedure sort(a, n, U); array a; integer n; Boolean U;
  begin integer m,h,k; real extr;
    for h := n step -1 until 2 do
      begin m := 1; for k := 2 step 1 until h do
        if U then begin if a[k]>a[m] then m := k end
          else if a[k]<a[m] then m := k;
        extr := a[m]; a[m] := a[h]; a[h] := extr;
      end
  end;
```

Fig. 9.15. Sort up or down.

* After the mathematician George Boole (1815-1864).

The statement

**if** $U$ **then begin if** . . . **then** . . . **end**
**else if** . . . **then** . . . ;

of Fig. 9.15 could be replaced by the more compact statement

**if** $U \equiv a[k] > a[m]$ **then** $m := k$;

in the case of $U$ being **false**, the statement $m := k$ would then be executed if $a[k] \leqslant a[m]$.

Having declared a procedure *sort* as above, we could put a sequence of numbers $s[1], \ldots, s[p]$ into *descending* order by means of the statement

*sort*($s$, $p$, **false**);

## 9.9 Implementation

Although Algol is a language for describing computational processes without reference to any particular machine, an Algol programme is presumably intended to be used on some machine which can translate Algol into its own code and then perform the desired computation.

Algol recognizes that a *translator* for a particular machine must have its own rules for input and output procedures, and also its own conventions for representing the 116 basic symbols of Algol. Actual translators may impose a few further restrictions on the full generality of Algol to take account of the finite size of the store and also to simplify the process of translation and obtain a more efficient programme. The different conventions adopted for different makes of machine may be compared with variations of dialect between different groups of people talking the same language.

The programmes and procedures given in this chapter have all been tested on the Elliott 803 computer in the University of Reading after making such minor changes as are required by the conventions of the Elliott Algol compiler.

## Conclusion

This chapter is intended as an introduction to Algol. We have described all the main features of the language, but without necessarily defining in detail the effect of some of the rather subtle constructions for which the Algol Report[**14**] provides a definition.

APPENDIX

# PAPER TAPE CODES

| | *The Ferranti 5-hole paper tape code* | | *"Numerical Value"* | | *The Elliott 5-hole paper tape code* | |
|---|---|---|---|---|---|---|
| | *Figures* | *Letters* | | | *Figures* | *Letters* |
| ○○○.○○ | Figure Shift | | 0 | ○○.○○○ | Blank Tape | |
| ○○○.○● | 1 | A | 1 | ○○.○○● | 1 | A |
| ○○○.●○ | 2 | B | 2 | ○○.○●○ | 2 | B |
| ○○○.●● | * | C | 3 | ○○.○●● | * | C |
| ○○●.○○ | 4 | D | 4 | ○○.●○○ | 4 | D |
| ○○●.○● | ( | E | 5 | ○○.●○● | <(or $) | E |
| ○○●.●○ | ) | F | 6 | ○○.●●○ | = | F |
| ○○●.●● | 7 | G | 7 | ○○.●●● | 7 | G |
| ○●○.○○ | 8 | H | 8 | ○●.○○○ | 8 | H |
| ○●○.○● | $\neq$ | I | 9 | ○●.○○● | ' | I |
| ○●○.●○ | = | J | 10 | ○●.○●○ | , | J |
| ○●○.●● | − | K | 11 | ○●.○●● | + | K |
| ○●●.○○ | $\nu$(or $\approx$) | L | 12 | ○●.●○○ | : | L |
| ○●●.○● | Lf | M | 13 | ○●.●○● | − | M |
| ○●●.●○ | Sp | N | 14 | ○●.●●○ | . | N |
| ○●●.●● | , | O | 15 | ○●.●●● | >(or %) | O |
| ●○○.○○ | 0 | P | 16 | ●○.○○○ | 0 | P |
| ●○○.○● | > | Q | 17 | ●○.○○● | ( | Q |
| ●○○.●○ | $\geqslant$ | R | 18 | ●○.○●○ | ) | R |
| ●○○.●● | 3 | S | 19 | ●○.○●● | 3 | S |
| ●○●.○○ | → | T | 20 | ●○.●○○ | ? | T |
| ●○●.○● | 5 | U | 21 | ●○.●○● | 5 | U |
| ●○●.●○ | 6 | V | 22 | ●○.●●○ | 6 | V |
| ●○●.●● | / | W | 23 | ●○.●●● | / | W |
| ●●○.○○ | ×(or $\psi$) | X | 24 | ●●.○○○ | @ | X |
| ●●○.○● | 9 | Y | 25 | ●●.○○● | 9 | Y |
| ●●○.●○ | + | Z | 26 | ●●.○●○ | £ | Z |
| ●●○.●● | Letter Shift | | 27 | ●●.○●● | Figure Shift | |
| ●●●.○○ | . | . | 28 | ●●.●○○ | Space | |
| ●●●.○● | $n$(or ') | ? | 29 | ●●.●○● | Carriage Return | |
| ●●●.●○ | Cr | £(or $\pi$) | 30 | ●●.●●○ | Line Feed | |
| ●●●.●● | Erase | | 31 | ●●.●●● | Letter Shift | |

Sprocket holes are represented by . and holes by ●. Both codes use an odd number of holes to represent the digits 0,1,2, . . . , 9 to guard against occasional failures of the input and output equipment. The most common type of fault is losing or gaining a single hole and this would give a non-numerical character.

# REFERENCES AND SUGGESTIONS FOR FURTHER READING

*Part I.*

1. S. H. Hollingdale, High Speed Computing: Methods and Applications; E.U.P., 1959.
2. R. K. Livesley, An Introduction to Automatic Digital Computers; Cambridge U.P., 1957.
3. K. A. Redish, An Introduction to Computational Methods; E.U.P., 1961.
4. P. Henrici, Lecture Notes on Elementary Numerical Analysis; Wiley, 1962.
5. National Physical Laboratory, Modern Computing Methods; H.M.S.O., 1961.
6. J. Todd (ed.), Survey of Numerical Analysis; McGraw-Hill, 1962.

*Part II.*

7. A specification of the Mark 3 Autocode for the 803 Electronic Digital Computer; Elliott Computing Division (Borehamwood, Herts.), 1962.

8a. The Pegasus Autocode; List CS217A, Ferranti Ltd. (68 Newman Street, London, W.1), 1959.

8b. Description of the [Sirius] autocode; List CS302, Ferranti Ltd., 1961.

8c. Extension of the [Sirius] Autocode; List CS334, Ferranti Ltd., 1962.

9. Mercury Autocode Manual; List CS242A, Ferranti Ltd., 1959.
10. D. D. McCracken, A Guide to FORTRAN Programming; Wiley, 1961.

*Part III.*

11. H. Bottenbruch, Structure and Use of ALGOL 60; Journal of the ACM, **9** (Apr. 1962), 161-221.
12. E. W. Dijkstra, A Primer of ALGOL 60 Programming; Academic Press, 1962.
13. D. D. McCracken, A Guide to ALGOL Programming; Wiley, 1962.
14. P. Naur (ed.), Revised Report on the Algorithmic Language ALGOL 60; Computer Journal, **5** (January 1963) 349-367; also available in Communications of the ACM, **6** (January 1963) 1-17; and in Numerische Mathematik, **4** (January 1963), 420-453.

# INDEX

The index is given in tabular form to allow easy cross-referencing between the four programming languages of Parts II and III. Where different words are used to describe similar concepts, corresponding words are listed instead of page references; these words may then be found in their own right elsewhere in the index. Words in CAPITALS are instructions of an autocode, and words in **bold** type are basic Algol symbols. There is *no* index to Part I.

A list of examples, in order of their appearance, is given at the end, with page numbers and an indication showing which programming language has been used.

| | Pegasus-Sirius | Elliott 803 | Mercury | ALGOL |
|---|---|---|---|---|
| Expression | | | | |
| Arithmetic expression | | | | 88–89 |
| Conditional expression | | | | 98 |
| General expression | | | 58 | } Arithmetic |
| Index expression | | | 57 | } expression |
| **for** statement | | CYCLE | Cycle | 90 |
| Function procedure | | | | 94 |
| Functions | 31, 40–41 | 50 | 60, 66 | 89 |
| Global variables | | | | 93 |
| **go to** | Jump | JUMP | JUMP | 88, 93 |
| HALT | STOP | WAIT | 68 | |
| HOOT | | | 68 | |
| Identifier | | | | 87 |
| Index | 29 | Integer variable | 56 | **integer** |
| Input | 34–35 | 47 | 69 | 89 |
| Integer, **integer** | index | 43 | index | 88 |
| Interlude | 37 | | | |
| Jump, JUMP | 30–31, 39, 41 | 51 | 61–62 | **go to** |
| JUMPDOWN | | SUBR | 73–74 | procedure call |
| Label (for data) | Name | 47 | | |
| Label (for instructions) | 30 | Reference number | 61, 74 | 88, 93 |
| Local variables | | | | 92 |
| Logical operators | | | | 98–99 |
| Loop stop | 31 | STOP | END | |
| Modifier | 29 | Suffix | Suffix | Subscript |
| Name | 38 | Label (for data) | TITLE (Directive) | |
| Number | 34, 40 | 43–44, 47 | 56, 69 | 89 |
| Output | 35–37 | 47–48 | 70–72 | 89 |
| Optional output | 37 | 54 | 71–72 | |
| **own** | | | | 93 |
| Parameters | | | | 94–95 |
| PRESERVE | | | 79 | |
| Procedure, **procedure** | | subroutine | (sub) routine | 93–96 |
| procedure call | | | | 94 |
| recursive procedure | | | | 96 |
| Quicky | | | 81 | |
| **real** | variable | variable, floating-point | variable | 88 |
| Reference number | Label | 42, 46 | Label | Label |
| REPEAT | | 43, 53–54 | 63–66 | |
| RESTORE | | | 79 | |
| RETURN | | EXIT | 73–74 | |
| ROUTINE | | | 74, 81–82 | **procedure** |

## *List of Examples*

| | |
|---|---|
| Least Prime Factor of $n$. | 6–7 |
| Factorizing integers. | 6, 8, 19–20, 52(E), 97(A) |
| Calculating $(1+1/n)^n$. | 9, 32(P), 42–43(E), 62–63(M) |
| Evaluation of $f(x,y,z)$. | 9–11 |
| Iterative method for square root. | 11–12, 31(P), 62(M), 91(A) |
| Mean, variance, standard deviation. | 11–12, 36(P), 49(E), 69(M), 87–88(A), 90(A) |
| Adding a sequence of numbers. | 14–15, 45(E), 66(M) |
| Evaluating a polynomial. | 14–15, 32–33(P), 46(E), 66(M), 94(A), 96(A), 98(A) |
| Input of a set of numbers. | 14–15, 34(P), 69(M), 78(M), 93(A) |
| Tabulating a polynomial. | 16, 94–95(A) |
| Input of a matrix. | 17, 67(M), 91–92(A) |
| Largest element of sequence. | 18, 66(M), 91(A) |
| Sorting numbers into order. | 18, 33(P), 66–67(M), 99–100(A) |
| Example of a switch. | 19, 68(M), 97–98(A) |
| Solution of quadratic equations. | 65(M) |
| Table of functions with layout. | 71(M) |
| Subroutine for cube root. | 73(M) |

A = Algol. E = Elliott Autocode. M = Mercury Autocode. P = Pegasus Autocode