AGRenton

ENGLISH ELECTRIC LEO MARCONI

KDF 9

ALGOL programming

An English Electric-Leo-Marconi mini-manual

KDF 9

ALGOL programming

ENGLISH ELECTRIC-LEO-MARCONI COMPUTERS Ltd.

KIDSGROVE STOKE-ON-TRENT STAFFORDSHIRE   Telephone: Kidsgrove 2141

Publication 1002 mm (R)   1000565   price: five shillings

ALGOL PROGRAMMING FOR KDF 9

C O N T E N T S

CONTENTS
(Continued)

CONTENTS
(Continued)

LIST OF ABBREVIATIONS USED IN THIS MANUAL

A	array identifier
AE	arithmetic expression
AN	array name
AP	actual parameter
BE	boolean expression
BS	basic statement
BV	boolean variable
DE	designational expression
DM	number of dimensions
DV	device number
FE	format expression
FL	for list
FLE	for list element
FP	formal parameter
FS	for statement
L	label
LAY	layout
LB	lower bound of subscript
LO	logical operator
LS	letter string
LV	logical value
N	unsigned number
O	arithmetic operator
P	procedure identifier
R	relation
RO	relational operator
S	statement
SAE	simple arithmetic expression
SBE	simple boolean expression
SDE	simple designational expression
ST	string
SUB	subscript
SW	switch identifier
T	type
UB	upper bound of subscript
UBS	unlabelled basic statement
US	unconditional statement
V	variable

1. AUTHOR'S INTRODUCTION 1.

ALGOL 60 is a programming language for describing numerical processes and has the unique advantage of international recognition as a common language. ALGOL's inherent merits are the bases of its widening acceptance for both scientific and engineering applications. The power of its statements often surprises newcomers. Its conciseness avoids much of the tedium in other forms of programming, simplifies the programming of complex problems, and makes it an acceptable medium for solving the occasional problem. The use of conventional symbols of mathematics and the borrowing of ordinary English words to form ALGOL symbols helps to make an ALGOL program easy to read and understand. The ALGOL identifiers are much more easily recognised and distinguished by the human eye than the numerical storage representation of computer codes. These advantages, by making programming easier, also enable a program to be written in a shorter time and result in fewer mistakes.

Of particular value are the ALGOL programs and procedures published throughout the world, which are immediately available to the user of the language. By their means he has access to the work of recognised experts in the field of numerical analysis and to a wider variety of computer programs and techniques than can be obtained by using one machine code only.

The ALGOL 60 language is defined in an official publication entitled:

"Revised Report on the Algorithmic Language ALGOL 60"*

There are practical objections to the implementation of the complete language for programming use on KDF 9. For the information of those already familiar with ALGOL, KDF 9 ALGOL is a proper subset of ALGOL 60 consisting of the complete language restricted as follows:

(1) No integer labels.

(2) No own arrays with "dynamic bounds".

(3) Each formal parameter must appear in the specification part of the procedure; actual parameters corresponding to a formal parameter called by name to which assignments are made, or which is specified as an array, must have the same type as specified for the formal parameter.

(4) Actual procedures used in place of the same formal parameter must have similar specification parts.

The ALGOL 60 Report allows procedure bodies to be expressed in "non-ALGOL language". For KDF 9 this possibility offers two advantages:

(i) Procedure bodies in KDF 9 user code may be used for the realisation of input-output facilities, or perhaps to obtain increased speed of execution of computational procedures;

---

*The Computer Journal, Vol. 5 No. 4. January 1963.

1. Author's Introduction (cont.) 1.

(ii) Segmentation of large programs becomes convenient - a code body can be a call of an already translated ALGOL procedure; a preliminary description of segmentation is given in a KDF 9 Library Service note - ALGOL Note 1.

It must be understood that the input-output facilities described in this manual are not part of the language as such; they are provided in the form of procedures and consequently can be accepted, or rejected and others used in their place. In practice some procedures will come to be regarded as standard; but, equally, the range of available procedures may be extended to cover requirements as yet unforeseen.

Before a program written in the ALGOL language can be run on KDF 9 a compiler is needed to produce an equivalent program in machine code. Two such compilers accepting the same ALGOL programs are provided.* One aims at fast compiling and is of particular application to ALGOL programs in the testing stage. The other takes longer to compile but produces a faster object program, and is therefore more suitable at a later stage of program development. Given the Advance Control facility of KDF 9 it is predicted that there will be little difference in speed of operation between the translated program and an ordinary hand-coded version, when this second compiler is used.

The present edition of the manual is a revision of the "Simple Introduction to ALGOL Programming for KDF 9" (December 1961) and includes a description of those aspects of KDF 9 ALGOL omitted in that document. Subscripted variables and arrays are introduced immediately after ALGOL statements. The section on input and output of data is re-written to convey the new system as now being implemented. Switches and designational expressions are introduced in Section 21. This could be omitted at a first reading. Another new section (Section 23) on the advanced use of procedures discusses some important ideas, although there again paragraph 23.4 dealing with parameters which are switches or designational expressions could be omitted on first reading. Own variables, procedure bodies in code, and strings are considered in new appendices.

Systems adapted to the needs of KDF 9 users, programmers, and operators are being built around the KDF 9 ALGOL compilers both for compiling, testing and running programs. Since description of these is expected to appear in a separate publication it is not attempted here, except that the final section of this manual touches slightly upon testing facilities.**

For a working knowledge of ALGOL, merely reading the text of the manual is hardly sufficient. The reader should attempt at least a fair proportion of the problems and if possible find someone capable of correcting his answers.

---

*For descriptions of the compiling methods see the following papers:

(1) "A multi-pass translation scheme for ALGOL 60" by D. H. R. Huxtable and E. N. Hawkins, Annual Review in Automatic Programming, Vol. III, 1963.

(2) "The Whetstone KDF 9 ALGOL Translator" by B. Randell to appear in Automatic Programming Systems, A.P.I.C. Studies in Data Processing, Vol. IV.

**The description is now published in English Electric-Leo ALGOL Notes 1 and 3

1. Author's Introduction (cont.) 1.

We gratefully acknowledge that some of the examples and problems are due to Dr. P. Naur and are indebted to Mr. M. Woodger for reading the manuscripts of both editions. Comments on the first edition received from many different sources, especially Prof. H. Rutishauser, have greatly helped in making the present revision.

It is perhaps also appropriate here to acknowledge help and encouragement over long periods received by our compiler writers from Professors A. van Wijngaarden, E. W. Dijkstra, W. L. van der Poel, and Dr. Naur.

J. S. GREEN, Ph.D.
ENGLISH ELECTRIC-LEO COMPUTERS LTD.
Kidsgrove, Stoke-on-Trent, Staffs.

TWO SIMPLE PROGRAMS

Here is an ALGOL program:

```
begin  real x, y, z;
       open (20);
       x := read (20);
       y := read (20);
       close (20);
       z := x + y;
       open (10);
       output (10, z);
       close (10)
end
```

This program will read two numbers supplied by means of punched paper tape. It will add the numbers together and then punch the result on paper tape. (The output paper tape may be printed when desired on an off-line flexowriter).

The above program illustrates some of the elements of ALGOL programming which we shall now proceed to examine. The actual operations on the computer are stimulated by the statements:

x := read (20), y := read (20), z := x + y, and output (10, z).

The first two of these read two consecutive numbers from device number 20, a paper tape reader, respectively assigning them as values to x and y. The third statement

z := x + y

takes the values of x and y, adds them and assigns the result to the new variable z. Finally, the fourth statement takes the value of z and punches it out on a paper tape punch, device number 10, in a standard form.

The reader will note that besides the four statements which stimulate the actual operations of the program, it contains also the underlined words begin and end at the beginning and end of the program respectively, a rather odd phrase, real x, y, z, and four further statements containing the words 'open' and 'close'. The underlining indicates a word that is to be taken as a basic ALGOL symbol. The begin and end brackets, as they are called, are used to bracket together pieces of program which are to be treated as one whole.* In this particular case they enclose a single program.

*A vertical line is often inserted to connect a corresponding begin and end. This may help to improve the appearance of a program by showing up its structure, but it has no operational significance.

The phrase real x, y, z is called a declaration and the particular declaration given here states that the quantities represented by x, y and z are to be treated as ordinary numbers. Any arithmetic performed upon these numbers will use a floating decimal point. The statements using the word 'open' are concerned with preparing the reading and writing devices the numbers of which appear as arguments. Statements using the word 'close' shut down the devices specified. Finally, the reader will also notice a sprinkling of semicolons. These are used to mark the divisions between declarations and statements.

Here is another program:

```
begin   real x, y, z;
        open (20);
        x := read (20);
        y := read (20);
        close (20);
        z := (x↑2 + 3) × (x + 1) × (x × y - 2)/3;
        open (10);
        output (10, z);
        close (10)
end
```

This program again reads the values of x and y, but computes a much more complicated arithmetic expression before finally punching out the result. Writing the formula for z in normal mathematical form we have:

$$z = (x^2 + 3)(x + 1)(xy - 2)/3.$$

By comparing this with the ALGOL form readers will be able to appreciate the meaning of the ALGOL arithmetic operator symbols. This example serves to illustrate the inherent power of an ALGOL statement. Even more complicated expressions are allowed and the rules for constructing them will be described later.

In ALGOL the symbols x, y, and z are known as identifiers and in the two programs given here they represent variables which take numerical values. The 'read', 'output', 'open' and 'close' occurring in the programs are also called identifiers, but they are used to identify a particular process or procedure to be followed by the computer.

The reader may now wish to glance at a larger and more practical type of program. Such a program has been provided in Appendix 1 of this manual, but at this stage it is not expected to be fully understood by the reader.

In KDF 9 ALGOL it is possible to stipulate completely the form in which results are to be laid out when printed. The program of Appendix 1 contains such stipulations about layout, the result of which can be seen in the results sheet following the program. The headings, column layout, and line spacing are all fixed by statements in the program itself.

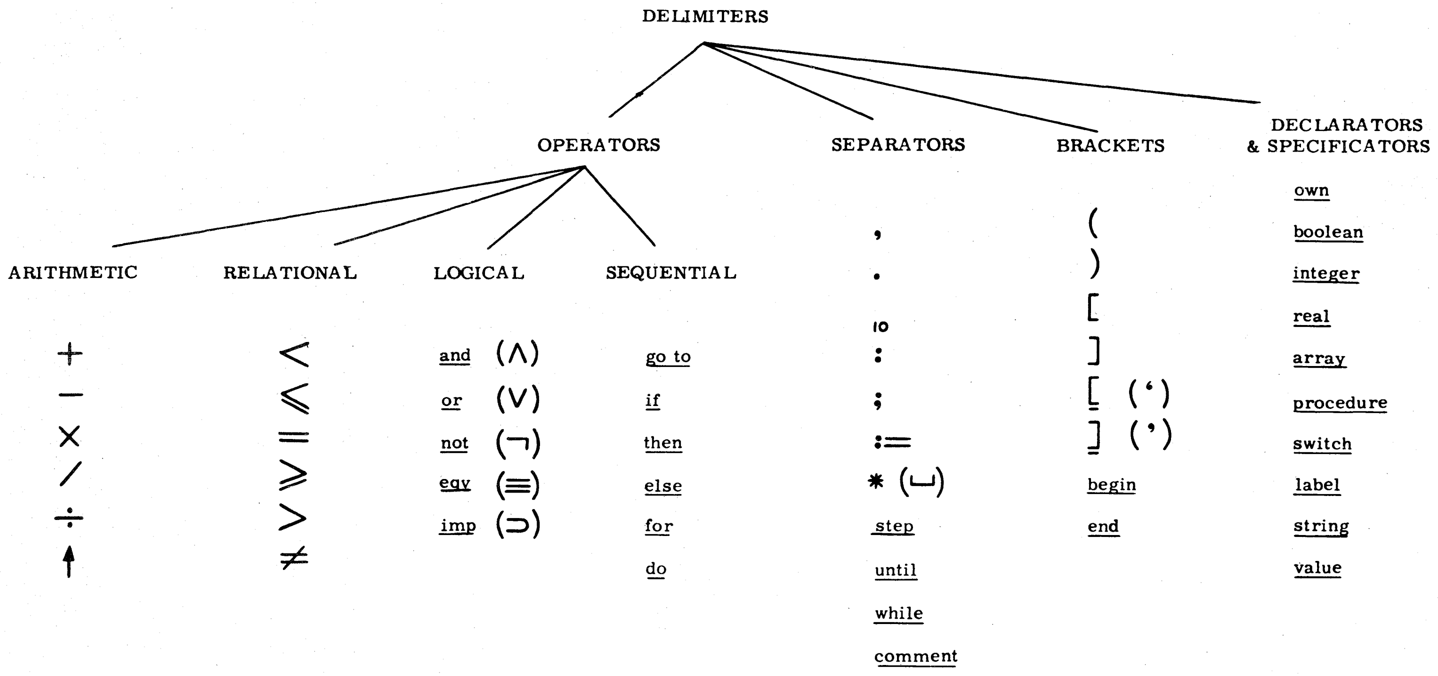


FIGURE 1 KDF 9 ALGOL DELIMITERS (8-CHANNEL PAPER-TAPE VERSION)

Wherever the KDF 9 ALGOL symbol differs from the corresponding symbol of the ALGOL 60 Reference Language, the latter is given in parentheses.

3. PROGRAM STRUCTURE AND BASIC SYMBOLS 3.

In the previous section we have attempted to convey some idea of the general structure of an ALGOL program by means of two simple examples. We propose in the following sections to examine the detailed grammatical structure of the ALGOL language, considering first the basic ALGOL symbols and in later sections the various language entities which may be built up from these symbols. The review includes:

1. Basic symbols.
2. Numbers.
3. Identifiers.
4. Expressions.
5. Statements.
6. Declarations, blocks and programs.
7. Procedures.

Some of the above entities will provide ideas new to the reader but few of them are inherently difficult to understand. Together with their associated rules they are required in order to systematize the expression of computational processes. Computers are not sufficiently versatile to absorb information about problems without such systematization. Because of this it is the duty of the programmer to obey the rules in formulating his problem.

The building bricks of an ALGOL program are called basic symbols. These are:

1. The letters of the English alphabet, both lower and upper case.
2. The digits 0 to 9.
3. The logical values <u>true</u> and <u>false</u>.
4. Symbols called delimiters.

Delimiters are:

1. Operators.
2. Separators.
3. Brackets.
4. Declarators and specificators.

Figure 1 lists delimiters in diagrammatic form. Some of the symbols in the diagram have a conventional significance and may be clear to the reader; others have no obvious significance. It may help him to note that sequential operators define the path to be taken through the program; separators serve the purpose of marking divisions between certain ALGOL entities, while declarators and specificators are symbols used to describe the properties of identifiers. The meaning of symbols not understood at this stage should become clear later.

All the basic symbols in KDF 9 ALGOL have been collected together in this section for reference. The ALGOL Report allows different 'hardware representations' for equipment with different sets of symbols and the set shown in Figure 1 is that available for use with eight-channel paper tape on KDF 9. Wherever this differs from the official ALGOL 60 reference language, the latter is given in parentheses. There is another 'hardware representation' available for equipment using five-channel paper tape listed in Appendix 8, but throughout the rest of this manual the eight-channel representation is used.

The reader should note that when words from the English language have been appropriated for use as basic symbols and given a particular ALGOL significance, they have also been distinguished by means of an underline. Thus the logical values <u>true</u> and <u>false</u> and delimiters such as <u>if</u>, <u>begin</u>, <u>integer</u>, and <u>label</u> must always retain their underlines.* They are treated as single symbols, the component letters having no individual significance.

Insertion of blank spaces makes no difference to the meaning or operation of any part of ALGOL.** This facility enables the programmer to write his program in a format which makes it easier for others to follow its course of action.

When used in a program the basic symbols are strung together in a linear sequence with appropriate spacing, making the end of one line of program continue on the next. As already stated, the basic symbols are used to build up decimal numbers, identifiers, expressions, statements, declarations, blocks, procedures, and, ultimately, programs according to certain rules. We now explain the purpose of these entities, and rules for their construction.

---

*Bold type is allowed in lieu of underline in published ALGOL 60 programs.

**When spaces with operational significance are required in strings (to be explained later), the symbol * is inserted to indicate their position.

4. DECIMAL NUMBERS 4.

Normal signed and unsigned decimal numbers using the digits 0 to 9 may be written in ALGOL and have the ordinary meanings.* A decimal point may only be used when it is followed by a fractional part, consisting of at least one digit. Use of decimal integer exponents is also allowed, and these must be written with the base 10 inserted below the line in small type, thus:

decimal number $_{10}$ integer exponent.

The exponent may be signed or unsigned. The preliminary decimal number may be omitted, while if the subscript $_{10}$ appears the integer exponent may not be omitted.

The following examples are allowed:**

0 16 $+6$ -79 $+10\ 127\ 783$

123.56 -0.00312 $+.65$ 74.0 $.00$

$7.5_{10}6$ $-33.261_{10}+2$ $+2_{10}-1$ $-.7_{10}5$ $_{10}-15$

The following examples are NOT allowed:

$23.$ $4_{10}.25$ $3\times_{10}2$ $14._{10}5$ $+6_{10}2.$ $10,000$

15_{10} $3.5_{10}(-7)$

Numbers in ALGOL and variables denoting numbers are said to be of type <u>integer</u> or <u>real</u>. Type <u>integer</u> refers to integers having neither exponent part nor decimal fraction part. Type <u>real</u> refers to any allowed form of number which is not of type <u>integer</u>. Integer arithmetic is normally used within the computer for type <u>integer</u> numbers and floating point arithmetic for type <u>real</u>.

Examples of type <u>integer</u>: 0 2 $+63$ -9710

Examples of type <u>real</u>: $+6.0$ -2.931 $6_{10}4$

The maximum working accuracy available in KDF 9 ALGOL for <u>real</u> quantities is between eleven and twelve significant decimal figures. An <u>integer</u> quantity must lie in the range -2^{39} to $+2^{39}-1$.

---

*Though defined in the ALGOL 60 Report, signed numbers are never in fact used in programs, signs always being invoked via the definition of expressions (See Section 6). They may however be used for input data or results in KDF 9 ALGOL.

**For the purpose of the lists of examples shown in this and the next section a string of five or more spaces is used to separate each example.

Problems

(1) Write numbers having the same values as the following, but which do not include an exponent part.

$+7.293_{10}8$	${}_{10}+3$	$-{}_{10}-6$
$98.12_{10}+2$	$-.1834_{10}-5$	$-4.8_{10}3$

(Solutions to Problems will be found in Appendix 7.)

(2) The values given by the following numbers may, in some cases, be expressed more economically by using a number with an exponent part. Show where this is the case.

17000	-0.00134	-0.0020041298
1000	1.0024	170

(3) Some of the following sequences of characters represent ALGOL numbers, some do not. Mark those which do.

-.0 08	-17.2.30	13.411 732
$+13.47_{10}+18$	$\underline{+}4.2$	$2.48_{10}n$
$4 \times {}_{10}-2$	$-88_{10}-7$	$\times 643.2$
(16.20)	$1,24_{10}3$	$12._{10}8$

5. IDENTIFIERS 5.

Mention of identifiers has already been made in Section 2. They may be single letters of the English alphabet, upper or lower case, or sequences of letters and numbers.* The first symbol of a sequence must be a letter. The following could be used as identifiers.

i	Days 1335
J1	exp
abCD43e	Delta alpha

In accord with Section 3 the spaces within the identifiers Days 1335 and Delta alpha are ignored by the ALGOL translator. Note that though the sequence A256b might be used as an identifier, 256b may not.

Identifiers are used for a variety of purposes. Amongst others, they may denote labels which mark reference points in the program, and they may also denote variable quantities which take a value in the usual mathematical sense.

An identifier which is a variable is said to be of type <u>real</u>, <u>integer</u>, or <u>boolean</u>. Variables of type <u>real</u> and <u>integer</u> were mentioned in Section 4. Variables of type <u>boolean</u> can take the logical values <u>true</u> or <u>false</u>. The means of defining the type of a variable will be explained in Section 18.

Problem

Some of the following sequences of characters can be used as identifiers, others cannot. Mark those which can.

begin	p7.2	7VPQ
a × v	Start value	<u>V</u>7
4711	number	a29v3
ppp3	Q(2)	epsilon

*Note that though the length of an identifier may be almost unlimited only the first eight characters are significant to the Whetstone produced KDF 9 ALGOL translator, while 155 characters are significant to that produced at Kidsgrove.

SIMPLE ARITHMETIC EXPRESSIONS

Numbers, those identifiers which represent variables, and other ALGOL entities having a single numerical value may be used in combination with arithmetic or logical and relational operators and certain sequential operators to form arithmetic expressions or boolean expressions. Initially we shall restrict our attention to subclasses of both these types of expression, namely, simple arithmetic expressions (considered in this section) and simple boolean expressions (considered in the next).*

It is even necessary to leave a general definition of simple arithmetic expressions in ALGOL to Section 8. However, we may now say that they include arithmetic expressions as understood in the normal mathematical sense, when these are written in the linear form which follows:

$$(\mathrm{O})\ \begin{matrix}\mathrm{N}\\ \mathrm{V}\end{matrix}\ \mathrm{O}\ \begin{matrix}\mathrm{N}\\ \mathrm{V}\end{matrix}\ \mathrm{O}\ \begin{matrix}\mathrm{N}\\ \mathrm{V}\end{matrix}\ \mathrm{O}\ \ldots\ldots\ \mathrm{O}\ \begin{matrix}\mathrm{N}\\ \mathrm{V}\end{matrix}\ \mathrm{O}\ \begin{matrix}\mathrm{N}\\ \mathrm{V}\end{matrix}$$

Here N stands for an unsigned number, V for a variable of <u>real</u> or <u>integer</u> type, and O for an arithmetic operator. The diagram is intended to indicate that N and V are interchangeable. The initial operator may only be an adding operator (+, -) and the broken parentheses indicate that in any case its presence is optional. At least one operand (N or V) must be present in an arithmetic expression.

Example:

$$2 \times x \uparrow 3 + n \div 2$$

The meanings of the operators used in this example are given below.

The order of execution of arithmetic operations follows certain definite rules. The operations are executed in order of occurrence from left to right unless the adjacent operation has a higher priority according to the following list:

1st	$\uparrow$
2nd	$\times$ / $\div$
3rd	+ -

Parentheses may be introduced within a simple arithmetic expression to override the order of evaluation given by the above rules provided that the enclosed symbols form a legitimate arithmetic expression. The arithmetic expression with its enclosing parentheses may be introduced within the simple arithmetic expression in any situation where an unsigned number or variable is allowed.

The arithmetic operators have the following meanings:

*We use this last phrase to maintain the parallel with simple arithmetic expressions. Both the old and the revised ALGOL 60 Reports use the phrase "simple Booleans".

$\uparrow$ is the sign of exponentiation. The base precedes the sign and the exponent follows. The operation is effected as in ordinary arithmetic with the following comments and exceptions. No values of base and exponent which would lead to infinite, indeterminate, or imaginary results are allowed, and when the exponent is real the value of the base may never be negative. The result of exponentiation is of the same type as the base, if the exponent is integer, and positive or zero. Otherwise the result is of type real.

$\times$ + - all have their conventional meanings. The type of the result is integer if both operands are integer, otherwise the result is real.

/ $\div$ both denote division. The first operator may be used with any combination of operands and produces a result of type real. The operator $\div$ is only used for two operands both of type integer and yields a result of type integer as follows:

$$n \div m = \text{sign}(n/m) \times \text{whole number part } (\text{modulus}(n/m))$$

The type of any result obtained by the operation of the above rules is as stated. If, for example, the result of some operation involving real type numbers happens to have an integer value, its type is not thereby changed from real to integer. In terms of the internal working of the computer, though the result happens to be an integer it is still in floating-point form.

Notes:

(1) In multiplication the multiplication sign must never be omitted. One may write $5 \times y$ and $(a + 2) \times b$, but not $5\ y$ and $(a + 2)b$.

(2) Two operators must not appear adjacent to one another. One may write $+3 \times (-x)$ and $y\uparrow(-4)$, but not $+3 \times -x$ and $y\uparrow -4$.

Examples of simple arithmetic expressions:

$$2 + 2\uparrow 3 = 2 + 8 = 10$$

$$(2 + 2)\uparrow 3 = 4\uparrow 3 = 64$$

$$1 + 2 \times 5 - 3\uparrow 2 = 1 + 10 - 3\uparrow 2 = 11 - 3\uparrow 2 = 11 - 9 = 2$$

The results of these three expressions are of integer type. The following give real type:

$$3/2 - .5 = 1.5 - .5 = 1.0$$

$$9\uparrow .5\uparrow 3 - 7 \div 2 = 3.0\uparrow 3 - 7 \div 2 = 27.0 - 7 \div 2$$
$$= 27.0 - 3 = 24.0$$

If $x = 4.5$, $y = 2.3$,

$$x + 3 \times y\uparrow 2\uparrow 2 = 4.5 + 3 \times 5.29\uparrow 2 = 4.5 + 3 \times 27.9841$$
$$= 4.5 + 83.9523 = 88.4523$$

Problems

(1) Evaluate the following expressions stating the type of the final result.

(i) $-4.6/4 \times (16 + 2)$

(ii) $+60 - 5 \times (3 + 2\uparrow(4 - 1))$.

(2) Some of the following sequences are arithmetic expressions, some are not. Mark those which are.

(i) $a \times b/c\uparrow d/e \times f$

(ii) $+a \times -b$

(iii) $2_{10}6 \times 4.3 + Q$

(iv) $2 \times {}_{10}6/4.3$

(v) $3.84_{10}(7 + n)/4$

(vi) $PQ\uparrow + 7.3$

(vii) $-(+(-v))$

(viii) $p/qrs \times tu - v$

(3) Assuming that at a certain point in a program the values of seven simple variables are as follows,

$va = 2,\ vb = 3,\ vc = 4,\ vd = 5,\ ve = 6,\ vf = 7,\ vg = 8,$

find the values of the following expressions:

(i) $va + vc \times vb/ve$

(ii) $vd \times (vc + vg)/ve/va$

(iii) $vc\uparrow(vd - vb)$

(iv) $vf\uparrow va \times (vf - vc)/vb/(vb + vc)$

(v) $vc\uparrow vb\uparrow va$

(vi) $(ve - vf - va) \div vc$

(vii) $vc \div (vg - vb)$

(viii) $(vg - vd)\uparrow vb \div ve$

(4) Write the following mathematical expressions as ALGOL expressions, without using redundant parentheses:

(i) $S + \dfrac{s - t}{v^2}$

(ii) $(U - W)\left(1 - \dfrac{a^3}{k(a - k)}\right)$

(iii) $a^{n + m}$

(iv) a^{b^n}

(v) $a^b + s^n$

6. <u>Simple Arithmetic Expressions</u> (cont.) 6.

(vi) $(q^v)^g$

(vii) $\dfrac{p^q}{r^{s + t}}$

(viii) $\dfrac{a - \dfrac{b}{c(d - e^{f + q})}}{h^{i(j - k)} + q^{\left(\frac{m}{n + p}\right)}}$

7. SIMPLE BOOLEAN EXPRESSIONS 7.

A boolean expression is a rule for computing a logical value. The result may be either the value true or the value false. The boolean expressions which occur in practice usually also belong to a subclass called simple boolean expressions.

A simple boolean expression consists most frequently of a single relation which takes the value true or false. By a relation we mean two simple arithmetic expressions separated by means of one of the relational operators:

$$< \quad \leqslant \quad = \quad \geqslant \quad > \quad \neq$$

These operators have their conventional mathematical meanings.

An example of a relation might be:

$$n = 0$$

This relation takes the value true if n is zero and the value false if n is not zero. Other examples might be:

$$n \times h \times (n \times h + 2 \times z) > 11.51$$
$$(a\uparrow 2 + b\uparrow 2)\uparrow 2 < a + b$$

The form of a relation can be depicted as follows:

SAE RO SAE

where SAE stands for a simple arithmetic expression and RO for a relational operator. In performing the operations involved in such a relation to find its logical value, the simple arithmetic expressions are evaluated first, from left to right, and the relational operation is performed last.

A simple boolean expression need not be a relation. It could be merely a logical value or a boolean variable. It could take a complicated form involving a number of relations, boolean variables and logical values, the values of which are operated upon by means of the logical operators not ($\neg$), and ($\wedge$) and or ($\vee$), amongst others. Appendix 2 describes the forms which are allowed, but the reader may wish to leave this to a second reading.

Problem

If i = 2, j = 3. x = 4.5 and y = 2.2, what are the values of the following simple boolean expressions:

(i) $i \times j \geqslant i + j$

(ii) $j/i < x/y$

(iii) $(x + y) \times (x - y) \neq 0$

(iv) $i - i \div 5 \times 5 = 0$

8·1 If Clauses

It is extremely useful in any programming language to be able to make a program choose its course of action depending upon the situation arising at run time. Such facility is allowed in ALGOL by means of the 'if clause'. Depending upon the truth or falsity of a boolean expression the program will either obey different instructions or supply the values of different expressions.

For example we might wish to have an arithmetic expression which supplies the value of $x^2 + 1$ if x is greater than zero but otherwise supplies the value -1. We could write the ALGOL expressions to do this as follows:

```
if x > 0  then  x↑2 + 1  else  -1
```

The first part of this expression,

```
if x > 0  then
```

is called an if clause. An if clause is always written in the form:

```
if BE then
```

where BE stands for a boolean expressions (as yet not fully defined but including simple boolean expressions). The basic symbols if and then are sequential operators.

Example of another if clause:

```
if lambda ≥ ·70710678 then
```

8·2 Use of the If Clause in Arithmetic Expressions

As already shown in the first of the above examples it is possible to extend the idea of the arithmetic expression by means of the if clause. Besides the simple arithmetic expressions considered in Section 6 it is also legitimate to have arithmetic expressions commencing with an if clause and completed by two alternative expressions, the first a simple arithmetic expression, the second itself an arithmetic expression. Thus, an arithmetic expression may be of the form,

```
if BE then SAE else AE
```

where SAE and AE stand for a simple arithmetic expression and arithmetic expression respectively.

Examples:

(a) $\underline{\text{if}}\ n = 0\ \underline{\text{then}}\ 0.5\ \underline{\text{else}}\ 1$

(b) $\underline{\text{if}}\ \text{lambda} \neq 0\ \underline{\text{then}}\ \text{alpha} \times (1 - \text{alpha}) \times \text{exp}(\text{lambda}\uparrow 2)/$
$(\text{lambda} \times 2.3282180)\ \underline{\text{else}}\ 0.48394$

In the above examples both the alternative expressions following then are simple arithmetic expressions. We could, however, take our more complicated definition for an arithmetic expression and use it for the expression following the symbol else. We might then obtain arithmetic expressions like those which follow,

(a)
$$\underbrace{\underline{\text{if}}\ i = 2\ \underline{\text{then}}\ \underbrace{p - q}_{\text{SAE}}\ \underline{\text{else}}\ \underbrace{\underline{\text{if}}\ i = 3\ \underline{\text{then}}\ p + q\ \underline{\text{else}}\ 0}_{\text{AE}}}_{\text{AE}}$$

(b) (Appendix 2 must be read in order to appreciate this example.)

$\underline{\text{if}}\ p < 0\ \underline{\text{and}}\ q < 0\ \underline{\text{then}}\ (p \times q - p + q)\uparrow 2$

$\underline{\text{else}}\ \underline{\text{if}}\ p > 0\ \underline{\text{and}}\ q > 0\ \underline{\text{then}}\ p \times (q + 1)\uparrow 2$

$\underline{\text{else}}\ \underline{\text{if}}\ p = 0\ \underline{\text{and}}\ q = 0\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ 0$

8•3 Use of the If Clause in Boolean Expressions

Boolean expressions run parallel to arithmetic expressions; they may use if clauses in a similar manner. Thus a boolean expression may be of the following form:

$$\underline{\text{if}}\ \text{BE}\ \underline{\text{then}}\ \text{SBE}\ \underline{\text{else}}\ \text{BE}$$

Of course, boolean expressions also include simple boolean expressions.

The following example shows a boolean expression of a fairly complex form. The quantities B1, B2, B3, B4, B5, B6 and B7 are variables of type boolean and some are used as simple boolean expressions and others as boolean expressions.

$$\overbrace{\underline{\text{if}}\ \overbrace{\underline{\text{if}}\ \underbrace{B1}_{\text{BE}}\ \underline{\text{then}}\ \underbrace{B2}_{\text{SBE}}\ \underline{\text{else}}\ \underbrace{B3}_{\text{BE}}}^{\text{BE}}\ \underline{\text{then}}\ \overbrace{B4}^{\text{SBE}}\ \underline{\text{else}}\ \overbrace{\underline{\text{if}}\ \underbrace{B5}_{\text{BE}}\ \underline{\text{then}}\ \underbrace{B6}_{\text{SBE}}\ \underline{\text{else}}\ \underbrace{B7}_{\text{BE}}}^{\text{BE}}}^{\text{BE}}$$

The grammatical structure has been indicated by means of bracketing.

8·4 A Use for Parentheses

An arithmetic expression or boolean expression commencing with an if clause may be incorporated within a simple arithmetic or simple boolean expression wherever a variable of the corresponding type is allowed, if and only if, it is enclosed within parentheses. This means that simple arithmetic and simple boolean expressions may be quite complicated. In practice such forms do occur occasionally. The following is an example of a simple arithmetic expression incorporating an arithmetic expression in parentheses.

$$J0 + (x - y) \times (\underline{if}\ n = 0\ \underline{then}\ 0.5\ \underline{else}\ 1)$$

Problems

(1) Write an arithmetic expression which will evaluate $\frac{2t}{1 + t^2}$

if A is greater than pi/2, otherwise will evaluate $\frac{1 - t^2}{1 + t^2}$

(2) Write down an arithmetic expression which will evaluate

$$\begin{array}{ll} x - 1 & (x < 0) \\ x^2 - 3x + 4 & (0 \leqslant x \leqslant 1) \\ x + 1 & (x > 1) \end{array}$$

STANDARD FUNCTIONS

The scope and value of ALGOL expressions are enhanced by a facility for inserting functions, just as variables may be inserted. One merely writes down a function's name with appropriate argument or arguments. Here we consider certain standard functions which may be used, although other functions are also available as explained in Section 19 on Procedures.

The standard functions are some of the more frequently occurring functions of analysis and are listed below:

abs (AE)	for the modulus (absolute value) of the value of the expression AE.
sign (AE)	for the sign of the value of AE (+1 for AE $>$ 0, 0 for AE = 0, -1 for AE $<$ 0).
sqrt (AE)	for the square root of the value of AE.
sin (AE)	for the sine of AE radians.
cos (AE)	for the cosine of AE radians.
arctan (AE)	for the principal value in radians of the arctangent of the value of AE.
ln (AE)	for the natural logarithm of the value of AE.
exp (AE)	for the exponential function of the value of AE.
entier (AE)	for the largest integer not greater than the value of AE.

These functions operate indifferently on arguments both of type real and integer, which must be arithmetic expressions. The functions all yield values of type real except for sign (AE) and entier (AE) which have values of type integer. When quoting a standard function within a program, it is unnecessary to make there any specification of the effect expected from this function.

The following examples show the use of standard functions in arithmetic expressions:

FUNCTION	ARITHMETIC EXPRESSION
abs (AE)	abs (1 - 2 × J1/J0)
sqrt (AE)	(1 - alpha)/ sqrt (2 × alpha)
exp (AE)	J0 + exp(- x↑2) × (**if** n = 0 **then** 0.5 **else** 1)

The effect of the function entier is shown by the following results:

entier (6.99) = 6

entier (-4.2) = -5

A useful expression is entier (x + 0.5) which takes the value of the nearest integer to x.

Problem

Write the following expressions in ALGOL using standard functions:

$$e^{2|\cos 3a|}, \qquad \sqrt{\left\{\log \arctan \sqrt{a^2 + b^2}\right\}},$$

$$\frac{a \cos x + b \sin x - 1}{a \cos^2 x + b \sin^2 x + 1}$$

STATEMENTS

Those assemblies of basic symbols which form units of operation within an ALGOL program are called statements. Statements written consecutively are usually also executed consecutively, and two independent statements written consecutively are always separated by a semicolon, thus:

```
S; S
```

The statements contained in the two simple programs of Section 2 obey this rule. We repeat the statements of the first of these programs below for the reader to note that this is so.

```
open (20);
x := read (20);
y := read (20);
close (20);
z := x + y;
open (10);
output (10, z);
close (10)
```

It is possible to write a statement containing other statements within itself by forming either a block or a compound statement and we shall consider these new ALGOL entities in later sections. We consider now some of the possible forms of the simple statement.

11· <u>ASSIGNMENT STATEMENTS</u> 11·

Some of the ALGOL statements appearing in the previous section are assignment statements, for example:

$$z := x + y$$

This statement is executed by giving the quantity z the value of x + y.

The symbol, :=, is the assignment symbol and is pronounced as "is assigned the value of" or "becomes".* The complete statement would read:

z is assigned the value of x + y.

The above example is particularly simple, but more complex forms are allowed. Thus, on the right hand side of the assignment symbol, one could have any arithmetic expression, or even a boolean expression if the variable on the left side were also of type <u>boolean</u>. It is possible to extend the left hand side by writing down a list of variables (called left part variables) with assignment symbols inserted to separate each from its neighbour. The value of the expression is then assigned to all the left part variables.

The assignment statement in its general form may therefore be illustrated as follows using V to stand for any variable:

$$\underbrace{V := \quad V := \ldots\ldots\ldots \; V :=}_{\text{Left part list}} \begin{cases} AE \\ BE \end{cases}$$

The name left part list is given to the list of all the left part variables together with the assignment symbols as shown in the diagram.

Examples of Assignment Statements:

$$h := 0.1$$

$$JO := n := 0$$

$$x := z + n \times h$$

$$JO := JO + \exp(-x \uparrow 2) \times (\underline{\text{if}}\ n = 0\ \underline{\text{then}}\ 0.5\ \underline{\text{else}}\ 1)$$

$$B1 := B2 := B3 := \underline{\text{false}}$$

$$Bool := n \neq m + 1$$

Types in assignment statements must obey certain rules, for the most part a fairly obvious set. Thus, all variables in a left part list must be of the same type. If the variables are <u>boolean</u>, so must be the expression on the right. If the variables are of type <u>real</u> or <u>integer</u> the expression must be arithmetic. However, it is allowable to have the

*The symbol := has a different meaning from the symbol =, denoting equality. The latter asserts the current situation to have a particular property, while the former performs an operation which may change the current situation.

arithmetic expression differing in type from the variables. In this event it is understood that the numerical value of the expression is transferred to real variables and the largest integer not greater than AE + 0.5 to integer variables (that is, the nearest integer). Note that the rule for assignment of a real arithmetic expression to an integer variable does not correspond to the system followed in integer division.

Example:

Find the action of the following assignment statements, given that n is of integer type, and x and y real.

```
n := 1;
x := 3.2;
y := x + 1;
n := n ÷ 2;
x := x + y /2↑(n + 1);
y := if n = 0 then x + y else x - y
```

We form a table containing a column for each variable, in which each new value of this variable is entered.

n	x	y
1	3.2	4.2
0	5.3	9.5

Final values:- n = 0, x = 5.3, y = 9.5

Problems

(1) Using the scheme of the example above, follow the action of the following statements and find the final values of the variables. They are all to be taken as type real.

```
a := b := 7;
p := a + 3 × b - 2.3₁₀-1;
q := p + (a + 3)/(-b - 13);
a := p := q - b × 0.2
```

(2) Using the same system find the final values of real variables r1, ra, rb, and integer variables n, i, j.

```
n := 5;
r1 := n/(n + 15);
rb := n + 6/(6 × r1 + 0.5);
i := n := n - 2;
j := rb - i;
ra := (j - i) × r1 × (rb - 4);
r1 := ra + rb + n + i + j + 8 × r1;
rb := (r1 - rb × n + j - ra)↑(rb - j) + ra;
j := n := 1 + n ÷ (j - 3);
i := n + ra
```

(3) Using the same scheme again find the final values of the real variables ra, rb, the integer variable ia, and the boolean variables ba, bb.

```
ra := 7.5;
ia := 5;
rb := 3 × ra - 2 × ia;
ba := rb>ia and ia>ra;
ra := 2 × (ra - ia) -1;
ba := not ra>ia or ba;
bb := ba and rb>ia and ra<rb
```

(This example requires a knowledge of the contents of Appendix 2).

12· GOTO STATEMENTS AND LABELS 12·

12·1 Goto Statements

Goto statements usually take the form:

<u>goto</u> L

where L stands for any label. They interrupt the normal sequential flow in the execution of statements by causing a jump to the statement prefixed by the label L. The label L may be any identifier not used for some other purpose,* while the basic symbol <u>goto</u> may also be written as <u>go to</u> if desired. The following might occur as goto statements.

```
goto   R         goto   L25
goto   SKIP      goto   x y Z P
goto   Repeat
```

Corresponding to the label appearing in the goto statement, the same label must be inserted elsewhere in the program to specify the destination. Insertion of this label is performed by labelling statements, as now described.

12·2 Labelled Basic Statements

The two forms of statement, assignment and goto statements already mentioned are included in the category of unlabelled basic statements (UBS). A dummy statement also exists and is included in the same category. This is simply an empty space which does nothing.

A label may be prefixed to an unlabelled basic statement in the following way:

L : UBS

The result is called a basic statement (BS).

The general form for a basic statement allows any number of labels:

L : L : .... L : UBS

It also includes the unlabelled basic statement as a special case. (It will now be seen that a dummy statement might sometimes be useful for setting a label).

*In ALGOL 60 an unsigned decimal integer may also be used as a label but this is not allowed in KDF 9 ALGOL.

Problem

The following piece of program generates a sequence of values for SUM. Find the first four of these values. The variables p, q and SUM are real, while n is integer.

```
       n := 1;
       p := 0.5;
       SUM := 0;
       q := 1;
loop : SUM := SUM + q/n;
       q := q × p;
       n := n + 1;
       goto loop
```

<u>FOR STATEMENTS</u>

ALGOL has a special form which enables the execution of any statement to be repeated a number of times. This is called the 'for statement'. It is the most appropriate ALGOL equivalent to a repeated loop in normal machine language programs and next to the assignment statement it is probably the most valuable statement available.

Two simple examples will help to explain what a for statement is like before we define the general form.

<u>for</u> i := 1 <u>step</u> 1 <u>until</u> 5 <u>do</u> x := x + 12

<u>for</u> x := 2 <u>while</u> y<0 <u>do</u> y := y + x

The first of these for statements increases the value of x by 12 on five consecutive occasions at the same time incrementing the value of i by 1 until it becomes greater than 5. The second example repeats the statement, y := y + x, for as long as y<0, keeping x at the value 2 during that time.

13.1 <u>The General Form of the For Statement</u>

The general form of the for statement is written as follows:

<u>for</u> V := FL <u>do</u> S

where V stands for a controlled variable and S for any statement. FL stands for a 'for list' which we now explain.

The for list is constructed of for list elements (FLE) according to the form,

FLE, FLE, ... FLE

A for list element may take any of the following forms:

AE

AE <u>step</u> AE <u>until</u> AE

AE <u>while</u> BE

where the basic symbols <u>step</u>, <u>until</u> and <u>while</u> are separators, separating the arithmetic expressions (AE) and boolean expression (BE).

The for list gives a rule for computing the values which are consecutively assigned to the controlled variable before each execution of the statement following <u>do</u>. This sequence of values is obtained from the for list elements by taking these one by one in the order in which they are written. The effect of the three types of for list element may best be explained by means of examples.

13.2 Arithmetic Expression Element

for x := (p + q)↑3, p × q **do** y := y/x

This for statement assigns the value of the arithmetic expression (p + q)↑3 to x and then executes the statement y := y/x. It then returns to the for list and assigns p × q to x. The statement y := y/x is again executed. The for statement has now been completed and control passes to the next statement in the program.

13.3 Step-until Element

for n := 1 **step** 1 **until** 10 **do** m := n↑3

This statement calculates the cubes of the first ten integers. It begins by assigning the value 1 to the controlled variable n and then obeys the statement following the symbol **do**. The controlled variable is now increased by a step of 1 making it n = 2. The statement is again obeyed using the new value of n, and n increased once more. The process continues until n becomes greater than 10 when the for statement is finished and all cubes up to 10^3 have been calculated (but only 10^3 remains as the value of m).

Any arithmetic expression may be used for the initial value, the step and the limit in the step-until element. This may lead to negative steps or even to steps which change sign during the execution of the for statement. A complete specification of the action in such circumstances is given in Appendix 3.

Problem

(1) Follow the action of the following statements

Delta x := 0.1;

for x := 0 **step** Delta x **until** 0.55 **do** y := (1 - x)↑2

(2) Write a for statement to add together the first n integers. First solve the problem by writing a preliminary assignment statement before the for statement, then find another solution which only requires a for statement.

13.4 While Element

for k := i + 1 **while** i × (i - 1)<20 **do** i := k + 2

In this for statement the value of the arithmetic expression i + 1 is repeatedly assigned to k and the statement i := k + 2 executed for as long as the boolean expression i × (i - 1)<20 is true. Assuming an initial value for i of i = 0, the following values of the variables are obtained.

	k	i(i - 1)	i
Initially	-	-	0
	1	0	3
	4	6	6
	7	30	

The values of i and k immediately before leaving the for statement are i = 6, k = 7.

Problems

Follow the action of the following for statements.

(1) **for** p := p + 2 **while** p↑2 + q↑2<100 **do** q := p + 1
where before entry p = 1 and q = -7.

(2) **for** i := 2, 5, 6 **step** 1 **until** 10, -1 **while** m<0 **do**
m := i x (i + 1)

(3) If p, q, r, s are **real** and k, m are **integer**, find the values assigned to controlled variables in the following for statements and the final value of s:

```
p := 1; q := 2; r := 3; s := 0;
for k := p + q, q - p, r × p - q do s := s + k;
for m := q step r until 7 × q + 1 do s := s - m;
for k := 2, s, 2 step 2 until 6 do s := s + 2 × k;
for m := s + 45, m + 2 while s<0 do s := s - m;
for k := 1 step 1 until 5 do
    for m := 3 step -1 until 0 do s := s + k + m
```

13.5 Miscellaneous Notes on For Statements

Note (1) In ALGOL the controlled variable has no defined value after the for statement has been completed by exhaustion of the for list. However, the value left by one for list element may be used in the next element of the same list, as in

for i := 1, i + 1 **while** ...

Note (2) Labels may be prefixed to a for statement. The complete form of a for statement is:

L : L : ... L : **for** V := FL **do** S

Note (3) Exits from within a for statement body, that is the statement following a do, by means of goto statements are allowed. In such an event the controlled variable keeps its current value on exit. (The importance of this and the following note will be better appreciated when later sections of the manual have been read and it is realised that a for statement body can contain many statements.)

Note (4) A goto statement outside a for statement may not refer to a label within the for statement; that is, a jump into a for statement body from outside is not allowed.

COMPOUND STATEMENTS

In Section 10 the reader was told that statements might be grouped together to form blocks or compound statements. We now write down the form to be taken by the compound statement:

```
L : L : ..... L : begin S; S; ..... S; S end
```

The sequence of one or more statements is surrounded by the statement brackets, begin and end. These two basic symbols enable the sequence of statements to be employed as one whole.* Labels may also be prefixed if required, but are not essential. Note also as a matter of punctuation that the final statement S in the sequence need not be followed by a semicolon; there is no following statement from which it must be separated.

Examples:

```
(1)  begin x := z + n × h;
            JO := JO + exp(-x↑2) × (if n = 0 then 0.5 else 1)
     end
(2)  begin i := 2; n := 1; h := h/2;
            goto R
     end
```

The compound statement may be used wherever a statement is allowed, in particular, after the do in a for statement. This extends the scope of the for statement, and enables a number of statements contained within the compound statement to be obeyed repeatedly. Thus Example (1) above forms part of a for statement in the specimen program of Appendix 1.

Problems

(1) Use a for statement to evaluate the product

$$(1 - \frac{1}{4})(1 + \frac{1}{9}) \;....\; (1 - \frac{(-1)^n}{n^2})$$

(2) Write a for statement to evaluate the function

$$y = \frac{1}{a^2 + b^2}\left[b^2 + \frac{a}{2}(1 - e^{-2a}) + \frac{2ab}{a^2 + b^2}\left\{e^{-a}(a \sin b + b \cos b) - \right.\right.$$

where $a = \frac{t}{T}(1 - s)$, $b = \frac{2s}{1 - s}$

for $s = 0.1\ (0.1)0.9$.

*The programmer will often find it helpful to connect the begin and end by a vertical line. This will tend to bring out the program structure but is not by any means essential. It also helps to ensure that an end corresponding to a begin is not omitted.

15. CONDITIONAL STATEMENTS 15.

Basic statements and compound statements (but not for statements*) are classified as unconditional statements (US). There is also a conditional statement and this takes one of the following forms:

$$L : L : \ldots\ldots L : \underbrace{\overbrace{\underline{\text{if}}\ BE\ \underline{\text{then}}}^{\text{if clause}}\ US}_{\text{if statement}}\ (\underline{\text{else}}\ S)$$

$$L : L : \ldots\ldots L : \overbrace{\underline{\text{if}}\ BE\ \underline{\text{then}}}^{\text{if clause}}\ FS$$

where FS stands for a for statement.

Examples:

if x = 0 then y := 1 else y := x - 1

if Boo then for I := 1 step 1 until m do n := n $\times$ (n - 1)

The if clauses and if statement are essential components of the forms of conditional statement in which they are marked. The broken parentheses around else S following the if statement in the first form indicate that this part may be omitted if there is no alternative statement to be executed. Thus an if statement alone can always be a conditional statement.

In the first form the conditional statement is executed as follows: if BE has the value true, then US is obeyed and the remainder of the conditional statement is ignored; otherwise, if BE has the value false, then US is ignored and S is obeyed. If the conditional statement is merely a labelled or unlabelled if statement, so that S does not exist and also BE happens to have the value false, then the statement produces no action beyond that caused by the evaluation of BE. In the second form, if BE has the value true then FS is obeyed; otherwise the statement again produces no action beyond that caused by the evaluation of BE.

Further Examples:

if abs(1 - 1/lambda)$>_{10}$-5 then goto Repeat

if v$>$u then X : q := n + m else goto R

Problems

(1) Find the final values of all variables when the following statements have been executed. The variables u and W are real and B boolean. (Appendix 2 is needed in solving this example).

*In accord with the Revised Report on ALGOL 60, paragraph 4.5.1.

```
         u := 3;
         B := true;
repeat:  W := u - 2;
         if u↑2 - 1/u>0 and W>-2 then u := 1/u
         else if B then goto Z
         else goto end;
Z:       B := false;
         u := W + 2 × u;
         goto repeat;
end:     B := u≥W
```

(2) Construct a loop to evaluate iteratively a root of the equation,

$$x^2 + x = 16,$$

using the formula

$$x = \frac{16}{x + 1}$$

and the starting value $x = 3.0$.

Use a conditional statement to exit from the loop when x has been determined to four decimal places. Afterwards, write a single for statement which will evaluate the root and follow its action for three iterations.

(3) Write a conditional statement which will cause a jump to four different points in a program, labelled P, Q, R and S. Make the jumps depend respectively upon two boolean variables, B1 and B2, by jumping to P, if both true, to Q if B1 true and B2 false, to R if B1 false and B2 true and to S if both false. When finished check that you have no if following a then; this will not be the case if the definitions of the present section have been followed. (Refer to Appendix 2 if necessary, in solving).

(4) Making use of a compound statement, write a for statement to evaluate,

$$y = \frac{x^n + 1}{x^n - 1}$$

for the first 100 integers n, the value of x being already known. Make provision to jump out of the loop whenever $x^n - 1$ is zero.

16· SUBSCRIPTED VARIABLES AND ARRAYS 16·

In computational work it is often necessary to perform the same operation on many different sets of data. When this is so, it is very convenient to be able to allocate a single name or identifier to groups of data and distinguish between individual items by means of subscripts. The notation using an identifier with subscripts, familiar in mathematics, is available in ALGOL, where it is known as the notation of subscripted variables.

In the particular form it takes in ALGOL we write the identifier common to the variables followed by square brackets enclosing the subscripts, for example,

ar [i, j]

As the subscripts are varied one obtains the various subscripted variables which are said to form the elements of an array. This array has the common identifier as its name; in the example above, this is ar.

The form taken by any subscripted variable may be depicted as follows:

A [SUB, SUB, ... SUB]

Here A stands for the array identifier. SUB stands for a subscript which may be any arithmetic expression. If this arithmetic expression (AE) is <u>real</u> then the largest integer not greater than AE + 0.5 is taken as the value of the subscript (i.e., the nearest integer). The subscripts are evaluated in the order of occurrence.

Examples:

ABC [1], f [i - 2], sigma [p + q, 4, p - 2]
Scorpion [k, l, m, n]

Subscripted variables may be of one of the three types <u>real</u>, <u>integer</u> and <u>boolean</u>, but variables corresponding to a single array identifier must be of one type only. In the same way as simple (non-subscripted) variables are admitted to the class of ALGOL variables, so are subscripted variables. It follows that the uses of subscripted variables to be described in the following sections are allowed.

16·1 Use of Subscripted Variables in Expressions

Expressions involving subscripted variables may be written, and at execution time the program will evaluate the expressions using the subscripted variables selected according to the current values of their subscripts.

Examples of arithmetic expressions using <u>real</u> and <u>integer</u> type subscripted variables as operands follow:

Mx [d1, d2]
a [j] × a [k + j] - b [3,2]

```
if A>B then X [i × j] else Y [i]
PQ [1, 2, if j<2 then r else s]
```

The following might be boolean expressions using subscripted variables:

```
Boolean [i, j, k]
WX [3] > YZ [rho]
```

16·2 Use of Subscripted Variables in Statements

The important use of subscripted variables as left part variables in assignment statements is allowed. In such a case the subscript expressions occurring in any left part variables are evaluated before the expression on the right of the assignment. When more than one left part variable is subscripted, subscript expressions are evaluated in the order of occurrence, i.e., from left to right.

Examples:

```
for i := 1 step 1 until n do
        E [n] := E [n] + A [i]×S [i]↑n

para 1 := arr 1 [i] :=,arr 2 [i,j] := 0
```

Finally, subscripted variables may also be used as controlled variables in for statements but special care should be taken when the values of the subscripts of the controlled variable are liable to be changed during the execution of the for statement. The actual mode of operation in such cases accords with the action of for list elements defined in Appendix 3.

Problems

(1) Calculate the final values of the variables involved in the following statements:

```
B [1, 2] := B [2, -1] := V [3] := 2;
i := 3 × B [2, -1] -2;
B [1, i ÷ 2] := i := 1;
V [B [1, 2] + B [2, -1]] := B [1, i] := 6 -V [3]
```

(2) Two one-dimensional arrays CAT and DOG each have 15 elements with subscripts commencing at 1. Write statements to evaluate SC the sum of the squares of the elements of CAT, and SCD the sum of the products of corresponding elements of CAT and DOG.

17· INPUT AND OUTPUT OF DATA 17·

The ALGOL language as so far defined provides no means for the initial setting of program parameters when these are to vary from job to job, nor does it provide means of printing out the results of program operation before they are lost. Thus, to produce ALGOL programs of practical value, it is necessary to add to our stock of statements and functions forms which provide input and output facilities.

The ALGOL Report indicates a means of providing these through ALGOL procedures which have bodies written in user-code. (Procedures are explained in detail in Section 19). In agreement with this indication English Electric-Leo has produced a KDF 9 ALGOL input/output scheme which the programmer may use. If, however, he knows the details of how the peripheral devices work on KDF 9 at the user-code level, there is nothing to prevent the user writing his own scheme, or making additions to the English Electric-Leo system. The compilers work independently of what input/output scheme is adopted since the text which defines the scheme must be written into the program and is processed just as any normal ALGOL text (see Section 19·6).

The scheme written by English Electric-Leo follows.

17·1 Device Numbers

Since there are a number of different input and output devices on KDF 9, a word is needed about the way the programmer calls for a particular device. The KDF 9 ALGOL statements and functions specified later in Section 17 allow him to write down a device number which will call a particular kind of device according to the ranges given below.

Input/Output Device	Device Number
Monitor typewriter*	00
Special input/output devices	01 - 07
Paper tape punch (8-channel)	10 - 17
Paper tape reader (5 or 8-channel)	20 - 27
Line printer	30 - 37
Card reader	40 - 47
Paper tape punch (5-channel)	50 - 57
Free	60 - 67
Line printer/Paper tape punch, common output	70 - 77
Magnetic tape	100 - 107 110 - 117

*The monitor typewriter should be used as little as possible, preferably not at all. This device will be used for purposes other than any the programmer may have.

Though the programmer has full control over the kind of device to be called by using a device number in the correct range, the computer (via a fixed control program called the director) in collaboration with the operator decides which actual device will be chosen for a particular device number.

17·2 Simple Forms for Reading and Writing Numbers

The form for reading a decimal number in characters is

read (DV)

where DV stands for a device number.

The identifier read is a function for use within arithmetic expressions. It is of type real and has as its value the next number on the input device, DV. Any device which is suitable for reading may be specified by the device number. Any arithmetic expression is valid as a device number, the nearest integer value being used to call the device.

The number to be read on the input medium must be an ALGOL number (see Section 4 for definition) and must be delimited by an ALGOL basic symbol which is not a digit, +, -, ., $_{10}$. A failure message will be printed and the program thrown off the machine if a number being input as data is non-ALGOL, out of range, or has an exponent out of range. The non-ALGOL symbol → (end message symbol, '?' for 5-channel working) must appear after the last delimiter of any data paper tape.

Examples:

```
j := read (21); x := (read (j) + 1)↑2
```

(The value assigned to J is being used as the device number in the second statement).

The simplest form for writing data to an output device is

output (DV, AE)

This is an ALGOL statement. It evaluates the arithmetic expression AE, which could of course be a variable, and outputs it to the device, DV. The number produced is in standard floating decimal with an 11 place signed mantissa in the range, $1 \leq N < 10$, followed by subscript $_{10}$ and a 2 digit signed integer exponent. Each number output by this statement is followed by a semicolon and a new line symbol, so that a print-out for more than one such number produces a single column.

Examples:

```
output (12, a-b);  output (10, k [1,p]).
```

17·3 Further Input and Output Statements

The following forms are available for the input and output of boolean data, and binary and decimal arrays:

read boolean (DV)

This function of type boolean takes the value of the next boolean value on the input device, either the symbol true or false.

write boolean (DV, BE)

This statement outputs the value of the boolean expression BE as either false or true followed respectively by one or two spaces.

read binary (DV, A, [AN])

This statement finds an array with the array name AN on the magnetic tape and reads it to the ALGOL array A. The array name must be enclosed by the string quote symbols [and] (square brackets underlined) as shown.

write binary (DV, A, [AN])

This statement stores the array A on magnetic tape in a form suitable for input by the statement read binary and gives it the array name AN.

read array (DV, A, [AN])

This statement reads from paper tape an array headed by the array name AN and certain other information. (See reference given at end of Section 17).

write array (DV, FE, DM, A, [AN])

This statement outputs the array A with preliminary information including the array name AN. The format expression FE specifies the layout of the elements of the array according to rules explained in Sections 17·4 and 17·5. DM stands for the number of dimensions of the array which must be specified. The elements of the array are listed so that the earlier subscripts change faster.

The use of a format expression in another most important statement allows a fine control over the output of simple decimal numbers. This statement is

write (DV, FE, AE)

which, like output (DV, AE) above, outputs the value of the arithmetic expression AE on the device DV. The format expression FE denotes an argument of type integer and provides the number of a layout which itself specifies the particular field and format required in output. The form and meaning of the layout and format expression are explained in the next two sections.

Examples:

```
write (12, format ([ -ddd.d]), x × (x↑2-1));
write (11, f1, a × b↑(k + 1))
```

17·4 The Layout

The layout provides a picture of the number which is to be printed. It shows where digits, zeros, spaces, sign, decimal point, and, in the case of a floating number, the exponent are to be printed in the output field. It may also call for a new line or new page on the output medium or print a semicolon to separate one number from the next. This makes the write statement very versatile.

We shall now proceed through the various facilities in more detail, showing how the layout is constructed.

(1) Digits Wherever a digit is required in the output field we put a letter d in the corresponding position in the layout. The letter n may be used in the first digit position in which case if the number is too small to fill the digit layout, zeros on the left are suppressed. Zeros in the units position and to the right of the decimal point are never suppressed.

(2) Decimal Point The decimal point is inserted in the appropriate position, when required.

(3) Sign The sign + inserted before all d's and the decimal point will ensure that either + or - appears in the result as appropriate. When n appears in the layout and zeros on the left are suppressed. the sign is moved to the right.

The sign - inserted in the layout has the same effect as + except that a space is inserted instead of + for positive numbers.

The symbol ≠ causes a sign (either + or -) to be printed but always in the position specified.

Finally, if there is no sign in the layout, no sign is printed.

(4) Spaces The letter s inserted in the layout causes a blank space to be printed in the corresponding position. A maximum of 15 spaces are allowed in front of the sign, and these initial spaces may be abbreviated by inserting a single s preceded by the number required. They are still available when no sign is present in which case up to the first 15 spaces are counted as initial spaces.

(5) Zeros Zeros may be inserted at the end of a decimal layout (one having no exponent). These allow the printing field to float keeping the number of significant figures specified by n and d.

(6) Exponent A floating point number in ALGOL form will be output if the layout includes a mantissa and an exponent. The mantissa should be of the form "d." followed by a fractional part containing only d's and s's. The exponent immediately following the mantissa should be of the form "$_{10}$", followed by a sign, followed by "nd". The sign used in the exponent may take any of the three forms mentioned above and has a similar effect. Any symbol in the layout following the exponent must be a terminator.

(7) Terminators When required the layout may be concluded by one of the following symbols which have the effect specified.

;	A semicolon is output in the position specified.
c	A carriage return line feed is output.
p	A page change is output on a line printer and page shift on a paper tape punch.

The following nine combinations of terminators are allowed:

; c p cc ccc ;p ;c ;cc ;ccc

Examples of layouts:

```
sndd.ddds000s
7s≠nddd
-d.dddd₁₀+nd;
sss+dd.d;ccc
```

17·5 The Format Expression

The format expression provides a means of calling a particular layout. Using a function called format it is possible to associate an integer with the layout. Thus, using LAY to stand for a layout,

format ([LAY])

will provide the integer corresponding to LAY.
This integer may be used as a format expression parameter in the statement,

write (DV, FE, AE).

The string quote symbols [and] which enclose the layout are essential.

Example:

write (30, format ([+sddd.ddds;c]), A[j + 4]).

Whenever the same layout is to be used to output more than one number it is advantageous to assign the integer value produced by the function format to an integer variable, and use the variable as the format expression.

Example:

```
F := format ([ndddc]);
for i := 1 step 1 until n do write (30, F, List [i])
```

17·6 Input and Output of Text

There are two statements dealing with the input and output of text.

write text (DV, ST)

This statement outputs the text written as ALGOL basic symbols in the string ST. (For explanation of strings see Appendix 6). The string contains the text for output enclosed by string quotes [and]. Editing symbols c, p and s preceded if desired by an integer and enclosed by additional quotes may be inserted in the string to produce the effect of carriage return line feed, page change, and space, respectively. The integer before one of these letters specifies the number of such symbols to be output. Alternatively for space, one or more asterisks may be inserted in the text without additional quotes.

Examples:

```
write text (12,[[p] Result [c7s] x*=**])
write text (13,[[5c4s]])
```

copy text (DV, DV, ST)

This statement copies ALGOL basic symbols from the input device specified by the first parameter, to the output device defined by the second parameter. The third parameter consists of either one or two basic symbols in string quotes: for one basic symbol copying continues from the actual starting position of input to the first occurrence of this symbol; for two symbols, copying starts immediately after the occurrence of the first symbol and ceases on occurrence of the second. The basic symbols inserted in the third parameter are not themselves copied.

Example:

```
copy text (20, 12, [;;])
```

17•7 Initialisation and Closure of Devices

Certain statements are required in order to allocate and de-allocate actual devices corresponding to those called by the ALGOL programmer and inform the operator of the choice for purposes of loading and unloading tape reels, etc. These statements are as follows:

open (DV)

This statement must precede the first use of a device DV and automatically produces on that device the effect of carriage return line feed followed by case normal. It applies to all devices save the monitor typewriter and magnetic tape decks. The monitor typewriter requires no initialisation by the ALGOL program, while magnetic tapes are initialised by the find statement to be described next.

Example:

```
open (22)
```

find (DV, ST)

This statement will look at all tapes loaded and find the tape with the label referred to in the string ST. The corresponding deck is allocated as the device with number DV. The string may enclose in quotes ⌊ and ⌋ either the number of a device from which a tape label may be read within string quotes, or the tape label itself.

Examples:

```
find (103, ⌊KDSGR562⌋)
find (100, ⌊21⌋)
```

When a device has been initialised the operator is informed of the device number allocated, and a standard format is automatically output, including program identification and blanks and leaving the device at the beginning of a line ready for the programmer's output.

close (DV)

This statement closes device DV and should be applied to all initialised devices before the program ends. After closure a device must be re-initialised by an open or find statement before use again. Closing a device early helps to reduce buffer storage. The close statement has no application to the monitor typewriter.

17•8 Manipulation of Magnetic Tapes

Various statements for the manipulation of magnetic tapes are available or under consideration. Amongst these there is

interchange (DV)

This statement is used to change a magnetic tape deck from a reading to a writing mode, and vice versa. See Algol Users Manual for further details.

skip (DV, N)

This statement skips N binary arrays on the magnetic tape corresponding to device number DV.

17.9 Restrictions

Apart from the restriction limiting initial spaces in a layout to 15 as already mentioned, a total of 23 n's, d's, zeros and s's are allowed from the close of initial spaces to a subscript $_{10}$ (or the close of the layout if there is no exponent). The layout should not allow more than 12 significant digits (n and d) in output.

Up to 120 positions maximum will be available per printing line in output on the paper tape punch, line printer and magnetic tape. For cards the full field of 80 characters will probably be available when using the punch.

If a number does not fit the layout an alarm printing occurs. This means that it appears on a fresh line to the standard layout;

$\neq$d.ddddsddddsdddds$_{10}\neq$nd;c

Each alarm printing will be preceded by an asterisk and as the layout shows will end with a semicolon. An alarm printing will not disturb the overall layout.

Details of other facilities in this input/output scheme can be found in the ALGOL Users Manual.

Problems

(1) Write a for statement which reads ten numbers from paper tape and sums their squares.

(2) Assign a format expression to the integer variable f and use it in a for statement which reads 100 numbers and outputs their cubes to the format ddd.dd in a single column on a paper tape punch.

(3) Write an output statement which will print on a line printer the heading -

Co-ordinates of the Parabola, y↑2 = 4x.

(4) Write statements to produce on a line printer the positive co-ordinates of the parabola $y^2 = 4x$ in two parallel columns with headings x and y. The abscissa x should take the values 0(0.01)5.

(5) Write a piece of program to read N integers (each less than 10^6) from reader 20, and print them out in a column with their prime factors in a parallel column on punch 10. Restrict the search to the factors two and three only. When both occur, list them with a semicolon as separator (thus, 2;3). Allow three spaces to separate the two columns.

(6) Write statements which will output the diagonal elements of a 22 × 22 array called BRUTE, elements commencing BRUTE [1,1], on the paper tape punch 11. Make the output appear five elements per line in columns each separated by five spaces. Use a floating decimal point allowing eight significant digits in the mantissa.

DECLARATIONS, BLOCKS AND PROGRAMS

Having considered most of the forms of statement allowed in an ALGOL program, we shall now consider how the statements ought to be arranged and what means exist for cementing them together to form a complete program. This should put the reader in a position to write simple programs.

18.1 Declaration of Simple Variables

We come first to the idea of the declaration. This may have been entirely new to the reader when he read of it in Section 2. The declaration is the programmer's means of conveying information to the ALGOL translator about the kind of quantities represented by the identifiers used in the program. This makes it possible for the translator to treat each identifier in the way most appropriate to its kind in allocating storage and using arithmetic routines.

The rules associated with the declaration are as follows:

(1) All identifiers having an operational significance except those representing labels and standard functions must be declared.*

(2) For simple variables the declaration consists of a basic symbol denoting the type of the variable (<u>real</u>, <u>integer</u> or <u>boolean</u>), followed by a list of simple variables separated by commas. A declaration for such variables will look like

T V, V, ... V

where T stands for a type symbol and V for a variable. Each variable listed after a given type symbol is of that type. For variables of a different type, different declarations must be made using the appropriate type symbols. The order chosen in writing more than one type declaration or in listing variables is immaterial.

(3) A declaration must be placed at the head of the block to which it is intended to apply. (Blocks are considered in Section 18.3).

As already mentioned in discussing types of numbers in Section 4 type <u>real</u> denotes real quantities which are to be treated in floating point. Type <u>integer</u> quantities are treated using fixed point integer arithmetic.

In addition to the type symbol appearing at the head of a simple variable declaration the symbol <u>own</u> may appear. The meaning and use of this symbol is explained in Appendix 4.

*This rule includes the identifiers used for input/output statements and functions which must all have procedure declarations. For the purposes of the present chapter and the problems appearing at the end, these declarations are omitted. They will in practice be obtained automatically from the ALGOL procedure library in accordance with the rules explained in Section 19.7.

One or two other basic symbols are used in declarations in order to specify such things as arrays, switches, and procedures. The declaration of ALGOL procedures will be considered when we come to discuss procedures in Section 19. Switch declarations are discussed in Section 21, while array declarations are considered in the following section.

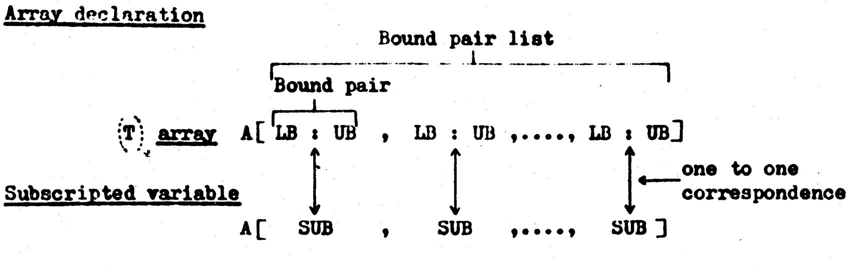
Examples of Simple Variable Declarations:

integer	i, j
real	alpha, P, stress, radius
boolean	STABLE, B1, B2, B3

18·2 Array Declarations

Just as a simple variable must be defined before its use, so before using a subscripted variable the array to which it belongs must be defined. This is done by means of an array declaration which appears along with other declarations in the head of an appropriate block. The object of the array declaration, besides noting the existence of the array, indicates the type of its elements, whether real, integer or boolean. It also limits its size by noting upper and lower bounds on the subscripts.

The form of an array declaration and its correspondence to a subscripted variable may be shown as follows:


Array declaration
Bound pair list
Bound pair
(T) array A[LB : UB , LB : UB ,...., LB : UB]
one to one correspondence
Subscripted variable
A[SUB , SUB ,...., SUB]

T stands for type and may be real, integer or boolean. If no type symbol appears in the declaration then the elements of the array are understood to be of type real. array is a declarator in the list of basic symbols.

A stands for the array identifier.

LB stands for lower bound and UB for upper bound.

SUB stands for subscript.

By means of bracketing the diagram also defines the meaning of a bound pair and a bound pair list. The following notes on the bound pair list are important.

(1) The bounds may be arithmetic expressions and are evaluated in the same way as subscript expressions. Thus if the arithmetic expression (AE) is real then the value of the subscript is taken as entier (AE + 0.5). (See Section 9 for the standard function entier).

(2) The bound pairs give the bounds of corresponding subscripts and a subscripted variable is only defined if its subscripts lie within these bounds. An array is not defined if a lower bound is greater in value than its upper bound.

(3) The order of evaluation of the bound pairs is from left to right.

(4) The bound pairs are evaluated every time control reaches the array declaration and only the current values of these bounds are valid in considering the legitimacy of some subscripted variable (See Note 2).

(5) A bound pair expression may only depend on variables and functions which are declared in a block enclosing the block for which the array declaration is valid. Note (5) should be understood more clearly after reading Section 18·3. A more thorough explanation of the point will be found in Section 22·2.

Examples of array declarations:

```
array AB[1:10, 1:10, 1:3]
real array M1[ p+n : q, p+m : r]
integer array ARRAY [1 : if i = 0 then n else 2 × n]
```

The form of the array declaration defined above is not as general as it could be. In fact it may be extended so that any number of array declarations may be strung together and repetition of similar information eliminated. Thus, after the symbol array a list of identifiers may appear, followed by a bound pair list enclosed in the usual square brackets. Each of the identifiers is thereby declared as representing an array of the same type as all the others in the list and having the same number of dimensions and the same upper and lower bounds on its subscripts. Other arrays of the same type but having different dimensions and/or bounds may be declared by adding them on to the above declaration and following them with the new bound pair list.

Apart from the appearance of own for which see Appendix 4, the general form for an array declaration may be depicted as follows:

```
(T) array A, A, ... A[BPL], A, A,...A[BPL], ...A, A,...A[BPL]
```

BPL stands for bound pair list.

Further examples of array declarations:

```
array A, B, C[1:1, 1:m, 1:n]
boolean array b1[0:20], b2, b3[-10:n↑2, -5:n × m -1]
```

18•3 Blocks

A block of ALGOL program is constructed in the same form as a compound statement but with the essential addition of at least one declaration. The form of a block may be represented thus:

$$L : L : \ldots\ldots L : \overbrace{\underline{\text{begin}}\ D\ ;\ D\ ;\ \ldots\ D}^{\text{Block head}}\ ;\ S\ ;\ S\ ;\ \ldots . S\ \underline{\text{end}}$$

where L stands for a label, D for a declaration and S for a statement. Note that the declarations are all followed by a semicolon. A declaration appearing in the block head only applies for the corresponding block. A block need not be labelled.

An important use of blocks is provided by the following complete definition of the unconditional statement.

$$\text{Unconditional statements include} \begin{cases} \text{basic statements} \\ \text{compound statements} \\ \text{blocks} \end{cases}$$

The inclusion of blocks as unconditional statements means that blocks may be used within for statements (Section 13), compound statements (Section 14), conditional statements (Section 15) and also within other blocks.

Examples:

(1) The two simple programs given in Section 2 are both blocks. We reproduce one of them here in an abbreviated form.

```
begin  real x, y, z;
  |    open (20);
  |    x:= read (20);
  |    ....
  |    ....
end
```

There is one declaration in the head of this block, namely

real x, y, z

and the reader will notice that it is correctly followed by a semicolon.

(2) Insert appropriate declarations and begin and end brackets to make the statements of the problem in Section 12•2 into a block:

```
begin   real p, q, SUM;
        integer n;
        n := 1;
        p := 0.5;
        SUM := 0;
        q := 1;
loop :  SUM := SUM + q/n;
        q := q × p;
        n := n + 1;
        goto loop
end
```

Problems

Insert appropriate declarations and begin and end brackets to make the statements of the following problems appearing earlier in this manual into blocks.

(1) Prob. (1) Section 11.

(2) Prob. (3) Section 11.

18·4 Definition of a Program

A program is officially defined as a block or compound statement which is not contained within another statement and which makes no use of other statements not contained within it. Normally, a program will have identifiers declared at its beginning, so that it will be most naturally constructed as a block.

The reader will appreciate that the uses of blocks mentioned in the previous section allow a complicated program structure involving blocks within blocks. Since there are certain rather complex rules associated with the use of identifiers when inner blocks exist, it will be advisable for the beginner to restrict his programs to those containing one block with all declarations made in its block head. Section 22 will explain the restrictions on the use of identifiers when a complex block structure is used, and the value in some circumstances of such a structure.

Example:

Write a program to solve the equations $ax + by = c$, $Ax + By = C$ for n sets of coefficients a, b, c, A, B and C with a provision for failure when $aB - bA$ is small or zero.

(5) Write a program to group n integers between 0 and 99 into classes 0 - 9, 10 - 19, ... 90 - 99, and print out the number of integers in each class.

(6) Write a program to find the first n positive roots of

$$x \tan x = a$$

with an error less than e. Use the iterative relation

$$x_{r+1} = \tan^{-1} (a/x_r)$$

to improve an approximate root x_r. For the first root take 0.5π is an initial guess. For the second root take π plus the first root and so on.

(7) Write a program to find the area of a triangle given the length of the sides as data. Use [dd.dddd] as the layout of digits for output.

$\triangle = \sqrt{s(s - a)(s - b)(s - c)}$; $s = \frac{1}{2}(a + b + c)$, where a, b, and c are the lengths of the sides.

(8) Tabulate the binomial coefficients ${}^nC_r$, r = 0(1)n, for given n.

$$\left[\text{Note } {}^nC_r = \frac{n(n-1)(n-2)\ \dots\ (n-r+1)}{1.\ 2.\ 3\ \dots .\ r} \right]$$

19·1 The Purpose and Application of Procedures

ALGOL provides a facility similar to that of the subroutine in a machine-coded program. This facility is known as the procedure. It enables the programmer to use a single piece of program in a number of places in his program or even in different programs without having to rewrite it on each occasion to suit new parameters. To each procedure there is attached an identifier and each occurrence of this identifier within some ALGOL statement initiates a call of the procedure concerned.

As an example the programmer may decide to have a procedure which provides the tangent of an angle. He associates with this procedure the identifier TAN, and on any occasion when he desires the tangent to be evaluated, he merely writes down the procedure identifier together with the angle in which he is interested. Suppose the angle were x, then writing

TAN(x)

would give him the tangent of the angle x. He might decide to use this in an arithmetic expression such as

$$(2 \times \mathrm{TAN}(x) + 1)/(\mathrm{TAN}(x) - 1)$$

The use of a procedure like TAN obviously implies some means of defining its action. The means available is the procedure declaration. This declaration contains a body in which the operation of the procedure is defined (usually by means of ALGOL statements). It also has a heading in which the procedure identifier may be associated with a set of parameters, known as formal parameters. When the procedure is to be called, the programmer inserts the actual parameters upon which the procedure is required to operate instead of the formal parameters to which they correspond. At the time of executing this call the procedure body is entered and obeyed.

In different calls, different sets of actual parameters may be used. In the example mentioned above the procedure TAN might be declared as TAN(z), where z is a formal parameter. The call of this procedure mentioned earlier uses x as an actual parameter to correspond to z, but on some other occasion a different actual parameter might be used and the programmer might call for TAN(y), or even TAN(x × y).

There are two ways of using procedures. TAN(z) above is being used to provide function designators TAN(x) and TAN(x × y) which supply values through the procedure identifier for use in the expressions in which they occur. It is also possible to call procedures by procedure statements which are used in a manner similar to other ALGOL statements. In this case, though information may be supplied to the program via the parameters, a value is not supplied through the procedure identifier.

19·2 Procedure Declarations and Corresponding Calls

As already suggested, the declaration of an ALGOL procedure specifies its action. This declaration is inserted in the block head to which it applies in the same way as any other kind of declaration. Figure 2 shows the form taken by a procedure declaration, and also included is the procedure statement or function designator to show the correspondence between the declaration and the call.

In this diagram P stands for a procedure identifier, AP for an actual parameter and FP for a formal parameter. T stands for a type symbol placed before the ALGOL symbol, procedure, the broken parentheses indicating that it is not always required. When the procedure defines the value of a function designator, the type symbol must be included in the declaration and specifies the type of this value.

In the procedure declaration the basic symbol procedure is followed by the procedure heading which includes the procedure identifier, formal parameter part, value part and specification part. The procedure heading is followed by a procedure body.

The formal parameter part consists of a formal parameter list enclosed in parentheses. The formal parameter list in turn consists of one or more formal parameters separated by parameter delimiters. The actual parameter part, actual parameter list and actual parameters in the procedure call are analogous. In both cases the parameter delimiters may be commas as in Figure 2 or they may take the form:

) LS : (

where LS stands for a letter string. This enables the programmer to include an indication of the meaning of parameters in the formal parameter list.

A formal parameter is simply an ALGOL identifier, while an actual parameter which corresponds to it might be an arithmetic, boolean or designational expression, or an array, procedure or switch identifier or a string (see Section 19·5). The correspondence of formal and actual parameters shown in Figure 2 means that there must be the same number of actual parameters in a call as formal parameters in the declaration. Taking the parameters in order there must also be a compatibility in kind and type. The specification part of the procedure declaration mentioned in Figure 2 defines the kind and type of formal parameters and is described in Section 19·5.

Information concerning the way actual parameters are to be treated is provided by the value part of the declaration. This is described in Section 19·4.

The value and specification parts complete the procedure heading in the declaration and are followed by a procedure body. This commonly contains a number of ALGOL statements within which the formal parameters appear, being used as variable identifiers, array

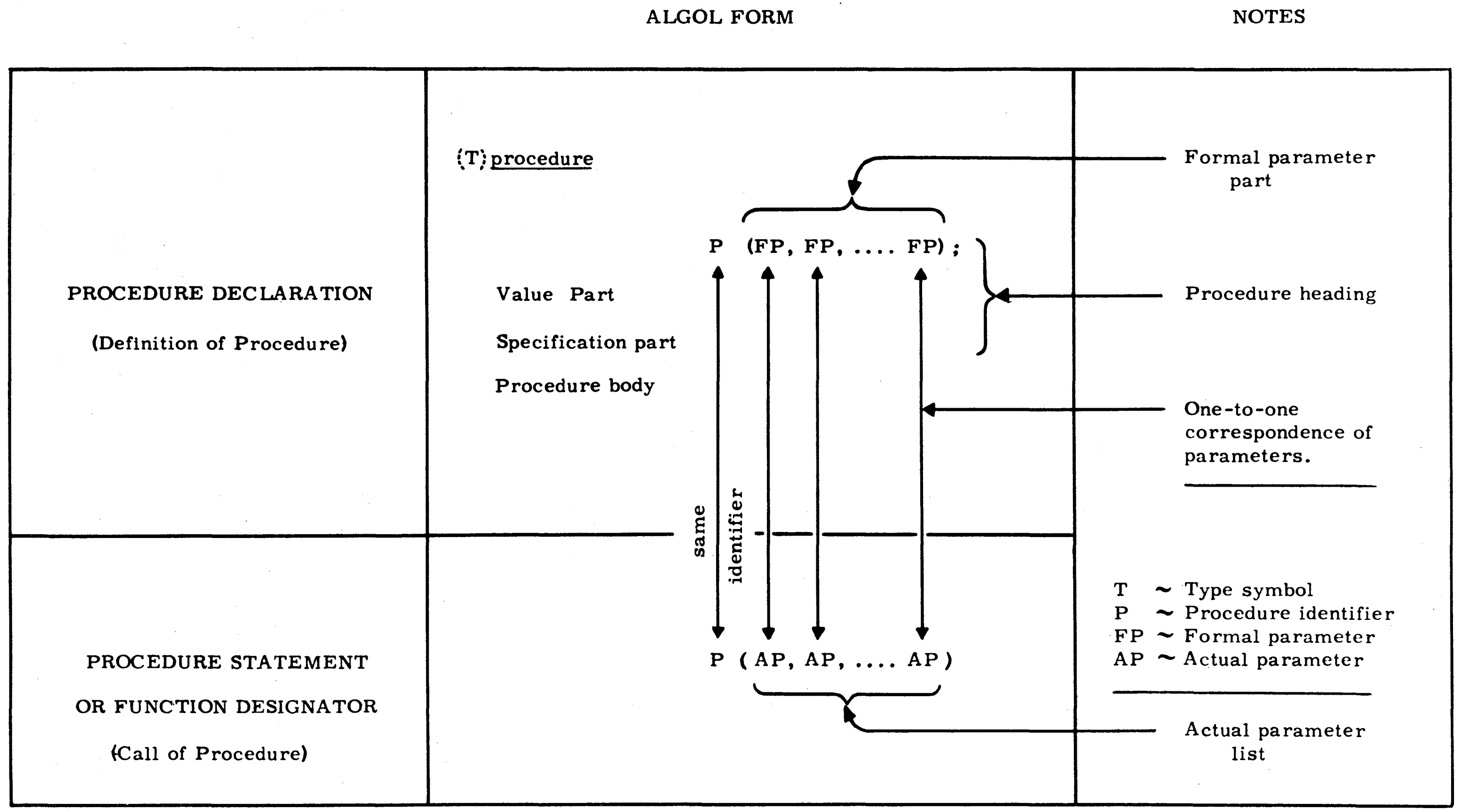

ALGOL FORM
NOTES
PROCEDURE DECLARATION
(Definition of Procedure)
{T} procedure
Value Part
Specification part
Procedure body
P (FP, FP, ... FP) ;
same identifier
P (AP, AP, ... AP)
PROCEDURE STATEMENT
OR FUNCTION DESIGNATOR
(Call of Procedure)
Formal parameter part
Procedure heading
One-to-one correspondence of parameters.
T ~ Type symbol
P ~ Procedure identifier
FP ~ Formal parameter
AP ~ Actual parameter
Actual parameter list

FIGURE 2

identifiers, etc. At each execution of the procedure certain changes associated with the parameters are made and these statements are then obeyed.

Here is an example of a program which contains a procedure declaration and a call of that procedure by a procedure statement.

```
begin  real a, b, D;
       procedure  EXAMPLE (x, y) Result: (R);
       value x, y;
       real x, y, R;                                   Procedure
           begin                                       Declaration
                    x := x + y;
                    R := x↑2 + y↑2
           end ;
       a := 1;
       b := 2;
       EXAMPLE (a, a+b, D);                             Call
       write (30, format ([d.dddd10+nd]), D)
end
```

The procedure declaration of EXAMPLE uses the three formal parameters x, y and R. The procedure statement supplies the actual parameters a, a+b and D to correspond. In this example the body of the procedure consists of a compound statement containing two assignment statements.

19.3 Declaration of Procedures Defining a Function Designator

For a procedure to be used as a function designator, the procedure body appearing in the declaration must contain one or more statements which assign a value to the procedure identifier. At program run time at least one of these assignments must be executed per call of the procedure. The value held by the procedure identifier on exit from the procedure body is used as the value of the function designator in evaluating the expression in which this function designator occurs. A procedure which is to be used as a function designator must always have a type symbol (T in Figure 2) appearing at the commencement of the procedure declaration as already mentioned. This declares the type of the values taken by the function designator, whether real, integer or boolean.

The following is a declaration of a procedure for use as a function designator:

```
real procedure  TAN(z);
     value  z;  real  z;
     if abs(z)>1.570796326 then goto Failure
     else  TAN := sin(z)/cos(z)
```

In this procedure abs, sin and cos are the identifiers of standard functions; while Failure is a label in the main program.

19.4 The Value Part

The value part of the procedure heading, which immediately follows the semicolon after the formal parameter part takes the following form:

value FP, FP,........FP;

A formal parameter appearing in the formal parameter list may or may not appear in the list following the basic symbol value. If it does appear then the formal parameter is said to be called by value otherwise it is said to be called by name. If no parameters are to be called by value, then the value part will be empty.

The difference between the call by value and call by name lies in the different ways the parameters are treated on entry to and execution of the procedure at the time control reaches the procedure call. The beginner sometimes finds considerable difficulty in understanding these different treatments. To help him it is suggested that he take some examples and carefully work through them executing exactly the operations specified in the next two paragraphs. One example is given in the text following these two paragraphs and the reader may construct others for himself.

Call by Value

When formal parameters are called by value, the corresponding actual parameters are evaluated and assigned as initial values to the formal parameters before entry to the procedure body occurs. After entry to the procedure body operations are performed upon the formal parameters as specified by the ALGOL text.

Call by Name

Before entry to the procedure body formal parameters called by name are replaced in the text of the body by the corresponding actual parameters. Parentheses are placed around these actual parameters whenever possible. After entry to the procedure body operations are performed upon the actual parameters using the revised text. This may occasion the evaluation of the actual parameters at any time during the execution of the procedure body.*

That there is a real difference between calling by value and name can be illustrated by the program at the end of Section 19.2. In this program the procedure EXAMPLE has two of its formal parameters appearing in the value part, thus:

value x, y;

The procedure is called once by the statement EXAMPLE (a, a+b, D) with the variables a and b currently holding the values 1 and 2 respectively. We work out below the effect of the procedure call (a) when the procedure declaration is altered by making the value part blank, so that all formal parameters are called by name, and (b) when the value part stands as in the program above, so that some parameters are called by value.

(a) All formal parameters called by name

Formal parameters called by name are replaced by the actual parameters in the text. The two statements in the procedure body therefore become

```
a := (a) + (a + b);
D := (a)↑2 + (a + b)↑2
```

The procedure body is now executed with the following effect upon the variables.

	a	b	D	x	y	R
Initially	1	2	-	-	-	-
On entering procedure body	1	2	-	-	-	-
After first statement	4	2	-	-	-	-
After second statement	4	2	52	-	-	-

(b) Formal parameters x and y called by value, R called by name

In this case, the second statement only is modified and that because R is called by name. We have,

```
x := x + y;
D := x↑2 + y↑2
```

*When a duplicate use of identifiers has been made, such as will be described in Section 22, confusion between identifiers inserted via formal parameters called by name and the same identifiers already occurring within the procedure with other meanings is automatically avoided by a systematic change of identifiers involved.

Before entry to the procedure body the actual parameters a and (a + b) are evaluated and their values assigned to x and y. The procedure body is now executed with the following effect upon the variables.

	a	b	D	x	y	R
Initially	1	2	-	-	-	-
On entering procedure body	1	2	-	1	3	-
After first statement	1	2	-	4	3	-
After second statement	1	2	25	4	3	-

From the above example it will be seen that, when called by name, formal parameters are dummies, i.e., no values are actually assigned to them. In fact they are truly formal. When called by value they serve the purpose of working variables local to the procedure, while the variables used in the corresponding actual parameters remain unchanged, (unless the procedure body as appearing in the declaration explicitly assigns to them).

Sometimes the kind of call of a formal parameter, whether by value or name, has no effect upon the final result of a procedure. Thus, as the reader may check for himself in the example quoted above, when we have x called by value and R by name, whether y is called by value or name makes no difference to the final values of the program variables a, b and D. For the sake of efficiency of translation it is recommended that call by value be specified whenever possible.

Note that R cannot be called by value in the above example because an assignment of the value of D to R would be involved on entering the procedure and at this time D has no value. Note also that the insertion of parentheses around actual parameters called by name can be essential. Thus a different result would be obtained in (a) above, if the a + b were not so enclosed in the statement assigning to D.

19.5 The Specification Part

The specification part of the procedure heading is very like the declaration list which occurs in the head of a block. It gives information about the kinds and types of the formal parameters used in the procedure. In KDF 9 ALGOL all formal parameters must be included in the specification part with full specification. (This is not essential in ALGOL 60). The specification part appears in the form:

```
Specifier FP, .... FP;
Specifier FP, .... FP;
    .............
Specifier FP, .... FP;
```

where a specifier may be any of those listed in the following table (T stands for a type symbol).

Specifier	Corresponding Actual Parameter
T	arithmetic or boolean expression
(T) array	array identifier
(T) procedure	procedure identifier
label	designational expression
switch	switch identifier
string	string

Like the value part, the specification part closes with a semicolon.

The use of parameters, specified merely by a type symbol should be clear; the use of parameters specified by most of the other symbols in the above list is explained in some detail in Section 23 on the Advanced Use of Procedures; while an explanation of the general use of strings and the symbol string appears in Appendix 6.

In addition ALGOL only allows an actual parameter to be of a kind and type which is 'compatible' with those of the corresponding formal parameter. For example, a formal parameter which occurs as a left part variable in an assignment statement and is called by name can only correspond to an actual parameter which is a variable.* It is recommended as good programming practice that the types of formal parameters called by name and the corresponding actual parameters be not merely 'compatible' but the same.

An example of a specification part was provided by the declaration of the procedure EXAMPLE in Section 19·3. Thus:

```
real  x, y, R;
```

The specification part for another procedure might be:

```
integer i, j;
real  X, Y; real array K1;
integer procedure IT;
procedure P, Q;
```

19·6 The Procedure Body

The procedure body which follows the procedure heading may be any ALGOL statement. In practice it is usually also an ALGOL block. Of course, this definition allows the procedure body to include whole pieces of program containing many statements.

*KDF 9 ALGOL has a further restriction of a similar kind, for which see Restriction (3) on page 1 in the Introduction.

Means are also available for the programmer to write the procedure body in KDF 9 machine code, if he so wishes, and the ALGOL Users Manual (Section 5) explains how to do this.

It was explained in Section 19·4 that the procedure identifier could occur within its own body on the left of an assignment and that such procedures could be used to define function designators. It is also possible for the procedure identifier to be used in some other way within its procedure body, such as in an expression. If this is so, its occurrence signifies a new call of the procedure. Though such applications receive some discussion in Section 23·5, they are to be avoided by the inexperienced ALGOL programmer.

19·7 The ALGOL Procedure Library

The ability to use a procedure on a number of different occasions makes this ALGOL construction extremely useful. There are some procedures such as those for input and output, which are of such general application and required so often that it has been decided to include them in an ALGOL procedure library. This library saves the programmer having to write out the declarations of these common procedures each time he needs them. It includes pieces of ALGOL text as well as single procedures and is stored on magnetic tape.

Wherever an insertion in a program is required the symbol library must appear followed by a list of unsigned integers specifying the particular portions of ALGOL text to be inserted, see for example the program in Appendix 1, of this manual, and also Section 4 of the ALGOL Users Manual.

Contributions to the library will be published from time to time after thorough test.

Problems

(1) Trace the various parts of the procedure declaration for erfc given in the specimen program of Appendix 1. Ignore the commentary which will be explained in Section 20.

(2) What is the value of the function designator,

AP(1, 3, 5)

if its procedure declaration is as follows?

```
real procedure AP(a, d, n); value a, d, n;
     real a, d; integer n;
     begin  integer i;  real t;
            t := 0;
            for i := 1 step 1 until n do
                 begin
                        t := t + a;
                        a := a + d
                 end    ;
            AP := t
     end
```

(3) Assuming the above procedure declaration but with a and d omitted from the value list, what would be the final values of the variables on exit from the last of the following statements? The variables p, q, r, and s are real.

```
......
p := -1;  q := 2;  r := 3;
s := AP(q, - (p + q↑r) × p, 2);
......
```

(4) (i) Write a procedure declaration which defines a function designator Jo(x) based upon the series

$$J_0(x) = 1 - \frac{x^2}{2^2} + \frac{x^4}{2^2.4^2} - \frac{x^6}{2^2.4^2.6^2} + \ldots.$$

(ii) Use this procedure declaration in a program which evaluates the expression

$$J_0(x) + J_0\left\{J_0(x^2)\right\}$$

for x = 0(0.1)1.

20. COMMENT CONVENTIONS 20.

If the reader has studied the specimen program of Appendix 1, he will have noticed the occurrence of commentary. This is a convenient feature of ALGOL. By its means the programmer is able to record information which may help other users of his program or even himself at some later date when his memory of the program is dimmed.

The rule governing the use of comment is:-

Any sequence of basic symbols is ignored while the program is being executed and may be used as commentary if it belongs to one of the following categories.

(i) Sequences following ";" or "begin" which commence with "comment" and are closed by ";", but do not contain ";".

(ii) Sequences following "end" and not containing "end", ";" or "else".

Example:

```
begin   comment  A program to illustrate the use of comments;
        integer  i ; real  x;
        comment  start program begin for i := 1 step 1 until
                 10 do calculation end;
        for i := 1 step 1 until 10 do

            begin  if i = 5

                   then begin   x := i × (i + 1);
                                x := x↑x
                        end     the Special Case

                   else begin   x := i × (i - 1);
                                x := x↑(x + 1)
                        end     if i ≠ 5 then Normal Case

            end

end         program
```

When all commentary has been eradicated this example appears as follows:

```
begin   integer  i ; real  x;
        for i := 1 step 1 until 10 do
             begin  if i = 5
                    then begin   x := i × (i + 1);
                                 x := x↑x
                         end
                    else begin   x := i × (i - 1);
                                 x := x↑(x + 1)
                         end
             end
end
```

21. SWITCHES AND DESIGNATIONAL EXPRESSIONS 21.

Sometimes a programmer may wish to jump from a point in his program to one of a number of others depending on the value of some variable or expression. In common programming parlance he wishes to use a multiple branch or switch. Although it is possible to do this in ALGOL by means of conditional statements, they become long-winded for a many-branched switch. A much simpler facility is provided. The programmer writes down what is called a switch designator in place of a label in his goto statement. This looks precisely like a one-dimensional subscripted variable.

Example: goto EXIT [n]

The declaration of the switch identifier, however, looks very different from that of an array. It will usually contain a list of labels, and depending on the value of the subscript of the switch designator one of these labels is chosen for action in the goto statement. Before defining the form of a switch declaration precisely let us generalise the use of labels and switch designators in the goto statement.

A goto statement may take the form:

goto DE

where DE stands for a designational expression.

A designational expression is a rule for finding a label. In a manner analogous to the arithmetic expression it is defined as being either

a simple designational expression (SDE)

or of the form

if BE then SDE else DE

A simple designational expression is one of the following:

a label,

a switch designator,

or a designational expression enclosed in parentheses.

Further Examples of Goto Statements using Designational Expressions:

goto if $a<b$ then FAIL else CONTINUE

goto S[3 + (if $x>0$ then p else q)]

We may now define the forms allowed for a switch declaration and its corresponding switch designator. These are shown below.

Switch Declaration

$$\begin{tikzcd}
\underline{\text{switch}}\ SW := \overbrace{DE,\ DE,\ \ldots.,\ DE}^{\text{Switch List}} \arrow[d, leftrightarrow] \\
SW\,[SUB]
\end{tikzcd}$$

Switch Designator

where SW stands for a switch identifier.

The following is a possible declaration corresponding to the switch designator EXIT [n]

switch EXIT := L1, L2, FAIL [m]

The evaluation of the switch designator EXIT [n] at run time would proceed as follows. Depending on whether the value of n is 1, 2 or 3, the appropriate element of the switch list is selected. Thus the value of EXIT [1] is the label L1; for EXIT [2] it is the label L2. EXIT [3] leads to the switch designator FAIL [m] which in turn must be evaluated by referring to the switch list in the declaration of the switch identifier FAIL.

In the general case the evaluation of a designational expression at run time to produce a label proceeds as follows:

(1) The boolean expressions select a simple designational expression. If this is a label we have the result desired.

(2) If it is a switch designator, the numerical value of its subscript expression is calculated to the nearest integer in the same way as an array subscript. The result is used to select a designational expression from the corresponding switch declaration switch list counting these 1, 2, 3 etc., from left to right. (If the subscript expression is not within the number of entries of the list or is negative, the value of the switch designator is undefined and will produce a failure in the KDF 9 ALGOL system).

(3) The evaluation processes (1) and (2) are repeated for the new designational expression thus found using current values of all variables involved, and so on, until finally a unique label is reached.

Problem

Follow the action of the program below, noting all labels passed and the final values of n and m.

```
begin  integer n, m;
       switch  Branch := L1, L2, L3;
       switch  S := R, Branch[n - m], R;
       n := 3; m := 1;
   R:  goto if n>0 then Branch [n]
                   else  STOP;
   L1: m := n - m + 1;
   L2: n := n - 1;
   L3: goto S[m + 1];
   STOP:
end
```

22· BLOCK STRUCTURE AND ASSOCIATED RESTRICTIONS 22·

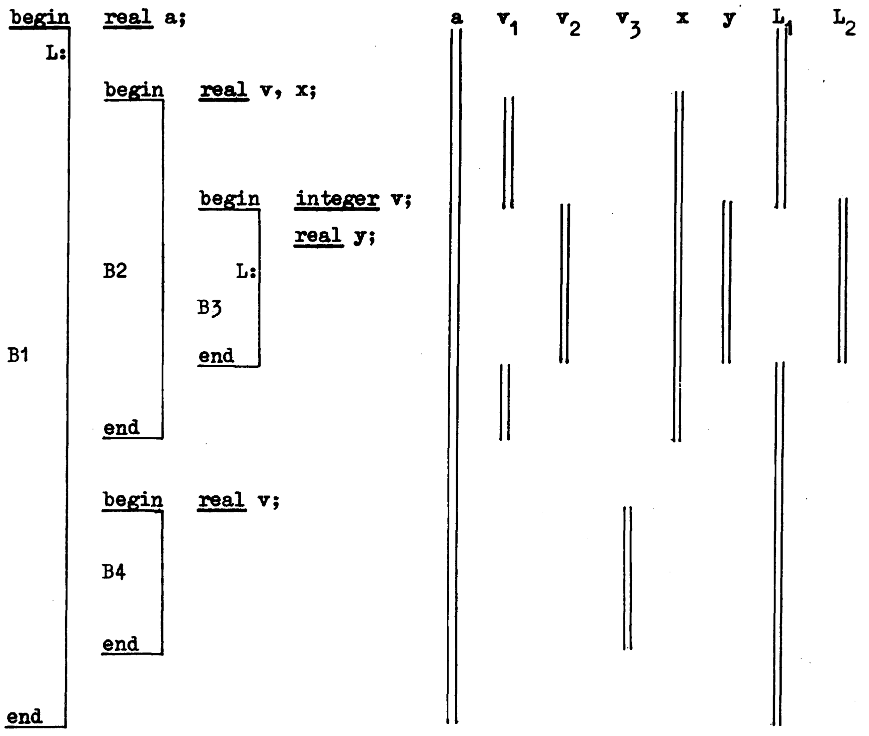
Let us remind ourselves at this stage that a block consists essentially of a sequence of statements preceded by one or more declarations, the whole being surrounded by begin and end brackets and containing a sprinkling of semicolons to act as separators. The form of a compound statement differs from a block in that it contains no declarations.

The beginner was advised in Section 18·4 to write his program as a single block with all declarations inserted at the beginning of the program. We now suggest that the reader make use of the advantages to be gained from a block structure. (These are mainly advantages in storage economy). As he reads this section he will appreciate that the block structure often allows identifiers to have more than one meaning. In his own programs, however, he is strongly urged to use identifiers uniquely.

The definition of Section 18·3 which allows a compound statement or a block to be considered as an unconditional statement makes it possible for the structure of a program to be quite complicated. Here we are interested in the structure arising from the use of blocks rather than that due to the occurrence of compound statements, because the declarations at the heads of the blocks impose certain restrictions on the use of the identifiers declared. The present section will explain these restrictions.

On page 78 (overleaf) we provide a program which illustrates how blocks may appear in sequence with other statements and blocks, or may be nested, that is, may appear within other blocks. The block structure in skeletal form is depicted on the right hand half of page 79.

```
begin  real  a ;
       open (20);
    L: a := read (20);
       a := a + 1/a;
       begin  real v, x;
              v := a + 1;
              x := v↑2 + v↑(-2);
              for v := 0 step 0.1 until 1.55 do
                  begin  integer v;
                         real y;
                         y := 0;
                     L:  for v := 0 step 1 until 10 do
                                 y := y + x↑v;
                         for v := 1 step 1 until 9 do
                             begin  y := y + y↑v;
                                    x := x + y
                             end
                  end    ;
              v := x/a
       end    ;
       if a>0 then goto L;
       begin  real v;
              v := a - 1;
              v := v↑2 + v↑(-2)
       end    ;
       if a<0 then goto L
end
```



```
begin  real a;
   L:
       begin  real v, x;

              begin  integer v;
                     real y;
       B2        L:
              B3
B1            end

       end

       begin  real v;

       B4

       end

end
```

Block Structure   Scopes of Identifiers Declared

It will be noticed that the program reproduced above consists of a block, the block B1. Blocks B2 and B4 are nested within B1 and block B3 is nested within B2. Blocks B2 and B4 follow the general sequence of the program within block B1.

Having determined the block structure of this or any program, we may relate to the structure the restricted scope of each of the entities represented by the identifiers appearing in the program. By scope of an entity we mean that part of the program where its identifier may be legitimately employed to represent it. For the present example the right side of the diagram shows the scope of the variables a, v, x, and y and the labels L determined according to certain rules now to follow.

In Section 18·1, it was stated that a declaration must be placed at the head of the block to which it is intended to apply. On its own, this rule is insufficient to fix the scope of the entities. Confusion of scopes is particularly liable to occur if an identifier is used to represent more than one entity. It is therefore necessary to add the following general rules:

(1) No identifier may be declared more than once in any one block head, nor may an identifier be both declared and occur as a label* or occur as a label twice in the same block.

(2) The entity represented by an identifier declared in a block head does not exist outside that block.

(3) The entity represented by an identifier declared in the head of a block is inaccessible in an inner block, if the identifier has there been re-declared or occurs as a label.*

(4) A label is not accessible from outside the block in which it occurs. This means that though it is possible to jump out of a block by means of a goto statement, it is not possible to jump into a block. All entries to a block must be through the begin.

(5) A label occurring in a given block is not accessible from an inner block, if the corresponding identifier occurs as a label in the inner block or has been declared in its head.

Before going on to discuss the application of these rules to the example given above, we illustrate each rule by noting incorrectly and correctly written programs.

INCORRECT PROGRAMS

CORRECT PROGRAMS

RULE (1)

```
begin   real x;
 |      integer x;
 |      .......
end
```

```
begin   real x;
 |      ......
 |      begin   integer x;
 |       |      .....
 |      end
end
```

*By occurrence as a label, we mean the occurrence of the identifier as a label on the left hand side of a statement and separated from it by a colon.

22· Block Structure and Associated Restrictions (cont.) 22·

```
        INCORRECT PROGRAMS                      CORRECT PROGRAMS


begin   real L;                         begin   real L;
        ....                                    ....
   L:   ....;                                   begin   real a;
        ....                                            ....
        goto L;                                    L:   ....;
        ....                                            ....
end                                                     goto L;
                                                        ....
                                                end
                                        end


begin   real x;                         begin   real x;
   L:   ....                               L:   ....
        ....                                    begin   real y;
   L:   ....                                       L:   ....
        goto L                                          goto L
end                                             end
                                        end
                                        (goto refers to second label L)
```

RULE (2)

```
begin   real x;                         begin   real x;
        ....                                    ....
        begin   real y;                         begin   real y;
                ....                                    ....
                y := ...                                y := ...;
        end     ;                                       write (10, format
        write (10, format                                   ([dd]),y)
                   ([dd]), y)                   end
end                                     end
```

INCORRECT PROGRAMS

RULE (3)

```
begin   real X;
        ....
        begin   integer i, j;
                ....
           X:   ....;
                X := i × j;
                goto X
        end
end
```

RULE (4)

```
begin   real a, b;
        ....
        begin   array M [2 : 50];
                ....
           L:   ....;
                ....
        end     ;
        goto L
end
```

CORRECT PROGRAMS

RULE (3)

```
begin   real X;
        ....
        begin   integer i, j;
                ....
           L:   ....:
                X := i × j;
                goto L
        end
end
```

```
begin   real X, Y;
        ....
        begin   integer i, j;
                ....
           X:   ....;
                Y := i × j;
                goto X
        end     ;
        X := Y
end
```

RULE (4)

```
begin   real a, b;
        ....
        begin   array M [2 : 50];
                ....
           L:   ....;
                ....
                goto L
        end
end
```

INCORRECT PROGRAMS

RULE (5)

```
begin   real x;
        ....
    L:  ....
        begin   boolean L;
                ....
                goto L
        end     ;
        ....
end
```

CORRECT PROGRAMS

```
begin   real x;
        ....
    L:  ....
        begin   real y;
            L:  ....
                goto L;
                ....
        end     ;
        goto L
end
```

(Note that the first goto refers to the second label L and the second goto refers to the first label L).

Returning to our example and applying to it the rules described above we obtain the scopes of a, v, x, y and L depicted in the diagram of block structure and scopes of declared variables to be found earlier in this section. Thus the variable a is declared for the outer block B1 and not declared again elsewhere (or used as a label). Rules (2) and (3) therefore allow it to be used in any statement of the program. The variable x is declared by block B2 and is not declared again. Rules (2) and (3) allow it to be used throughout B2 but not outside. The variable y is declared for block B3 and is not declared again. The Rules (2) and (3) allow it to be used throughout B3 but not outside.

The situation with regard to the variable v is a little more complex. It is declared afresh in the head of each of the blocks B2, B3 and B4. On each occasion this is equivalent to declaring a new variable which is entirely independent of the others. The entity represented by v in the declaration integer v, we call v_2. This applies throughout block B3, but v_2 is not accessible outside B3. The entity represented by v in the third declaration of v, we call v_3. This applies throughout block B4 but is not accessible outside. The entity represented by v in the first declaration of v we call v_1. It may not be used outside B2, and even within B2 it may only be used when the declaration of integer v in B3 does not apply, that is, outside block B3.

The first label L is accessible throughout the parts of the block B1 which are outside B3, while the second label L is only accessible within B3.

Problem

(1) In the following program find the scopes of all the identifiers. Follow the action of the program and find the values of those variables which are defined at the label STOP.

```
begin  real W, S, B, C;
       W := 8;
       S := 3;
       B := 2 × W - S;
       C := B - W;
       begin  real P, W;
              W := B - 2 × C;
              P := C↑2 - B;
          AA: W := P - 2 × W;
              C := C + 1;
              if W>1 then goto AA;
              S := W - P + S
       end    ;
       W := W - C + S;
STOP:
end
```

(2) In the above program find the number of unlabelled basic statements, basic statements, unconditional statements, statements, block heads, compound statements and blocks.

22·1 The Relation between Procedures and the Block Structure

As far as restrictions on identifiers are concerned, the body of a procedure is treated as if it were a block, whether this be so or not. Specifications are treated as declarations; so that formal parameters, in particular those called by value and therefore used as working variables, are no longer accessible after exit from the procedure body.

Of all the entities declared outside a procedure, its body may only operate upon those which are current at the time of the procedure call. This applies, whether they are inserted via the actual parameter list or already occur inherently within the procedure body as non-local quantities. KDF 9 ALGOL makes a further stipulation upon the latter class. All non-local quantities occurring in the procedure body as declared must be accessible at the time of the procedure declaration when they must have the same meaning as at the time of the call.

As noted in a footnote of Section 19·4 any conflict which arises between identifiers introduced to the body of a procedure via parameters called by name and identifiers already present within the procedure body is resolved by suitable systematic changes of the formal or local identifiers involved.

22·2 Restrictions Imposed upon Array Bounds by the Block Structure

The block structure also imposes restrictions upon array bound expressions occurring in an array declaration in some block head. They may only depend on variables and procedures which are non-local to the block for which the array declaration is valid. That is to say all variables and procedures which occur in these expressions must have been declared outside in some enclosing block. Standard functions and constants can of course always be used in these expressions, for these are considered to be declared in a block enclosing the whole program.

The effect of the above restriction is to prevent the use of any other than constant bounds in the outermost block of a program. It also restricts the data input of arrays with variable bounds, because the bounds cannot be read with the arrays themselves when more than one array is declared in a single block head.

The restriction on reading the bounds of an array of data with the array itself reduces the convenience of using ALGOL arrays to represent matrices. However, a matrix scheme avoiding this difficulty and yet remaining within ALGOL will be found described in the 'English Electric' Manual, 'KDF 9 Matrix Scheme'.

22·3 The Influence of Block Structure on Switch Designators

Under the rules mentioned in Section 22 it is possible for identifiers occurring in designational expressions belonging to a switch declaration to have been re-declared with new meanings by the time the corresponding switch designator occurs. If this is so the conflict between the identifiers occurring in the designational expression and those whose declarations are valid at the place of the switch designator is resolved by suitable systematic change of the latter identifiers.

22·4 Use of the Block Structure

In order to avoid the complex restrictions on identifiers imposed by a block structure, it was suggested in Section 18·4 that the beginner should write his program as a single block with all declarations inserted at the beginning of the program. Of the rules in Section 22 above only Rule (1) is then necessary. Since by Rule (1) no identifier may represent more than one entity in any single block, the programmer of a single block must ensure that all quantities are represented by their own unique identifier.

It is, of course, possible to ensure the uniqueness in meaning by suitable choice of all identifiers and at the same time keep the block structure in accord with the recommendation at the beginning of Section 22. This simplifies the rules considerably but not as radically as in the previous paragraph. Only Rules (1), (2) and (4) are now necessary.

Although it would appear simpler and therefore better to write one's program as a single block without an inner block structure, such a block structure can be useful. Its main value lies in helping to economise on data storage requirements.

This is essential when a program is likely to over-reach the capacity of the computer, perhaps because the program is so large that little room is left for data and working space, or because it uses a number of large arrays. The block structure is also useful when it is necessary to construct parts of a large program independently of each other, as for example when more than one programmer is working on the same project. Division by block structure would avoid the confusion due to overlapping use of identifiers representing different quantities. Procedures, of course, can be used in this way and are of particular value when a block of program has wide application.

In the next section we return to the subject of procedures and extend their application further than the incomplete treatment of Section 19.

23.1 Jensen's Device

A very powerful use of ALGOL procedures involves what is called Jensen's Device.* This employs the ability to make actual parameters of a procedure depend upon one another. Suppose for example we declare the procedure Sum series as follows:

```
procedure Sum series (r) Term: (t) Order: (n) Result: (y);
          value n ; integer r, n; real t, y;
          begin  real s ; s := 0;
                 for r := 1 step 1 until n do
                        s := s + t;
                 y := s
          end
```

If this procedure were used in a statement of the form:

Sum series (i, T, m, R)

then its effect would be rather meagre resulting simply in assigning the value $m \times T$ to R.

However, if the parameter corresponding to t is made to depend on that corresponding to r, then n different terms in a series may be summed. Thus, using subscripted variables (though plenty of examples could arise in which subscripted variables are not used), the procedure statement

Sum series (i, A[i-1] × y↑(i-1), 12, R)

would evaluate the series,

$$A[0] + A[1]y + \ldots\ldots + A[11]y^{11} = \sum_{i=1}^{12} A[i-1]\, y^{i-1}$$

and assign the result to R. The coefficients of the series are stored as the elements of the array A.

Other examples using the above procedure are mentioned in the problems appearing after Section 23.6.

23.2 Array Identifiers as Parameters

A rather different use of arrays from that mentioned in the previous section arises when formal parameters of procedure declaration are specified as arrays. In this case the actual parameter cannot be a subscripted variable but both formal and actual parameters appear simply as array identifiers.

*The device, which is used in the example 'Innerproduct' in the ALGOL Report, is due to J. Jensen of Regnecentralen, Copenhagen.

The actual arrays upon which operations are performed already have their bounds declared in the program outside the procedure. It is therefore unnecessary to specify bounds on the formal array appearing in the procedure declaration. The array specification in the procedure specification part merely appears as:

```
(T) array A, A, .... A;
```

The symbol, array, is here used as a specifier as allowed by Section 19.5. Note that in KDF 9 ALGOL formal and actual arrays must exactly correspond in type.

A good example of the use of an array in a parameter list is found in the procedure Transpose of the ALGOL 60 Report, para 5.4.2. The first parameter conveys the name of a two dimensional square array, while the second parameter conveys its size.

```
procedure  Transpose (a) Order: (n); value n;
           array a; integer n;
           begin   real w; integer i, k;
                   for i := 1 step 1 until n do
                       for k := 1 + i step 1 until n do
                       begin   w := a [i, k];
                               a [i,k] := a [k,i];
                               a [k,i] := w
                       end
           end     Transpose
```

Note that the parameter a appears in the specification part as array and in the body of the procedure is used with subscripts attached.

The reader might wonder at the need for parameters which are arrays. Why not use subscripted variables and insert extra parameters for use in varying the subscripts? This can be done, but may not be convenient as the reader will discover from revising the procedure Transpose above.

23.3 Procedure Identifiers as Parameters

The use of procedure identifiers in a procedure parameter list is another important facility. The specification in the specification part appears as:

```
(T) procedure P, P,.... P;
```

The following declaration of the procedure CONVOLUTE makes use of three procedure identifiers Int, g and h in its parameter list. Note the way these are specified and later used, g and h in function designators and Int in a procedure statement.

```
real procedure  CONVOLUTE (Int, g, h, a, b);
        value a, b;
        procedure  Int; comment The integration process required is
                                supplied through the parameter Int;
        real procedure  g, h; comment The parameters g and h supply
                                the functions appearing in the
                                integrand;
        real  a, b; comment The parameters a and b supply the lower
                            and upper limits of integration;
        begin real u, R;
              Int (g(u) h(u),u,a,b)Result:(R);
              CONVOLUTE :=R
        end
```

The procedure CONVOLUTE is intended to evaluate the integral

$$\int_a^b g(u)\ h(u)\ du$$

The names of various real procedures may be inserted for various functions g(u) and h(u), while various integration processes may be incorporated via the procedure identifier Int. The variable u declared in the procedure body is equivalent to the variable of integration.

23·4 Switch Identifiers and Designational Expressions as Parameters

Of the list of specifiers allowed to appear in the specification part of a procedure declaration by Section 19·5, there remain undiscussed the symbols label and switch. For both these the specifying symbol is followed by one or more formal parameters to which the symbols apply in a manner analogous to the specifications described in the previous two sections.

label is the specifier used when the formal parameter corresponds to actual parameters which are designational expressions, each actual having a label as its value. It is by this means that the programmer may best jump out of a procedure which is to be used in several contexts, say for a failure. The use of non-local labels would often be inconvenient.

Example:

```
    procedure  Complex Divide (a, b, c, d, e, f, Failure);
         value  a, b, c, d;
         real   a, b, c, d, e, f;  label Failure;
         begin  real g;
                g := c↑2 + d↑2;
                if g = 0 then goto Failure
                e := (a × c + b × d)/g;
                F := (b × c - a × d)/g
         end
```

The specifying symbol switch is used when the formal and actual parameters are switch identifiers. In this case a complete switch is transferred via the parameter list. The facility, which is illustrated by the following procedure will not be found of frequent application. Use is made of an own variable, for explanation of which see Appendix 4.

```
procedure  GOTO (S, bool);
     switch S; boolean bool;
     begin   own integer i;
             i := if bool then i + 1 else 1;
             goto S[i]
     end
```

The above procedure will cause a jump to a label of the actual switch supplied in place of S. If bool be false then the first label is used. If bool be true, the position of the label to be used is stepped on by one.

23.5 Recursive Use of Procedures

When an ALGOL program is being executed and control reaches the procedure identifier of a procedure statement or function designator, the identifier initiates a call of the procedure according to rules already explained in Section 19. It may happen that, in the process of executing the procedure, control reaches the same procedure identifier again in a position where it expects to give rise to a new call of the procedure. This is allowable and in the jargon is called a recursion.

We have already met a simple type of recursion without having called it such. The solution of Problem 4 (ii), Section 19 makes use of the procedure identifier Jo recursively in the function designator Jo(Jo(x↑2)). Here the second call of the procedure Jo arises from the arithmetic expression inserted as the actual parameter for the first call. The procedure Jo itself is not recursive, but the use made of it is.

In this example as in all cases of recursion the new call of the procedure sets up a new layer of storage for parameters and locally declared quantities,* so as not to interfere with those already current for the first call. Further levels of recursion may be entered at appropriate calls of the procedure, thus we might use the following function designator, Jo(x + Jo(Jo(x↑2))-1), which recurses twice.

*However, new storage is not required of course for parameters called by name, or for own variables and arrays which as far as their storage is concerned are treated as non-local to the procedure (See Appendix 4).

The above recursive use of a simple procedure by making a new call of the procedure in the actual parameter list is the simplest form of recursion. Another type of recursion arises when the new call of the procedure lies within its own body. The following is an example of a recursive procedure to calculate the binomial coefficient ${}^{n}C_{r}$:

```
integer procedure BC (n,r); value n,r; integer n,r;
        BC := if r = 0 then 1 else (n-r+1)/r × BC(n,r-1)
```

In this procedure the body contains two occurrences of the procedure identifier, the occurrence on the left of the assignment is not a call of the procedure and does not produce a recursion. The second occurrence does occasion recursions.

It should be pointed out that though the use of this second type of recursion may often produce an elegant ALGOL program, more often than not a less efficient use is made of KDF 9 by this means than by straight-forward ALGOL programming.

23·6 Use of Non-local Variables in Procedure Bodies

The manual has already stated in Section 22·1 that procedures may use non-local variables within their bodies (as long as for KDF 9 ALGOL these are accessible and have the same meaning both at declaration time and at the procedure call). Use of such non-local quantities can occasion unexpected consequences particularly if assignments are made to them within the procedure body.

One might have an apparently harmless function designator, Sheep (20), which however has the following declaration:

```
integer procedure  Sheep (s); value s;  real s;
        begin   Sheep := s;
                Wolf := 2 × Wolf end
```

A call of this procedure will not reveal openly the effect upon the non-local variable Wolf, and because of this hidden 'side effect' the two expressions:

```
                    Sheep (20) × Wolf
and                 Wolf × Sheep (20)
```

will lead to two different results.

We assume here that the operands occurring within an expression are always evaluated from left to right (in addition to operations which are usually performed in this order, see Section 6). This is the case with KDF 9 ALGOL. There is, however, no express ruling on this matter in the ALGOL 60 Report, so that other compilers may adopt a different order of evaluation and therefore produce a different result when 'side effects' are involved.

Other procedures than those used as function designators can produce 'side effects' but function designators are the more insidious in practice as they are capable of being used in the very varied positions allowed for expressions. Thus the mere evaluation of an array subscript or the obedience of a goto statement using a designational expression may produce an effect on other quantities.

There is, however, very little excuse for the average ALGOL programmer obscuring his program by the use of procedures having such hidden effects, since he can always bring their effects into the open by incorporating non-local variables in the procedure parameter lists, calling these variables by name.

Problems

(1) Use the procedure Sum series of Section 23.1 to sum to n terms

(i) an arithmetic progression, first term a, common difference d,

(ii) a geometric progression, first term a, common ratio r.

(2) Use the procedure Sum series to produce the effect of the procedure statements:

(i) Spur (A) Order: (7) Results to : (V)

(ii) Innerproduct (A[t,P,u], B[P],10,P,Y).

Procedure declarations of Spur and Innerproduct appear in the ALGOL 60 Report, para 5.4.2

(3) Construct a type procedure to evaluate the area under a curve using Simpson's Rule, expecting an array of the co-ordinates at equal intervals of the independent variable to be provided as one of the parameters.

$$\left[I = \frac{b-a}{3n}\,(y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \ldots\ldots + 2y_{n-2} + 4y_{n-1} + y_n)\right.$$

independent variable passing from a to b, co-ordinates y_i and n even.]

(4) Write a recursive procedure to discover the highest common factor of two integers p and q.

CHECKING AN ALGOL PROGRAM BEFORE TEST

Although the simplicity and efficiency of ALGOL programming is such that one may expect a fairly low error density when compared with other forms of programming, there is still a strong possibility of errors occurring, particularly in large programs. To prevent wastage of machine time in re-translation, it is most important that an ALGOL program should be checked by hand before translation is started. The following are points to notice particularly.

(i) Check that the underlining of basic symbols has not been forgotten.

(ii) Check that semicolons have not been forgotten. Look particularly at the ends of lines, between declarations and specifications, at the ends of declaration and specification lists, and following comments introduced by the basic symbol <u>comment</u>.

(iii) Check that each <u>begin</u> has a corresponding <u>end</u> and that each <u>if</u> has a corresponding <u>then</u>.

(iv) Check for the omission of compound statement brackets <u>begin</u> and <u>end</u>, such as those which should appear round for statement bodies and the branches of conditional statements, when they contain more than one independent statement.

(v) Check that an <u>if</u> never follows a <u>then</u>, or an arithmetic, logical, or relational operator.

(vi) Check that except within strings the exponent base $_{10}$ is only used within numbers and is always followed by a signed or unsigned integer number (not a variable).

(vii) Check that two arithmetic or two logical operators do not appear in juxtaposition.

(viii) Check that each opening bracket in an arithmetic expression has a corresponding closing bracket and vice versa.

(ix) Check that the multiplication sign × has not been omitted.

(x) Check that <u>integer</u> division ÷ is only used to operate upon <u>integer</u> operands.

(xi) Check that declarations of simple variables have not been omitted.

(xii) Check that specifications of formal parameters have not been omitted, and also that upper and lower bound information has <u>not been provided</u> for array specifications.

(xiii) Check that each variable has not been used before a value has been assigned to it.

(xiv) Check that after exhaustion of the for list of any for statement the controlled variable is not used again until an assignment is first made to it. (The controlled variable keeps its current value if exit from the for statement is by a goto statement <u>before exhaustion</u>).

(xv) Check that no division by zero, square root of negative quantity, logarithm of zero or negative quantity, or disallowed use of the exponentiation operation, etc., can occur.

(xvi) Check that the absolute value has been taken when testing magnitudes of quantities.

(xvii) Check the program thoroughly by following it through step by step using test data to enable one to check all parts of the program. Check also for special values of parameters, such as zero.

Since it is still possible that an ALGOL program may contain errors, even after the above checks have been made, automatic checking facilities are incorporated in the translator. Nearly every disobedience of the rules of KDF 9 ALGOL is discovered by the translation process and notice of it printed out, pin-pointing the position of the error in the program so far as this is possible. In the remaining few cases however, such as incompatibility of formal and actual parameters, the error is not discovered until the program is run. Again notice will be given. There are also other checks automatically made at run-time, such as those needed to ensure that the storage capacity of the machine is not exceeded or that numbers do not become too large during calculation.

It is, of course, not possible to check automatically for a wrong program. The translator will accept for translation any program which obeys the rules and the KDF 9 ALGOL system will run it. The programmer himself must compare the results produced with those he desires, before he may be sure of having the right program.

In order to help the programmer discover where a program has been written wrongly, he is able to output partial results and other information by means of program-testing procedures. The identifier of each such procedure must commence with a particular group of letters and is written and declared by the programmer himself. When the program is compiled in the testing mode, procedure statements and declarations using these identifiers are included; while in the non-testing mode they are excluded For further details see KDF 9 Library Service Note - ALGOL Note 1.

Problem

List the errors in the following program:

```
begin   comment Program to evaluate π
        real term, s, pi;
        n := 0;
        for n := n + 1 while abs(term)>₁₀-10 do
                term := 1/(2n - 1)↑4;
                s := s + term;
                pi := sqrt (sqrt(96s)
                write (10, format([d.dddsdddsddd]), pi)
end
```

(Problem continued overleaf).

The intention is to evaluate π from the series

$$\pi^4/96 = \frac{1}{1^4} + \frac{1}{3^4} + \frac{1}{5^4} + \ldots\ldots$$

continuing until terms are less than 10^{-10}. The program as given acove contains thirteen errors.

A SPECIMEN ALGOL PROGRAM

The specimen program supplied here should help to give readers some idea of the appearance of a program for a practically occurring problem in engineering. The program which is in strict KDF 9 ALGOL illustrates many of the programming tools available to the ALGOL user.

The problem and numerical method of solution is as follows:

Tabulate the functions $\lambda(\alpha)$, $\lambda \times (2\alpha)^{\frac{1}{2}}$ and $g(\alpha)$ for $\alpha = 0 \; (\Delta\alpha) \; 1.0$, where $\lambda(\alpha)$ is given by

$$\alpha(1 - \alpha) = (\tfrac{1}{2}\sqrt{\pi} \; e^{\lambda^2} \; \operatorname{erfc}\lambda)/\lambda$$

and $g(\alpha)$ is given by

$$\begin{aligned} g(\alpha) &= 2\lambda^2 \alpha(1 - \alpha), & \lambda \geqslant 1/\sqrt{2} \\ &= \alpha(1 - \alpha)e^{\lambda^2}/(\lambda\sqrt{2e}), & \lambda \leqslant 1/\sqrt{2} \end{aligned}$$

The function erfc λ is sometimes called the complementary error function and is given by the integral

$$\operatorname{erfc}\lambda = \frac{2}{\sqrt{\pi}} \int_{\lambda}^{\infty} e^{-x^2} \, dx$$

An initial value of λ is obtained from the approximation

$$\lambda = (1 - \alpha)/\sqrt{2\alpha}$$

This value of λ is improved using the following formulae in an iterative manner:

$$l = \sqrt{\left\{\frac{\sqrt{\pi}}{2} \frac{(1 - \alpha)}{\alpha} (\lambda \operatorname{erfc}\lambda \; e^{\lambda^2})\right\}}$$

and $$\text{new}\lambda = l + 0.835\alpha(l - \lambda)$$

When α is zero, λ becomes infinite and the above formulae cannot be used in numerical calculation. In this instance the limiting values are output directly.

It is not claimed that the numerical method used for solving the above problem is particularly efficient. In fact the method used to evaluate erfcλ, being based upon the trapezoidal rule, is rather slow (except when λ is near zero). For present purposes this does not matter as we wish to illustrate a form of programming rather than produce a fast program. The problem is taken from a paper by J. W. Miles (The Propagation of an Impulse into a Viscous-locking medium, A.S.M.E. Trans. Series E. Jour. Appl. Mechs. March 1961, 21 - 24).

```
begin   comment This is a program for a practically arising problem.  It
        illustrates the use of many of the facilities available to the
        ALGOL user;
        real  alpha, lambda, Delta alpha; integer i, f1, f2, f3, f4, f5;
        real procedure  erfc (z); value z; real z;
             begin  comment This procedure evaluates the complementary
                    error function of z using the trapezoidal rule.  It
                    halves the interval until the required accuracy is
                    attained, but avoids repeating the evaluation of
                    ordinates more than once;
                    real  x, h, J0, J1; integer n, i;
                    h := 0.1; J0 := 0; n := 0; i := 1;
                R:  J1 := J0;
                    for n := n, n + i while
                        n × h × (n × h + 2 × z)<11.51 do
                        begin  x := z + n × h;
                               J0 := J0 + exp(-x↑2)
                                     × (if n = 0 then 0.5 else 1)
                        end    ;
                    if abs(1 - 2 × J1/J0)>0.00001 then
                        begin  i := 2; n := 1; h := h/2;
                               goto R
                        end    ;
                    erfc := 1.128379 × J0 × h
             end    erfc;
        library  A6;  comment The word library under lined and followed by a
                      list of numbers separated by commas and ending with semi-
                      colon is an instruction to the KDF 9 ALGOL operating
                      system to insert at this point the specified passages of
                      ALGOL text from the library.  In this case library A6
                      is required - this contains the declarations of the
                      input and output procedures named open, close, read,
                      write, format, and write text.
        open (20);
        Delta alpha := read (20); close (20);
        comment Delta alpha is the only input data item required by the
        program;
        open (10);
        write text (10, [Propagation*of*an*Impulse*into*a*Viscous-locking*
                         Medium [4c4s] Delta*alpha*=*]);
        write (10, format ([d.ddddccc]), Delta alpha);
```

```
      write text (10, [[4s] alpha [15s] lambda [ 7s] lambda
                × sqrt (2 × alpha) [9s] g [2c 4s] 0.0000 [13s]
                  INFINITY [13s] 1.00000 [12s] 1.00000 [ c]]);
      f1 := format ([ssssd.dddd]);
      f2 := format ([12sd.dddd₁₀+nd]);
      f3 := format ([12sd.ddddd]);
      f4 := format ([12sd.dddddcc]);
      f5 := format ([12sd.dddddc]);
      i := 1;
      for alpha := Delta alpha step Delta alpha until 1.0 do
           begin  real l1, l2;
                  i := i + 1;
                  lambda := (1 - alpha)/sqrt (2 × alpha);
                  if alpha = 1 then goto SKIP;
         Repeat:  l1 := lambda;
                  l2 := sqrt (0.886227 × (1-alpha)/alpha × l1
                        × erfc (l1) × exp(+l1↑2));
                  lambda := l2 + 0.835 × alpha × (l2-l1);
                  if abs (1-l2/lambda)>₁₀-5 then goto Repeat;
           SKIP:  write (10, f1, alpha);
                  write (10, f2, lambda);
                  write (10, f3, lambda × sqrt (2 × alpha));
                  write (10, if i - i ÷ 5 × 5 = 0 then f4 else f5,
                        if lambda≥.70710678
                        then 2 × lambda↑2 × alpha × (1-alpha)
                        else if lambda ≠ 0
                             then alpha × (1 - alpha) × exp (lambda↑2)/
                                  (lambda × 2.3282180)
                             else 0.48394)
           end    ;
      close (10)
end   program
```

Layout of Results

Below we show the layout of results expected from the above program for the input data tape containing 0.1→. The results are first punched out on paper tape and subsequently tabulated.

Propagation of an Impulse into a Viscous-locking Medium

Delta alpha = 0.1000

alpha	lambda	lambda×sqrt(2×alpha)	g
0.0000	INFINITY	1.00000	1.00000
0.1000	2.0201_{10} +0	0.90340	0.73452
0.2000	1.2812_{10} +0	0.81029	0.52525
0.3000	9.2712_{10} -1	0.71814	0.36101
0.4000	6.9943_{10} -1	0.62559	0.24038
0.5000	5.3160_{10} -1	0.53160	0.26796
0.6000	3.9727_{10} -1	0.43519	0.30384
0.7000	2.8340_{10} -1	0.33532	0.34489
0.8000	1.8242_{10} -1	0.23075	0.38947
0.9000	8.9284_{10} -2	0.11979	0.43642
1.0000	0.0000	0.00000	0.48394

SIMPLE BOOLEAN EXPRESSION - GENERAL FORM

Amongst the logical operators which may be used in a boolean expression we have not ($\neg$), and ($\wedge$), or ($\vee$), imp ($\supset$) and eqv ($\equiv$).* Denoting these logical operators by LO, a logical value by LV, a boolean type variable by BV and a relation by R, we may write a simple boolean expression in the form:

$$\begin{matrix} \text{LV} & & \text{LV} & & & & \text{LV} \\ \text{BV} & \text{LO} & \text{BV} & \text{LO} & \cdots\cdots & \text{LO} & \text{BV} \\ \text{R} & & \text{R} & & & & \text{R} \end{matrix}$$

For example we might have the following simple boolean expression:

$$\underset{\text{BV}}{\text{B1}} \quad \underset{\text{LO}}{\underline{\text{and}}} \quad \underset{\text{LV}}{\underline{\text{true}}} \quad \underset{\text{LO}}{\underline{\text{or}}} \quad \underbrace{x<y}_{\text{R}}$$

The various components of the expression are marked.

The function of the logical operators is as follows:

not will change the value of the boolean quantity which follows it.

and will take the pessimistic view that the result of the operation on the two boolean quantities on either side of it is true if both have the value true, otherwise the result is false.

or will take the optimistic view that the whole has the value true if at least one of the two boolean quantities on either side has the value true otherwise the result is false.

imp short for implies, will produce the result true if the boolean quantity to the right of the symbol is at least as true as the boolean quantity to the left.

eqv short for equivalent, will produce the result true if the boolean quantities on either side have the same value.

Evaluation of a simple boolean expression proceeds from left to right except that the following order of precedence must be observed:

1st	arithmetic expressions in accord with Section 6 and 8
2nd	relational operators
3rd	not
4th	and
5th	or
6th	imp
7th	eqv

*The KDF 9 ALGOL (flexowriter) symbols are given here with the ALGOL 60 equivalents in parentheses.

Brackets may also be used within boolean expressions to alter the natural order of evaluations.

Examples:

(1) If x = 0, y = -2, z = 5, find the value of the boolean expression

<u>not</u> (x<2 <u>and</u> z>6 <u>or</u> 2 + y = 0)

The following steps are necessary. The first operator, <u>not</u>, is followed by a bracket which must be evaluated first. We have then,

<u>not</u> (<u>true</u> <u>and</u> z>6 <u>or</u> 2 + y = 0)
<u>not</u> (<u>true</u> <u>and</u> <u>false</u> <u>or</u> 2 + y = 0)
<u>not</u> (<u>false</u> <u>or</u> 2 + y = 0)
<u>not</u> (<u>false</u> <u>or</u> 0 = 0)
<u>not</u> (<u>false</u> <u>or</u> <u>true</u>)
<u>not</u> (<u>true</u>)
<u>not</u> <u>true</u>
<u>false</u>

Note: to be certain that 2 + y comes to exactly zero, y must be an <u>integer</u> type variable.

(2) For the same values of x, y and z, evaluate

<u>not</u> x<2 <u>or</u> z>6 <u>and</u> y ≠ 3

Again, we follow each step through.

<u>not</u> <u>true</u> <u>or</u> z>6 <u>and</u> y ≠ 3
<u>false</u> <u>or</u> z>6 <u>and</u> y ≠ 3
<u>false</u> <u>or</u> <u>false</u> <u>and</u> y ≠ 3
<u>false</u> <u>or</u> <u>false</u> <u>and</u> <u>true</u>
<u>false</u> <u>or</u> <u>false</u>
<u>false</u>

<u>Problems</u>

(1) If a = 1, b = 1.5, c = -0.3, d = 2 find the values of the following simple boolean expressions. a and d are <u>integer</u> type and b and c <u>real</u>.

(i) b<d
(ii) a + b≥(1 + c↑2 x d)
(iii) b>c <u>and</u> d<2
(iv) c↑d ≠ a <u>or</u> b↑2 - a = d <u>or</u> <u>not</u> b≥1.499

(2) If a, b, c and d are variables of type **integer**, which of the following are valid simple boolean expressions?

(i) a = 2↑d **and** 5 = 4 **or** **not** **true**

(ii) a = 2↑d **not** c - 1 **and** 2<a

(iii) b<(a **and** a<d) **and** (d<c) **or** c<b

(iv) **not** d + c ÷ b>b **or** **true** **or** c = 2

(3) If B1, B2, B3 are **boolean** variables such that B1 and B3 have the value **true** and B2 has the value **false**, find the value of the following boolean expression:

not (B1 **and** B2 **or** B3 **and** **true**) **or** **not** B3 **or** (B1 **and** **false**)

(4) Show that whatever the values of the boolean quantities b1 and b2, the value of the expression

(b1 **imp** b2) **eqv** (**not** b1 **or** b2)

is always **true**.

APPENDIX 3

ACTION OF FOR LIST ELEMENTS

The action of a for statement of the form

for V := A step B until C do S,

where A, B and C are arithmetic expressions, V is a variable and S a statement, may be described in terms of the following ALGOL statements.

```
    V1 := V := A;
    V2 := B;
L:  if sign (V2) × (V1 - (C))>0
    then goto Element exhausted;
    S;
    V2 := B;
    V1 := V := V + V2;
    goto L
```

V1 and V2 are auxiliary simple variables, V1 of same type as V and V2 of type real. V2 is used so that the above statements may evaluate the expression B only once per cycle. The statements are also arranged so that A, B and C are evaluated in the correct order. This is of importance if the expressions are such as to introduce side effects. The use of V1 helps to reduce the occurrence of side effects introduced via the subscripts of V, if it is a subscripted variable.

The statement,

goto Element exhausted

leads on to the next element in the for list which recommences assignment to the controlled variable according to this new element. If there is no new element in the for list, as in the for statement written above, control passes to the next statement in the program.

The action of the while element occurring in a statement of the form

for V := E while F do S,

where E is an arithmetic expression, F a boolean expression and V and S as above, may be described in terms of the following ALGOL statements.

```
L : V := E;
    if not (F) then goto Element exhausted;
    S;
    goto L
```

APPENDIX 4

OWN VARIABLES AND ARRAYS

The symbol own is available to designate variables and arrays as own. Own quantities like others may be used in the block and only in the block where they are declared. They differ from others in keeping their values unchanged on exit from a block, so that on re-entry to the same block access is available to the old values.

A simple variable or an array is designated own by preceding the type symbol by the declarator own in the type declaration, or array declaration. The type symbol may not be omitted for own array declarations. In parameter specifications, however, the symbol own may not be used.

Examples of declarations using own:

```
own integer x, y
own real array PIG [1:30, 1:40]
```

The following declaration would not be allowed;

```
own array A[1:10]
```

There is a restriction on own arrays in KDF 9 ALGOL. The bound pairs in their array declarations must be constant. In the jargon of ALGOL experts, 'dynamic own arrays' are not allowed.

The effect of recursion on an own quantity is the same as the effect of a normal re-entry to the block in which it is declared. One and the same quantity becomes available; no new quantity is defined on a fresh level.

PROCEDURE BODIES IN KDF 9 CODE

See ALGOL Users Manual for full details.

STRINGS

The form of a string may be defined as follows: A string is any sequence of basic symbols such that each string quote [or] contained therein has a corresponding string quote of the opposite kind; the closing quote] corresponding to an opening quote [must follow it later in the sequence; and the whole sequence must be enclosed in quotes [and].*

Strings are purely of use as parameters for procedures with bodies in code, such as the procedure called format used in Section 17 for output of results.

Within the machine strings are stored as sequences of basic symbols in an 8-bit internal code which is given in the last column of Appendix 8.

*In the ALGOL 60 Reference Language these symbols are ‘and’.

APPENDIX 7

SOLUTIONS TO PROBLEMS

Section 4

Problem (1)

+729300000	1000	-.000001
9812	-.000001834	-4800

Problem (2)

$17_{10}3$	$-134_{10}-5$
$_{10}3$	

Problem (3)

-.0 08	$-88_{10}-7$
$+ 13.47_{10}+18$	13.411 732

Section 5

Problem

begin	Start value	a29v3
ppp3	number	epsilon

Section 6

Problem (1)

(i) -20.7, real (ii) +5, integer

Problem (2)

(i), (iii), (iv), (vii), (viii).

Problem (3)

(i) 4 (ii) 5 (iii) 16 (iv) 7 (v) 4096 (vi) 0
(vii) 0 (viii) 4

Problem (4)

(i) S + (s - t)/v↑2	(ii) (U - W) × (1 - a↑3/ k /(a - k)
(iii) a↑(n + m)	(iv) a↑(b↑n)
(v) a↑(b + s↑n)	(vi) q↑v↑g
(vii) p↑q / r↑(s + t)	(viii) (a - b/c/(d - e↑(f + q)))/ (h↑i↑(j - k) + q↑(m/(n + p)))

Section 7

Problem

(i) true (ii) true (iii) true (iv) false

Section 8·4

Problem (1)

```
if A>pi/2  then 2 × t/(1 + t↑2)
           else (1 - t↑2)/(1 + t↑2)
```

Problem (2)

```
if x<0 then x - 1 else if x>1 then x + 1
           else x↑2 - 3 × x + 4
```

Various other answers are allowed, but any containing an if following a then are incorrect. Any containing $0 \leq x \leq 1$ as a boolean expression are also wrong. If required, it should be written $0 \leq x$ and $x \leq 1$.

Section 9

Problem

```
exp(2 × abs(cos(3 × a)))
sqrt(ln(arctan(sqrt(a↑2 + b↑2))))
(a × cos(x) + b × sin(x) - 1)/
    (a × cos(x)↑2 + b × sin(x)↑2 + 1)
```

Section 11

Problem (1)

Final values a = 25·87, b = 7, p = 25·87, q = 27·27.

Problem (2)

r1 = 23, ra = 2, rb = 10, n = 2, i = 4, j = 2.

Problem (3)

ra = 4, rb = 12.5, ia = 5, ba = true, bb = true.

Section 12·2

Problem

SUM = 0, 1, 1.25, 1.33333.....

Section 13·3

Problem (1)

Delta x	0.1						
x	0	0.1	0.2	0.3	0.4	0.5	0.6
y	1	0.81	0.64	0.49	0.36	0.25	

In this example 0.55 is used instead of 0.5 after <u>until</u> because of the possibility of rounding errors arising when dealing with <u>real</u> quantities. If 0.5 were used, the last values of x and y might get omitted.

Problem (2)

```
(i)  S := 0; for i := 1 step 1 until n do S := S + i

(ii) for i := 0 step 1 until n do
                  S := if i = 0 then 0 else S + i
```

Section 13·4

Problem (1)

p	1	3	5	7	9
$p^2 + q^2$		58	41	85	145
q	-7	4	6	8	

Problem (2)

i	2	5	6	7	8	9	10	-1
m	6	30	42	56	72	90	110	

Note that the for list element, -1 <u>while</u> $m < 0$, and in particular its boolean expression are not considered until after the three previous for list elements have been used up. See comments on the for list, Section 13·1.

Problem (3)

```
k = 3, 1, 1, s = 5;  m = 2, 5, 8, 11, 14,  s = -35;
k = 2, -31, 2, 4, 6,  s = -69;  m = -24, -22, -20, -18,
s = 15;
k = 1,  m = 3, 2, 1, 0,  s = 25
k = 2,  m = ditto, s = 39
k = 3,  m = ditto, s = 57
k = 4,  m = ditto, s = 79
k = 5,  m = ditto, s = 105
```

Section 14

Problem (1)

```
for i := 2 step 1 until n do P := (1-(-1)↑i/i↑2) ×
        (if i = 2 then 1 else P)
```

In practice one would write this more naturally in two statements, thus,

```
P := 1;
for i := 2 step 1 until n do P := P × (1-(-1)↑i/i↑2)
```

Problem (2)

```
for s := 0.1 step 0.1 until 0.95 do
        begin  a := t/T × (1 - s); b := 2 × s/(1 - s);
               y := (b↑2 + a/2 × (1 - exp(-2 × a))
                    +2 × a × b/(a↑2 + b↑2)
                    × (exp(-a) × (a × sin(b) + b × cos(b))
                    - b))/(a↑2 + b↑2)
        end
```

Section 15

Problem (1)

u = .86666...., W = -1.13333...., B = true

Problem (2)

x and y are real variables.

```
      x := 3.0;
loop: y := x;
      x := 16/(y + 1);
      if abs(y - x)>₁₀-5 then goto loop
```

An alternative solution not using a conditional statement is:-

```
for y := 3, 16/(y + 1) while abs(x - y)≥⏨-5 do x := y
```

y	3	4	3.2	3.8095....	....
x	3	4	3.2	.....	

After leaving the for statement the controlled variable y, which gives the answer to one step greater accuracy than x, will be lost.

Problem (3)

This problem is solved either by resorting to the contents of Appendix 2 or using a compound statement such as described in Section 14. Using the logical operator and of Appendix 2 we have the following solution:-

```
if B1 and B2 then goto P else if B1 then goto Q else if B2
          then goto R else goto S
```

Problem (4)

```
for n := 1 step 1 until 100 do
            begin   y := x↑n;
                    if y = 1 then goto Singularity
                            else y := (y + 1)/(y - 1)
            end
```

Other answers are possible.

Section 16·2

Problem (1)

```
B[1,1] = 4;  B[1,2] = 1;  B[2,-1] = 2;
   V[3] = 4;  i=1.
```

Problem (2)

```
SC:= SCD:=0;
for i:= 1 step 1 until 15 do
        begin   SC:= SC + CAT[i]↑2;
                SCD:= SCD + CAT[i] × DOG[i]
        end
```

Section 17·9

Problem (1)

```
for i := 1 step 1 until 10 do
              S := (if i = 1 then 0 else S) + read (20)↑2
```

S could be set at zero initially outside the for statement, eliminating the need for an arithmetic expression containing an if clause.

Problem (2)

```
f:= format ([ddd.ddc]);
for i := 1 step 1 until 100 do write (10, f, read (20)↑3)
```

Problem (3)

```
write text (30, [Co-ordinates * of * the * Parabola, * y↑2 = 4x.])
```

Problem (4)

```
write text (30, [**x [9s] y [c]]);
f1:= format ([d.dd]); f2:= format ([ssssssd.ddc]);
for  x := 0 step 0.01 until 5.005 do
              begin write (30, f1, x); write (30, f2, 2×sqrt(x)) end
```

Problem (5)

```
f:= format ([dddddsss]);
for i := 1 step 1 until N do
    begin  x:= read (20);
           B1 := x ÷ 2 × 2 - x = 0;
           B2 := x ÷ 3 × 3 - x = 0;
           write (10, f, x);
           if B1 and B2 then write text (10,[ 2; 3[ c]]);
           if B1 and not B2 then write text (10, [2[ c]]);
           if not B1 and B2 then write text (10, [ 3[c]])
    end
```

Problem (6)

```
f1:= format ([5s-d.ddddddd₁₀+ndc]);
f2:= format ([5s-d.ddddddd₁₀+nd]);
for i := 1 step 1 until 22 do
          write (11, if i ÷ 5 × 5 -i = 0 then f1 else f2,
                         BRUTE [i,i])
```

Section 18·3

Problem (1)

```
begin  real a, b, p, q;
       a := b := 7;
       p := a + 3 x b - 2.3₁₀-1;
       q := p + (a + 3)/(-b - 13);
       a := p := q - b x 0.2
end
```

Note that b cannot be taken as integer type because it appears in a left part list with real a.

Problem (2)

```
begin  real ra, rb; integer ia;
       boolean ba, bb;
       ra := 7.5;
         .....
         .....
       bb := ba and rb>ia and ra<rb
end
```

Section 18·4

Problem (1)

```
(i) begin  real S; integer i;
           open (20);
           for i := 1 step 1 until 10 do
                S := (if i = 1 then 0 else S) + read (20)↑2;
           close (20)
    end
```

Problem (4)

```
begin   real x, sum; integer r, i, n;
        open (20);
        n:= read (20);
        sum := 0;
        for r := 1 step 1 until n do
                  begin   x:= read (20);
                          i := sqrt(r);
                          if r = i↑2 then sum := sum + x
                  end     ;
        close (20)
end
```

Problem (5)

```
begin   integer i, k,n, f;
        integer array j [0:9];
        open (20); n:= read (20);
        for k:= 0 step 1 until 9 do j [k] := 0;
        for i:= 1 step 1 until n do
               begin   k:= entier (read (20)/10 + 0·05);
                       if k>9 then goto miss;
                       j [k] := j[k] + 1;
        miss:  end     ;
        close (20); open (10);
        f:= format ([ddddddc]);
        for k:= 0 step 1 until 9 do write (10, f, j [k]);
        close (10)
end
```

Problem (6)

```
begin   real e, x1, x2, a;
        integer i, n, f;
        open (21);  a:= read (21);  e:= read (21);  n:= read (21);
        close (21); open (10); f:= format ([nddd.dddddd000c]);
        x1 := -0.5 × 3.1415926536;
        for i := 0 step 1 until n - 1 do
            begin   x1 := 3.1415926536 + x1;
                    for x2 := i × 3.1415926536 + arctan(a/x1)
                        while abs(x2 - x1)⩾e do x1 := x2;
                    write (10, f, x1)
            end     ;
        close (10)
end
```

Problem (7)

```
begin   real a, b, c, s, Delta;
        open (20);
        a:= read (20); b:= read (20); c:= read (20);
        close (20);
        s := (a + b + c)/2;
        Delta := sqrt(s × (s - a) × (s - b) × (s - c));
        open (10);
        write text (10, [Delta * = *]);
        write (10, format ([dd.dddd]), Delta);
        close (10)
end
```

Problem (8)

```
begin   integer r, n, BC, f;
        open (20); n := read (20); close (20);
        f:= format ([dddddsddddddc]);
        open (10);
        for r:= 0 step 1 until n do
            begin   BC := if r = 0 then 1 else
                              (n - r + 1)/r × BC;
                    write (10, f, BC)
            end     ;
        close (10)
end
```

Section 19·8

Problem (1)

All the various parts of a procedure declaration appear, commencing with the type declarator, real, and the symbol procedure, and continuing with the procedure heading:-

erfc (z); value z ; real z;

The procedure heading is followed by the procedure body. In this example as is most usual it commences and closes with begin and end brackets.

The value part is: value z;

The specification part is: real z;

Problem (2)

35.

Problem (3)

p = - 1, q = 737, r = 3, s = 11; all others are undefined.

Problem (4)

```
(i) real procedure  Jo(z); value z; real z;
    begin  real term, y;  integer n;
           term := y := 1;
           for n := 1 step 1 until 12 do
               begin
                       term := -term × z↑2/(2 × n)↑2;
                       y := y + term
               end     ;
           Jo := y
    end
```

```
(ii) begin  real x; integer f1, f2;
            real procedure Jo(z); value z; real z;
            begin  real term, y; integer n;
                   .......      (as above)
            end    ;
            open (11);
            f1:= format ([d.dssssss]);
            f2:= format ([-d.dddddsddddc]);
            for x := 0 step 0.1 until 1.05 do
                begin  write (11, f1, x);
                       write (11, f2,
                              Jo(x) + Jo(Jo(x↑2)))
                end    ;
            close (11)
     end
```

Section 21

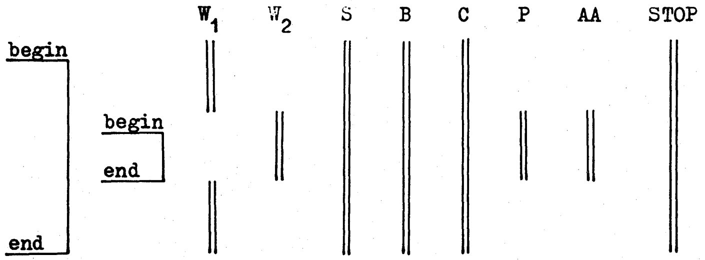
Problem

Labels: R, L3, L2, L3, L1, L2, L3, R, L1, L2, L3, R, STOP.

n = 0, m = 0.

Section 22

Problem (1)



Scopes

W = -8, S = -9, B = 13, C = 7.

Problem (2)

Unlabelled basic statements	12
Basic statements	14
Unconditional statements	16
Statements	17
Block heads	2
Compound statements	0
Blocks	2

Section 23·6

Problem (1)

(i) Sum series (i, a + d × (i-1), n, R)

(ii) Sum series (i, a × r↑(i-1), n, R)

Problem (2)

(i) Sum series (k, A [k,k], 7, V)

(ii) Sum series (P, A [t,P,u] × B[P], 10, Y)

Problem (3)

(Solution adapted from P. E. Hennion, Algorithm 84, Comm.A.C.M., No. 5, April 1962.)

```
real procedure SIM (n, a, b, y);
     value n, a, b;  real a, b;  integer n;  array y;
     begin  real s;  integer i;
            s:= (y [0] -y [n])/2;
            for i := 1 step 2 until n -1 do
                s := s + 2 × y [i] + y [i+1];
            SIM := 2 × (b-a) × s/(3 × n)
     end
```

Problem (4)

```
procedure HCF (p,q,R); value p, q;
        integer p,q,R;
        if  q = 0  then R := p
                   else  HCF (q, p-p ÷ q × q, R)
```

Section 24

Problem

1. π in comment is not a basic symbol of KDF 9 ALGOL as listed in Section 3.

2. ; omitted after comment.

3. n not declared.

4, 5. s and term both used before being assigned values.

6. Underlining omitted from while.

7. begin and end brackets omitted after do. The assignment to s should be included in loop, otherwise terms will not be summed.

8, 9. × sign omitted between 2 and n and also between 96 and s.

10.) omitted:- required to complete arithmetic expression.

11. ; omitted after assignment statement.

12, 13. Device 10 neither opened nor closed.

Appendix 2

Problem (1)

(i) true (ii) true (iii) false (iv) true

Problem (2)

Only (i) and (iv) are valid

Problem (3)

false

Problem (4)

b1	false	false	true	true
b2	false	true	false	true
not b1 or b2	true	true	false	true
b1 imp b2	true	true	false	true

It follows that the complete expression is always true.

BASIC SYMBOLS - STANDARD REPRESENTATIONS

BASIC SYMBOL (Reference Language)	8-CHANNEL (Flexowriter) VERSION	5-CHANNEL (Creed) VERSION	LINE PRINTER (Program Text) VERSION	8-BIT INTERNAL (Octal value) REPRESENTATION
a-z	a-z	A-Z	A-Z	046-077
A-Z	A-Z		A-Z ' '	014-045
0-9	0-9	0-9	0-9	000-011
true	true	*TRUE	TRUE ----	335
false	false	*FALSE	FALSE -----	315
+	+	+	+	301
-	-	-	-	321
×	×	×	*	261
/	/	/	/	241
÷	÷	*DIV	/ I	221
↑	↑	**	U P	201
<	<	*≥	L T	202
≤	≤	*>	L E	222
=	=	=	=	242
≥	≥	≥	G E	262
>	>	>	G T	302
≠	≠	≠	N E	322
≡	eqv	*EQV	EQV ---	303
⊃	imp	*IMP	IMP ---	263
∨	or	*OR	OR --	243
∧	and	*AND	AND ---	223

A·8 Appendix 8 - Basic Symbols - Standard Representations (cont.) A·8

BASIC SYMBOL (Reference Language)	8-CHANNEL (Flexowriter) VERSION	5-CHANNEL (Creed) VERSION	LINE PRINTER (Program Text) VERSION	8-BIT INTERNAL (Octal value) REPRESENTATION
¬	not	*NOT	NOT ---	203
go to	goto or go to	*GOTO	GOTO ----	210
if	if	*IF	IF --	205
then	then	*THEN	THEN ----	225
else	else	*ELSE	ELSE ----	245
for	for	*FOR	FOR ---	206
do	do	*DO	DO --	326
,	,	,	,	246
.	.	.	.	013
$_{10}$	$_{10}$	v	@	012
:	:	→	:	271
;	;	*,	;	230
:=	:=	*=	:=	265
␣	*	£	-	216
step	step	*STEP	STEP ----	266
until	until	*UNTIL	UNTIL -----	306
while	while	*WHILE	WHILE -----	226
comment	comment	*COMMENT	COMMENT -------	200
(	(	(	(	204
)	)	)	)	224
[	[	*(	(*	211

BASIC SYMBOL (Reference Language)	8-CHANNEL (Flexowriter) VERSION	5-CHANNEL (Creed) VERSION	LINE PRINTER (Program Text) VERSION	8-BIT INTERNAL (Octal value) REPRESENTATION
]	]	*)	) *	231
‘	[	*Q	(Q	215
’	]	*U	) Q	235
begin	begin	*BEGIN	BEGIN -----	214
end	end	*END	END ---	234
own	own	*OWN	OWN ---	217
Boolean	boolean or Boolean	*BOOLEAN	BOOLEAN -------	103
integer	integer	*INTEGER	INTEGER -------	102
real	real	*REAL	REAL ----	101
array	array	*ARRAY	ARRAY -----	110
switch	switch	*SWITCH	SWITCH ------	130
procedure	procedure	*PROCEDURE	PROCEDURE ---------	120
string	string	*STRING	STRING ------	172
label	label	*LABEL	LABEL -----	171
value	value	*VALUE	VALUE -----	237

INDEX

INDEX
(Continued)

INDEX
(Continued)

<u>I N D E X</u>
(Continued)

INDEX
(Continued)

NOTES

NOTES

NOTES

NOTES

NOTES