La Trobe University

DEPARTMENT OF MATHEMATICS

ALGOL 60 Programming on the DECSystem 10

David Woodhouse

Revised, August 1975

MELBOURNE, AUSTRALIA

ALGOL 60 Programming on the DECSystem 10

David Woodhouse

Revised, August 1975

© David Woodhouse
National Library of Australia card number and ISBN.
ISBN 0 85816 066 8

INTRODUCTION

This text is intended as a complete primer on ALGOL 60 programming. It refers specifically to Version 4 of the DECSystem 10 implementation. However, it avoids idiosyncracies as far as possible, and so should be useful in learning the language on other machines.

The few features in the DEC ALGOL manual which are not mentioned here should not be needed until the student is sufficiently advanced to be using this text for reference only.

Exercises at the end of each chapter illustrate the concepts introduced therein, and full solutions are given.

I should like to thank Mrs. K. Martin and Mrs. M. Wallis for their patient and careful typing.

D. Woodhouse,
February, 1975.

CONTENTS

Chapter 1:	High-level languages	1
Chapter 2:	Language structure of ALGOL 60	3
Chapter 3:	Statements: the sentences of the language	11
Chapter 4:	Standard functions	19
Chapter 5:	Input and Output	21
Chapter 6:	Arrays	31
Chapter 7:	For and while statements	34
Chapter 8:	Blocks and block structure	38
Chapter 9:	Procedures	42
Chapter 10:	String variables	60
Chapter 11:	Own variables and switches	64
Chapter 12:	Running and debugging	67
Bibliography		70
Solutions to Exercises		71
Appendix 1:	Backus Normal Form	86
Appendix 2:	ALGOL-like languages	88
Appendix 3.	Error messages	89
Appendix 4:	ASCII Character Codes	93
Index		95

CHAPTER 1

HIGH-LEVEL LANGUAGES

1.1 Compilers and Autocodes

In the early days of electronic computers (1940s and 1950s), programming was done in tedious and error-prone machine language. The obvious drawbacks of this approach, together with a widening vista of problems to be programmed for these machines, led to a new concept. This was the designing of a high-level language, together with a compiler to translate programmes in this language into machine language. The first high-level languages were called autocodes, one of the principal ones being Mercury Autocode, which has persisted in use to this day on some British machines. However, this is a language with a number of idiosyncrasies [1], and in the early 1950's, the IBM company, one of the leaders in the computer field, started designing a new language, which they called FORTRAN. This language was implemented on the IBM 704, and became operational in 1957; FORTRAN II became available in 1958, and FORTRAN IV in 1962. Certain peculiarities of FORTRAN, (such as 6-character identifiers; the first character of output being treated as a control character; etc.) stem from the particular hardware characteristics of the IBM 704.

FORTRAN, especially as initially produced, had a number of serious faults. One was the omission of any logical operators, or logical test; this defect was made good in FORTRAN IV. Another is the highly linear character of the code. The meaning of this latter will become clear later. An international committee therefore assembled to produce a specification of an algorithmic language, as opposed to a formula translator. The first version was produced in 1958, and the definitive revision in 1960 [2]. Hence the name of the language: 'ALGOL 60'. (Whenever the word 'ALGOL' is used in this text, reference to ALGOL 60 is intended.)

1.2 ALGOL 60

ALGOL has two main uses:

1. to communicate algorithms between people;

and 2. to describe algorithmic processes to a computer for execution.

It has gained such popularity in the former rôle, that more pro-

grammes have been published in ALGOL 60 than in any other computer language. In rôle 2, it found itself in direct competition with FORTRAN, which

a) had two years start (as FORTRAN II);

b) was the brainchild of the influential IBM company;

and c) had a complete, detailed system for input and output (IO).

The definition of ALGOL 60, in consistently adhering to a concept of machine independence, defined no input or output procedures at all. Consequently, each computer company had to design its own IO system. This completely destroyed programme portability. By the time some consistent and overall attempt was made at standardization, FORTRAN IV was on the scene with logical facilities, and ALGOL 60 was even further behind. However, apart from some clumsiness and forced verbosity in specifying output formats, ALGOL is pleasant and satisfying to use.

Minor updating of the language took place in 1962 and 1964. In 1968 there appeared ALGOL 68, which although based on ALGOL 60 is an entirely different language, with more ramifications than PL/1. It is not at present a serious competitor with ALGOL 60: in 1966, many more ALGOL 60 programmes were written than ALGOL 68 ones in 1974.

CHAPTER 2

LANGUAGE STRUCTURE OF ALGOL 60

2.1 Language Levels

The ALGOL report recognized three levels of language. The reference language is the definitive form; the publication language allows extensions of the reference language for ease of printing and reading (for example, exponents may be superscribed, subscripts subscribed, etc.); the hardware language for any computer will usually be a restriction of the reference language to the character set available on that machine.

2.2 The Characters of the Language

Basic or reference ALGOL, as defined in the report, and as used in most published programmes contains 116 symbols, namely:

1. Upper and lower case letters, and decimal digits (62).
2. true and false (2).
3. Delimiters: i.e. symbols which are not operands.

   (a) Operators: (i) $+ \quad - \quad \times \quad / \quad \uparrow \quad \div$ (6)

   (ii) $> \quad \geq \quad = \quad < \quad \leq \quad \neq$ (6)

   (iii) $\vee \quad \wedge \quad \equiv \quad \supset \quad \neg$ (5)

   (iv) goto if then else for do (6)

   (b) Separators: , . : ; := ␣ step until while comment $_{10}$ (11)

   (c) Brackets: ‘ ’ ( ) [ ] begin end (8)

   (d) Declarators: own Boolean integer real array procedure switch label string value (10)

True and false correspond to the FORTRAN logical values .TRUE. and .FALSE., respectively. In FORTRAN, these are 'compound symbols' containing 6 and 7 characters respectively. In ALGOL, they are regarded as two single characters or symbols. To indicate this, they are usually printed in heavy type, or underlined. A similar observation applies to goto, begin, end, label, etc..

Since most punching machines contain less than 116 symbols, any given hardware representation of the language will usually have a slightly different character set. The DECSystem 10 set is as follows:

1. 36 alphanumeric characters, that is uppercase letters and digits. (Lower case letters are allowed if they are available on the line printer. Both cases will be used in examples in this manual.)

2. TRUE and FALSE or 'TRUE' and 'FALSE'. An obvious economy on the basic character set is to write out all the multi-letter characters as the corresponding string of letters. This introduces a difficulty which we may exemplify from FORTRAN. In FORTRAN, an identifier may be any alphanumeric string of length ≤ 6, beginning with a letter. But 'DO' satisfies this. Therefore, 'DO' can be an identifier (i.e. a variable-name). But then, how does one (or, more importantly, the compiler) distinguish between its use as an identifier, and as a control element in the 'DO' statement? The answer is by the context. Consider, for example,

   DO 17 I = 2,K

   If DO here were an identifier, it would be on the left hand side of and assignment statement. Thus, on recognizing 'D','O', the compiler considers two possibilities, namely a DO statement or an assignment statement. Suppose that, for sheer perversity, you had an integer variable DO17I. Then, since the compiler ignores spaces, the two interpretations would be possible up to 'DO 17 I = 2'. But as soon as the next character was read in, it would be clear that a DO statement was intended, and it would be so interpreted. (Of course, it may be that you 'intended' 'DO17I = 2*K', and mispunched , for *; but in that case the resulting confusion serves you right! When I say that uses of special words as identifiers can be recognized by context, this is the case in a correct programme.)

   To return to ALGOL. Since it was designed under the assumption that, for example, <u>true</u> would be a single symbol, allowing the string TRUE to be used as both the logical value and an arbitrary variable name, would cause confusion. PDP-10 ALGOL therefore offers two alternative solutions:

   i) you have free choice of variable names provided all delimiter words are written in quotation marks, for example, <u>true</u> as 'TRUE';

   ii) delimiter words are written without quotation marks and these words are <u>reserved</u>, i.e. may not be used as identifiers. They

must also have at least one space character on each side of them.

ii) is the default option; i) may be selected by recompiling the standard compiler and using a special switch option.

3. (a) i) + - * / ↑ DIV.

↑ denotes exponentiation; * denotes multiplication.
/ denotes division, which produces a <u>real</u> result. For example, 7/3 = 1·333....3 .
DIV can only take integer operands, and has the same effect as integer division in FORTRAN, namely truncation (towards zero). For example

7 DIV 3 = 2
-7 DIV 3 = -2

and, in general,

-((-n) DIV (-m)) = (-n) DIV m = n DIV (-m) = -(n DIV m).

(A bug: sometimes, when m is a positive power of 2 and n is negative, n DIV m is evaluated to -((-n) DIV m) - 1, rather than -((-n) DIV m).)
The PDP10 compiler also provides REM: 7 REM 3 = 1.
In general, n REM m = n - (n DIV m)*m, so that
-(-n) REM m = n REM(-m) = n REM m = -((-n) REM (-m)).

ii) > >= = < <= # . The = sign is reserved for indicating the relationship of equality: 'Y = 3' asserts that Y has the value 3; it does <u>not</u> change the value of Y to 3 or anything else.

iii) OR AND EQV IMP NOT: all reserved words.

iv) GOTO or 'GOTO'; etc. (similar comments apply here as for TRUE and FALSE).

(b) As in the reference language except for the following. The single symbol ':=' becomes ':' followed by '='; '␣' becomes ' '; <u>comment</u> becomes COMMENT (or 'COMMENT', as above) or '!'; '$_{10}$' becomes '@' or '&'.

The last-named corresponds to the 'E' in FORTRAN real constants and indicates the decimal exponent. For example,
1·7&-3 = ·0017; 252@15 = $252*10^{15}$.

(c) ‘ and ’ both become ".

(d) The declarators are treated like the other multi-letter symbols.

2.3 The Words of the Language

2.3.1. Variable names or identifiers are alphanumeric strings of length ≤ 64, beginning with a letter. As compared to normal algebraic form, in which single-letter identifiers are used almost exclusively, this multi-character facility permits the use of variable names with mnemonic significance, such as RATIO, TOLERANCE, ABC123, etc.. Care should be taken not to select names which are too long, since it will be tedious to have to write the whole of a long name on several occasions; or too cryptic or frivolous (such as FRED or JACK), as their mnemonic association will soon be forgotten; or too similar (such as MARK and MASK) as confusion can easily occur. Sometimes the mnemonic is best indicated by two words, but spaces are not allowed in variable names. To overcome this, the legibility symbol '.' is allowed between letters, and is ignored. For example (unlike COBOL)

COUNT.DOWN = COUNTDOWN.

Avoid single letter identifiers, especially the letter O(Oh). Distinguish carefully between O(Oh) and 0(zero), and between I and 1.

Unlike in FORTRAN, all identifiers must have their type declared to be Boolean, integer, real, long real or string; thus:

integer X, Y, ZING; real LOOP, DISC; long real T

(The space must be present in long real.) The declarations must appear at the beginning of the programme (or block: see Chapter 8) before any executable statements.

This seems a chore, but has the advantage that a misprinted identifier is likely to show itself as a compiler error ('undeclared identifier') unless it somehow makes sense - for example, by being the same as another, declared, identifier.

2.3.2. Numeric constants are as in FORTRAN: integer, real or long real. An integer constant or variable I must have a value in the range $-2^{35} \leq I \leq 2^{35} - 1$. A real constant or variable X must be such that X = 0 or $1{\cdot}4\&-39 \leq |X| \leq 1{\cdot}7\&38$, and has a significance of about 8½ decimal digits. If $0 < |X| < 1{\cdot}4\&-39$, then X is replaced by zero. It may be written in fixed or floating point form. In floating point form, if no decimal part precedes the & or @, a value of 1 is assumed. (Thus, &3 = @3 = 1000.) Long real constants are formed by writing a real constant in floating point form, but

replacing the & or @ by && or @@. *Long real* constants and variables are held to a precision of 17 decimal digits and the same range is available as for reals (except that if $0 < |X| <$ 3&&-30, then X is represented to single precision, only).

2.3.3. *Octal* constants consist of the symbol % followed by up to 12 significant digits, which are considered to be right justified. For example, %37 denotes decimal 31. Octal constants may be used only in Boolean expressions. The words TRUE and FALSE are called *Boolean* constants, and are equivalent to the octal constants %777777777777 and %000000000000 respectively.

2.3.4. Up to five ASCII symbols (see Appendix 4) may be packed, right-justified, to give an integer-type constant. An ASCII constant comprises the symbol $, some character c, up to 5 ASCII symbols other than c, then another c. For example, $ A (in which c = the space character), or $.STOP. (in which c = .). These two constants have the octal values

000000 000101 and 000235 223720

respectively, since the ASCII values are packed thus

0 0000000 0000000 0000000 0000000 1000001

and

0 0000000 1010011 1010100 1001111 1010000

2.3.5. *String* constants are strings of symbols enclosed within quotation marks, thus: "LAMB CHOPS". For uniqueness of interpretation, the semi-colon and quotation mark may not appear alone in a string constant. The string LAMB;CHOPS must be written as "LAMB;;CHOPS" with the semi-colon duplicated; quotation marks must be similarly duplicated if they are required in a string constant. This has an intriguing effect if a desired string begins and ends with quotation marks. It is represented as a string constant thus:

" " " IS ANYONE HOME? " " "

Brackets are used to enclose special string output control characters (see Chapter 5) and so they too must be duplicated if a single occurrence is required in a string constant:

" ALPHA[[NUM]] "

The handling of string variables is described in Chapter 10.

2.4 The Phrases of the Languages

2.4.1. Arithmetic expressions are similar to FORTRAN, except that mixed mode (integer, real, long real) is permitted, and there is no complex arithmetic. For those unfamiliar with FORTRAN, it may be observed that an ALGOL expression looks like a normal mixed arithmetic and algebraic expression, using the arithmetic operators listed above and identifiers which may contain several characters. For example,

ALPHA+B*(13·2 - D↑E)

INDEX + 15·1 * FRN

The latter is interpreted by the compiler as

INDEX + (15·1 * FRN)

and not

(INDEX + 15·1) * FRN.

This is formalized into a precedence or hierarchy of operators, which is

first:	↑
second:	/ * DIV
third:	+ - (binary and unary).

If this hierarchy does not determine the order of evaluation of adjacent operators, a left-to-right order is adopted. For example, in

A * (B - C + D) ↑ E

the hierarchy specifies that the exponentiation be carried out before the product; within the parentheses, however, the operators have the same precedence, and are therefore evaluated from left to right, thereby interpreting B-C+D as (B-C)+D and not B-(C+D). Similarly, p↑q↑r means (p↑q)↑r (i.e. p↑(q*r)) and not p↑(q↑r).

Parentheses (but not brackets) may be used to emphasize or overrule the precedence rules.

2.4.2. Boolean expressions involve Boolean identifiers, constants, and operators, octal constants and arithmetic conditions. This means that all the first three classes of operators (3(a), p. 3) may occur in a Boolean expression; for example

X*Y↑2 ≥ I-5 AND NOT B IMP FALSE.

The hierarchy is: arithmetic operators (in the order listed above)

relational operators

NOT

AND

OR

IMP

EQV.

The above expression therefore denotes

(((X*Y↑2) ≥ (I - 5)) AND (NOT B)) IMP FALSE.

Again, parentheses may be used for emphasis of amendment.

The effect of the Boolean operators is to implement the elementary logical operations which may be defined in the usual truth table form, thus:

A	B	A AND B	A OR B	A IMP B	A EQV B
F	F	F	F	T	T
F	T	F	T	T	F
T	F	F	T	F	F
T	T	T	T	T	T

These operators may be used to operate on the bits of the word. The above table holds with 0 replacing F and 1 replacing T, and the operations are performed bit by bit. Thus, if A and B have been assigned Boolean values in the normal way such that A = *false* and B = *true*, then, in the store, A holds 36 zeros, and B holds 36 ones. A AND B is then formed bit by bit by ANDing the two first bits, then the two second bits, and so on. The result is 00...0, namely *false* as expected.

If, however, A has the bit pattern $\underbrace{0 \ldots 0}_{18\ \text{bits}}\ \underbrace{1 \ldots 1}_{18\ \text{bits}}$, while B is $\underbrace{1 \ldots 1}_{18\ \text{bits}}\ \underbrace{0 \ldots 0}_{18\ \text{bits}}$, then A and B each have the Boolean value *true*, but A *and* B = $\underbrace{0 \ldots 0}_{36\ \text{bits}}$ = *false*. Again, *not* A = B = *true*, so both A and *not* A have the value *true*. Thus, to obtain the desired results from Boolean expressions, which contain any Boolean variables other than those consisting of all 1's or all 0's, always consider the effect bit by bit.

To test a particular bit in A, we may write A *and* %n where n is the octal number with a 1 in the bit position to be tested, and zeros elsewhere. The value of this expression is *true* if and only if that bit is a 1.

In A <u>and</u> B <u>and</u> C

where A, B, and C are Boolean variables, the expression is known to be <u>false</u> as soon as one of A, B and C is found to be <u>false</u>. The PDP10 ALGOL compiler does not take advantage of this, however, since it works on a bit by bit basis, rather than inspecting the whole of one variable at a time. Effectively, therefore, A, B and C are each evaluated in full, even if A is false.

2.5 <u>Exercises</u>

1. Which of the following are valid PDP10 ALGOL identifiers?

(i) NAME	(ii) NEXT.VALUE	(iii) NEXT-VALUE
(iv) INDEX2	(v) INDEX.2	(vi) TWOTHIRDS
(vii) 2THIRDS	(viii) DIV	

2. Which of the following are valid PDP10 ALGOL expressions?

(i) A + -B	(ii) A + (-B)	(iii) A+(-B)
(iv) x↑(y↑z)	(v) A <u>and</u> <u>not</u> A	(vi) A = B
(vii) NUM = $.A.	(viii) A DIV B	(ix) ADIVB
(x) A-B + 1·3&-5		

CHAPTER 3

STATEMENTS: THE SENTENCES OF THE LANGUAGE

3.1 Assignment Statements

An assignment statement has the form

identifier := expression

and the result is to evaluate the expression and assign this value as the (new) value of the identifier. For example,

```
X := Y + Z - MARK
P := Q OR R AND S
```

The types of the identifier and expression must 'match' in that <u>either</u> each is of string type; <u>or</u> each is of Boolean type; <u>or</u> each has type integer, real or long real. If the types of the two entities are different, a type conversion takes place. If I is integral and X real, the assignment

```
I := X
```

results in I being assigned a value obtained by <u>rounding</u> X to the nearest integer.

Multiple assignments are allowed

```
I := J := K := 7
```

provided all the identifiers have the same type.

Assignment statements take very little time, and you should be aware of the possibility of reducing the number of operations involved, even at the expense of increasing the number of statements. For example,

```
X1 := (-B + SQRT(B↑2 - 4*A*C))/(2*A);  }
                                        } A
X2 := (-B - SQRT(B↑2 - 4*A*C))/(2*A);  }
```

should be replaced by

```
Y := SQRT(B↑2 - 4*A*C);                 }
X2 := 2*A;                              } B
X1 := (-B+Y)/X2 ; X2 := (-B-Y)/X2 ;     }
```

which involves only one more identifier. (SQRT is a function which obtains the positive square root of the expression which follows it in parentheses: see Chapter 4.)

The operation counts are:

	:=	+	-	*	/	↑	SQRT
A	2	1	5	6	2	2	2
B	4	1	4	3	2	1	1

in which the operations are listed in the order of increasing time required for their execution. (Assignment and taking the square root are not normally referred to as operations.)

3.2 Control Statements

Labels in ALGOL are identifiers, formed under the same rules as other identifiers (so numeric labels are not allowed). Any statement may be labelled using a colon, thus

```
LAB3:  COMP := X - Z * 2;
```

An unconditional jump is achieved by a goto statement, thus:

```
goto  LAB3;
```

Do not label every statement: the programme becomes illegible. Only label those to which control may be transferred from out of the written sequence. Aim, in fact, to use as few labels as possible. You will find this easier when you have learned about for and while statements (Chapter 7).

Since the location of ALGOL statements on punched cards or teletype lines is not specified, some device is needed to indicate the end of a statement. This is the semicolon. Thus:

```
begin
       integer j,push ; boolean rep ; real x,y,zoom ;
labl:  j:= push:= 5 ; rep:= true;
       x:= y↑2 - push*zoom ;
       goto labl
end
```

is a valid (albeit useless and, indeed, unending) ALGOL programme.

Note that (i) an ALGOL programme must begin with 'begin', followed immediately by the type declarations, if any, and then any other statements, and finish with 'end'. There is no STOP statement.

(ii) begin and end are like brackets: Cf a vector (x_1,x_2,x_3): the entities within the brackets are separated by commas, but x_1 need not be preceded nor x_3 followed by a comma. Analogously, we need no ';' after begin or before end.

3.3 Conditional Statements

3.3.1. A conditional statement has the form

```
if b.e. then S1 else S2 ;
```

(where b.e. denotes a boolean expression, and S1, S2 denote statements). For example,

```
if X<Y then I := 27 else Z:= X↑2 + Y↑2;
```

The FORTRAN 'logical IF' also exists, namely

```
if b.e. then S1;
```

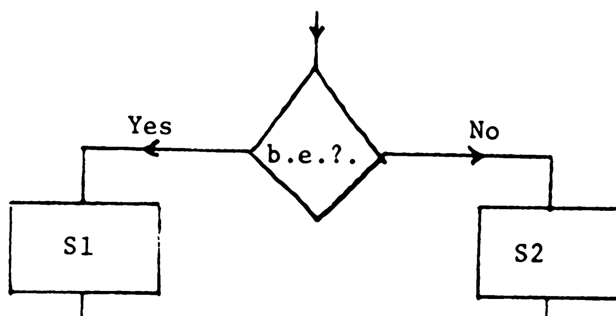
In FORTRAN this is quite straightforward:

```
      .
      .
      IF(b.e.) S1
      .
      .
```

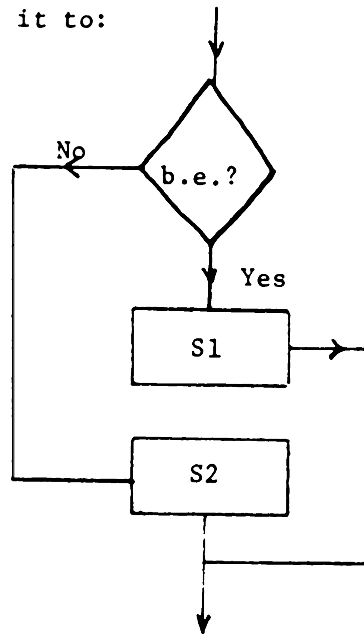
but the more general form requiring alternate execution of S1 or S2 has to be implemented in the clumsy form:

```
      .
      .
      IF(b.e.) GOTO 1
      S2
      GOTO 2
    1 S1
    2 .
      .
```

Compare this with the ALGOL conditional statement above. This is the import of the earlier claim that FORTRAN is more 'linear' than ALGOL. While ALGOL provides a direct implementation of the flow diagram:



FORTRAN converts it to:



The ALGOL form is easier to comprehend, since there are fewer labels and jumps.

S1 cannot itself be a conditional statement, as ambiguity then arises; for in

<u>if</u> b.e.1 <u>then</u> <u>if</u> b.e.2 <u>then</u> S1 <u>else</u> S2

it is not clear to which <u>if</u> ... <u>then</u> the <u>else</u> belongs.

Is it

<u>if</u> b.e.1 <u>then</u> (<u>if</u> b.e.2 <u>then</u> S1) <u>else</u> S2

or

<u>if</u> b.e.1 <u>then</u> (<u>if</u> b.e.2 <u>then</u> S1 <u>else</u> S2) ?

In the first case we have

b.e.1	b.e.2	statements executed
F	F	S2; S3
F	T	S2; S3
T	F	S3
T	T	S1; S3

while in the second

b.e.1	b.e.2	statements executed
F	F	S3
F	T	S3
T	F	S2; S3
T	T	S1; S3

(where S3 is the statement following the conditional statement). Comparison of the right hand columns of these two tables shows that the two interpretations are distinct. Clearly there is no ambiguity when the parentheses are inserted, and this is what is done, except that begin ... end are used. and not (...); thus we have

```
if b.e.1 then begin if b.e.2 then S1 end else S2;
```

and

```
if b.e.1 then begin if b.e.2 then S1 else S2 end;
```

Evaluation is from left to right, and no more than necessary is carried out. For example, if D < 0·2, then

```
if X <= ϕ then D := D*X else D:= D/X
```

will never give arithmetic overflow through division by zero, since if X = ϕ then only the first arm of the conditional is executed.

3.3.2. Conditional expressions may be defined analogously to conditional statements, replacing S1, S2 by expressions E1, E2. For example,

```
D:= if X <= ϕ then D*X else D/X
```

is an assignment statement in which the expression to the right of the assignment sign is a conditional expression. Conditional expressions, like conditional statements, may be arbitrarily complex, but use parentheses rather than begin, end to indicate groupings, for example:

```
X:= if ALPHA < BETA then (if GAMMA > ϕ then 17·5)
                    else Y↑2 + Z↑2 ;
```

However, you should not abuse this facility by constructing expressions which are too complicated, or the programme is difficult to read. For example,

```
K:= if X = 9 then (if Y = 7 then 2 else 1) else 3;
```

is legal but horrible, and should be replaced by

```
L:= if Y = 7 then 2 else 1;

K:= if X = 9 then L else 3;
```

or by

```
if X ≠ 9 then K:= 3
     else K:= if Y = 7 then 2 else 1;
```

The 'else' must be present in a conditional expression, or no value

would result for assignment to the identifier on the left when the condition fails. ALGOL has no 'arithmetic IF'.

3.4 Compound Statements

Any statement or statements may be grouped for consideration as a single entity by preceding it (them) by begin, and following it (them) by end. The resulting grouping is a single compound statement. For example, the statements S1 and/or S2 in the conditional statement above may be compound, thus:

```
if  b.e.1  then  begin  H := K;  JAMES := E*E - 5*T;
                        goto LAB4
                 end
           else  H := -H;
```

Formally,

```
Y := X;
```

is a simple statement, while

```
begin  Y := X  end;
```

is a compound statement. A block is a compound statement which contains one or more declarations.

3.5 Dummy Statement

The ALGOL dummy statement (cf. FORTRAN's 'CONTINUE') is just the empty string. As in FORTRAN it is used to locate a label. It is only needed immediately before an 'end'. For example,

```
          .
          .
          .
          JAB := 7;
    L3:
    end
```

(For and While Statements are described in Chapter 7.)

3.6 Programme Layout

The freedom of format allows statements to be packed line after line into an illegible mass, or to be spaced out in such a way as to indicate

the flow of the programme. The latter is aided by aligning each end under its corresponding begin and each else under its corresponding then, unless the entities are small enough to go on one line. Thus

```
          if  Y = 5  then  goto  lab3  else  Y := -Y;
```

but

```
          if  Y = 5  then  Z := X↑2 - A/B + 3*FIRST-LAST
                     else  X := EPSILON - A25 + 16·3&14;
```

and

```
          if  Y = 5  then  begin  X := 6;  goto L end;
```

but

```
          if  Y = 5  then  begin  X := 6;  b := 42 - a;
                                  Z := COMP-MARKSUM;
                                  goto  LINE27
                           end
```

The begin, end groupings may be nested, and should, within reason, be successively indented, thus

```
begin   .
        .
        .
     begin   .
             .
             .
     end
        .
        .
        .
     begin   .
             .
             .
          begin
             .
             .
             .
          end
     end
end
```

Otherwise, each first statement on a line should be aligned with the adjacent first statements, and not too many statements should be packed on each line.

3.7 Comments

It is useful to be able to include statements explaining, to the

human reader of the programme listing, the meaning of variables and the function of groups of statements. This sort of explanation may be inserted by use of the reserved word <u>comment</u> (or the symbol '!', but this is not standard ALGOL). The three possible ways of inserting explanations or comments are:

(i) ...; <u>comment</u> $\underbrace{\ldots}$; ...

(ii) ...; <u>begin</u> <u>comment</u> $\underbrace{\ldots}$; ...

(iii) ... <u>end</u> $\underbrace{\ldots}$; ...

In each case the brace indicates the comment string. In (i) and (ii) this string should contain no semicolon; in (iii) it may contain only letters, digits and spaces, and no reserved words.

Comments should be used freely and fully to enable programmes to be understood by others (and by yourself at some future date). They should indicate the aim, method and symbolism of a programme.

1.8 Exercises

1. Write an ALGOL programme to find the greatest common divisor of two integers.
2. Obtain the sum of the cubes of the first 100 positive integers.
3. Integrate $1/(1 + x^2)$ with respect to x over the range x = 0 to x = 1, (a) by the trapezium method, and (b) by Simpson's rule; using 20 strips in each case, and comparing the results.

CHAPTER 4

STANDARD FUNCTIONS

4.1 Standard Functions

A number of functions are provided in the ALGOL library, and may be used whenever the value they provide is required.

ENTIER(X) is defined if X is real or long real, and then has the largest integer value not greater than X. For example, if X = 3·7 and Y = -5·2, then

```
I := 5 + ENTIER(X);  J := 3*ENTIER(Y) ;
```

has the effect of setting I equal to 8 and J to -18.

The other arithmetic functions are as follows.

$$ABS(X) = \begin{cases} X & \text{if } X \geq \phi \\ -X & \text{if } X < \phi \end{cases}$$

X may be real, long real or integer, and ABS(X) has the same type as X.

ABS should usually be used when testing real and long real variables. It is quite possible for variables of these types to differ from the theoretical value by some units in the eighth (or seventeenth) significant figure, due to roundoff error. (The difference may be much more, if the process is unstable.) Therefore, to test whether A is zero, use not

<u>if</u> A = 0 <u>then</u> ...

but

<u>if</u> ABS(A) < EPS <u>then</u> ...

where EPS is a small positive number chosen appropriately for the process in hand.

SIGN(X) takes the same range of argument types as ABS but always produces an integer result, thus

$$SIGN(X) = \begin{cases} 1 & \text{if } X > \phi \\ \emptyset & \text{if } X = \phi \\ -1 & \text{if } X < \phi \end{cases}$$

SIN, COS, ARCTAN, SQRT, EXP, LN, TAN, ARCSIN, ARCCOS, SINH, COSH, and TANH each have a single real argument, and produce the obvious result, of real type.

LSIN, LCOS, LARCTAN, LSQRT, LEXP and LLN each take one long real argument and produce a long real result.

IMIN, IMAX, RMIN, RMAX, LMIN and LMAX each take up to 10 arguments, of the type indicated by the initial letter in each case. The result is of the same type and is the maximum or minimum value of the arguments. For example,

```
Y := IMAX(I,J,K) + SQRT(COS(X↑2 - A*B));
```

is a valid assignment statement. Other library facilities will be described later. They are called procedures (see Chapter 9).

The library procedure names are not reserved words, but it is sensible to treat them as such unless the user is re-defining a procedure to do the same task. For example, if he is defining a procedure to calculate sin x, it would be sensible to use the name SIN, but not otherwise.

4.2 Exercises

1. Use the Newton-Raphson iteration ($x_{n+1} = (x_n + A/x_n)/2$, $n = 0,1,2,\ldots$) to obtain the square root of a number A, with an error at most EPS.
2. Calculate the area of a planar triangle, given the co-ordinates of its vertices.
3. Obtain the polar co-ordinates of the sum of two complex numbers specified in polar form, and the cartesian form of the product of two complex numbers which are specified in cartesian form.

CHAPTER 5

INPUT AND OUTPUT

5.1 Peripheral Management

5.1.1. Device Allocation

A user programme may handle up to 16 peripheral devices simultaneously. These are known to the programme by the logical number (0-15) of the channel on which the device operates. A peripheral device is allocated to the programme by a call to one of the library procedures INPUT and OUPUT. So, for example to allocate the card reader for input on channel 5, the following statement is used:

```
INPUT(5, "CDR");
```

The choice of channel number is completely under the control of the programmer, provided that only one device is allocated to any one channel at a time. Subsequent to the above call to INPUT, any reference to channel 5 refers to input from the card reader.

```
INPUT(10, "DSK");
```

allocates the disc for input (from the disc to the programme); output to the disc must take place on a different channel; for example:

```
OUTPUT(11, "DSK");
```

The only exception to this rule is the teletype if the programmer is on-line. Thus, either

```
INPUT(2, "TTY")
```

or

```
OUTPUT(2, "TTY")
```

will allocate the teletype for input and output.

INPUT and OUTPUT may have 2, 3 or 4 arguments, as follows

$$\begin{Bmatrix} \text{INPUT} \\ \text{OUTPUT} \end{Bmatrix} \text{(chan, str, mode, buff).}$$

If there are n arguments, they are assumed to be the first n of these 4 (n = 2 or 3).

chan is the channel number as already described; $0 \leq \text{chan} \leq 15$.

str is a string constant or variable whose value is one of the following with the significance indicated:

DSK	disc	LPT	line printer
DTA	DECtape	PTR	paper tape reader
MTA	magnetic tape	PTP	paper tape punch
CDR	card reader	PLT	plotter
CDP	card punch	TTY	teletype

If s is a string variable (see Chapter 10), then subsequent to the assignment

s := "CDR" ;

INPUT(5, "CDR") and INPUT(5, s) are identical in effect.

mode indicates the form of the data to be transferred along the channel. Some possible values are:

ϕ : ASCII code. 7-bit bytes, packed left-justified, 5 characters per word. This would be the normal mode, representing readable text.

8 : Image mode. This is device dependent and uses 36-bit bytes. The buffer is filled with data exactly as supplied by the device.

11 : Image binary mode, used for the storage of binary data on a disc. Again, 36-bit bytes are used.

If this parameter is absent, ASCII mode is assumed.

buff specifies the number of buffers to be allocated for the device. The default is two, unless str = "TTY" when four are allocated, two for each of input and output.

The procedure RELEASE is used to release a device from a channel. Thus

RELEASE(5) ;

releases channel 5 and leaves it free for re-allocation. If an attempt is made to allocate a device to a channel which is already in use, an automatic RELEASE is performed and the new device allocated.

5.1.2. Channel Selection

Only one of the currently allocated channels may be open for input and one for output at any one time. A channel is opened (selected) as follows:

SELECTINPUT(chan1) ; SELECTOUTPUT(chan2) ;

Following these calls, all subsequent input is taken from the device on

channel chan1 and output sent to the device on channel chan2, until another channel or channels are selected. Such a change may be made at any time. An implicit or explicit RELEASE automatically deselects the channel, if necessary.

If input is called for when no input channel is selected or allocated (or both), then the teletype is allocated and selected by default; and similarly for output.

5.1.3. File Specification.

Disc and DECtape require the specification of a file as the source or destination of the data. This is achieved by

```
OPENFILE(chan, str);
```

where chan is the channel number and str is or contains a file name, for example

```
OPENFILE(7, "EXAMPL.FOR") ;
```

When this file is finished with, it should be closed by the call

```
CLOSEFILE(7);
```

An implicit or explicit RELEASE performs a CLOSEFILE if necessary. The end of the programme releases the devices on all channels.

5.1.4. Summary

To perform input one must:

i) allocate the desired device to a channel, e.g. INPUT(5, "DSK".);

ii) select the channel, e.g. SELECTINPUT(5) ;

iii) open a file (on a file device), e.g. OPENFILE(5, "EX.DAT");

and later one should

iv) close the file, e.g. CLOSEFILE(5) ;

and v) release the channel, e.g. RELEASE(5).

The procedure for output is similar.

5.2 Numeric Input and Output

5.2.1. READ(A,B,...,K); is used to input numeric data on the channel currently selected for input. READ may have up to ten arguments, of type integer, real, long real, Boolean (or string). The data may be written in any sensible form. No spaces or newline characters should be used within items of data, but should be reserved as separators and terminators. Terminate the data with one of these characters, and not just the end of the file.

Automatic data conversion takes place, a number for assignment to a Boolean variable being read as an integer. Thus, zero gives the value false, while any non-zero value gives true.

For a string variable, a " is sought, and the subsequent text up to, but not including, the next " is read. If this " is immediately followed by another quotation mark, a single occurrence of " is stored and reading continues; otherwise it ceases and the string stored as the value of the variable. For example,

```
string S ;
   .
   .
   .
read(S) ;
   .
   .
   .
```

together with the data input

"THE SHIP IS CALLED " "ALNWICK CASTLE" " "

assigns as the value of the string variable S the string

THE SHIP IS CALLED "ALNWICK CASTLE"

5.2.2. PRINT(id, intpt, decpt); is used to output numeric data on the channel currently selected for output. id is the name of the variable whose value is to be printed; it may be of type integer, real, long real or Boolean. intpt and decpt, if present, are non-negative integers, with the following interpretation.

i) intpt > 0, decpt ≥ 0 indicates fixed point printing with intpt places before the decimal point (usually: see below) and decpt places after it. The sign ('-' or a blank) is allocated a single space.
Examples:

If X = 356·37 then

PRINT(X,5,3) produces ␣␣␣356.370

(where ␣ denotes the space symbol).

PRINT(X,3,1) produces ␣356.4

(with rounding, not truncation).

PRINT(X,1,5) produces ␣356.37000

(All 3 digits before the point are output, even though intpt = 1. This is the exceptional circumstance producing more than intpt digits before the decimal point.)

PRINT(X,3,0) produces ␣356

(Note suppression of the decimal point.)

If X = -231.785 then

PRINT(X,3,2) produces - 231.78

If X = -231.775, then

PRINT(X,3,2) produces - 231.77

(Note '5' is always rounded down.)

PRINT(X,5,2) produces ␣␣- 231.77
PRINT(X,1,3) produces - 231.775

Thus, intpt + decpt + 2 symbols are (usually) produced, unless decpt = 0, when intpt + 1 symbols are produced. decpt = 0 is the normal value for integer and Boolean variables.

ii) intpt = 0, decpt > 0 indicates floating point printing, with decpt + 1 significant digits output. decpt + 7 or 8 symbols are produced (usually: see below), depending on whether one & (integer or real) or two &s (long real) are produced. Two digit spaces and a sign space are allowed for the exponent.

Examples:

If real X = 356.37, then

PRINT(X,0,3) produces ␣3.564&␣␣2
PRINT(X,0,5) produces ␣3.56370&␣␣2

If real X = 10^{-30}, then

PRINT(X,0,2) produces ␣1.00&-30

If real X = 2.732, then

PRINT(X,0,3) produces ␣2.732

(Note the suppression of the whole exponent part because the value of the exponent is zero.)

Integers are treated the same way as reals.
If long real X = -0.0025, then

PRINT(X,0,2) produces -2.50&& ␣ -3

iii) Abbreviations are treated as follows.

If X is real, PRINT(X,m) means PRINT(X,0,m);
PRINT(X) or PRINT(X,0,0) means PRINT(X,0,7).

If L is long real, PRINT(L,m) means PRINT(L,0,m);
PRINT(L) or PRINT(L,0,0) means PRINT(L,0,15).

If I is integer or Boolean,
PRINT(I,m) means PRINT(I,m,0);
PRINT(I) or PRINT(I,0,0) means PRINT(I,11,0).

5.3 Output Control

SPACE(n); TAB(n); PAGE(n); NEWLINE(n);

where n is an integer expression, cause n spaces, tabs, pages or newlines, respectively, to be output. SPACE(1) may be abbreviated to SPACE, etc..

Always plan your use of output statements so as to produce an intelligible layout of the results.

5.4 Text Output

5.4.1. This is effected by a call to the procedure WRITE which takes a single string argument.
Thus,

```
WRITE("RESULTS ARE:");
```

produces the output

```
RESULTS ARE:
```

as does the sequence

```
S:= "RESULTS ARE:" ;
WRITE(S) ;
```

Often one wishes to output a number of lines of text, and it is inconvenient to keep closing the WRITE parenthesis, insert NEWLINE(n), return to a WRITE, and so on. An example of this could be:

```
SPACE(15); WRITE("RESULTS"); NEWLINE(2); SPACE; WRITE
("COEFFICIENTS"); SPACE(5); WRITE("VALUE"); NEWLINE;
```

Control characters are provided which may be included in the WRITE

string argument, enclosed in brackets (cf. §2.3.5). These are

P	: new page	: equivalent	to	PAGE(1)
C or N	: new line	: "	"	NEWLINE(1)
T	: tab	: "	"	TAB(1)
S	: space	: "	"	SPACE(1)
B	: breakoutput	: "	"	BREAKOUTPUT

Using these, the following achieves an identical layout to the above:

```
WRITE("[15S]RESULTS[2NS]COEFFICIENTS[5S]VALUE[N]") ;
```

Recall that an actual occurrence of a bracket must be duplicated, so, for example, to obtain the following output

```
X[I]=
```

one must write

```
WRITE("X[[I]]=") ;
```

or its equivalent; while to obtain

```
"X[I]"
```

requires

```
WRITE("""X[[I]]""");
```

5.4.2. Teletype Handling

When the user is on-line and the teletype is being used for both input and output (either explicitly or by default) then care must be taken over the interleaving of input and output statements. Output for a device is put by the computer in the small area of reserved storage space known as the output buffer for that device. This output is normally sent to the device only when the buffer is full. If at some stage of the programme you want all the output to date to appear, regardless of the current buffer state, you use the statement BREAKOUTPUT. For example:

```
WRITE("HEADING[N]");
BREAKOUTPUT;
READ(X,Y,Z);
```

or, more briefly,

```
WRITE("HEADING[NB]");
READ(X,Y,Z);
```

This ensures that the word 'HEADING' is displayed and the teletype moves to the next line to accept the values of X, Y, Z. Otherwise, the programme

could be waiting for you to input X, Y and Z, with you unaware of this since the word 'HEADING' is still in the buffer, and hence not yet seen by you. You would therefore believe that execution had not yet reached this point: each of the user and the programme would be waiting for the other.

The use of breakoutput is unnecessary when the input and output media are different.

5.5 Byte Handling

A byte is a number of bits, typically 6, 7, 8 or 9. In PDP10 ALGOL, facilities exist for handling bytes of any size from 1 to 36 bits. The ASCII code (Appendix 4) represents characters by 7-bit bytes. The statement

insymbol(J);

where J is an integer identifier, causes the next byte (ASCII character) to be read from the currently selected input channel, and its value assigned to J. Similarly

outsymbol(n);

where n is an integer expression, causes the value of n to be output as a byte (ASCII character). Since ASCII constants are held in integral form, they may appear as arguments in outsymbol. For example,

outsymbol($.*.)

produces an asterisk on the current output channel.

Examples

1. <u>integer</u> i; i:= $.A. ; print(i) ;
   gives '65' on the output stream

2. <u>integer</u> i; i:= $.ABC. ; outsymbol(i) ;
   produces 'C' as output.

3. <u>integer</u> i; i:= $.ABC. ; outsymbol (i DIV 2↑14) ;
   outsymbol(i DIV 2↑7) ; outsymbol (i) ;
   produces 'ABC' as output.

nextsymbol(J); acts like insymbol(J); but does not advance the byte pointer. skipsymbol; simply advances the pointer.

These procedures are most useful for scanning input streams and assigning input values to string variables as a result of this (see Chapter 10).

5.6 End of File

The Boolean procedure IOCHAN may be used to check whether the end-of-file has been encountered on any peripheral device. Its argument is the number of the channel on which the device is allocated, and bit 29 is set if and only if the end-of-file is encountered. The Boolean expression

IOCHAN(chan) and %100

is therefore true if and only if the end-of-file has been reached on channel number chan.

To read all the bytes in a file on channel 3 into a string S, we could write:

```
            L:= 0;
      LAB:  if not IOCHAN(3) and %100  then
                     begin L:= L+1; insymbol (S.[L]); goto LAB end;
```

(See Chapter 10 for a description of strings and bytes.)

The 'obvious' Test

```
if not (IOCHAN(3) and %100) then ...
```

does not work (cf. section 2.4.2), as this Boolean expression has 1's in every bit position (except the 29th) for all possible values of IOCHAN(3). Thus, it is always true, the end of the file is overrun, and the run-time error indication

'ATTEMPT TO READ OR WRITE OVER END-OF-FILE'

is given. Check the two expressions for all possible bit combinations to see exactly what is happening. The best form of the test uses the while statement (section 7.2.3):

```
for L:= 1, L+1 while not IOCHAN(3) and %100 do insymbol(S.[L]);
```

5.7 Validation of Data

In its final form, a programme should have a data-checking procedure or section. The form of input data is specified in the programme documentation, but errors will occur in the preparation of data. At the least, these may cause the breakdown of a production run; alternatively spurious results may be produced, or some serious breakdown of the system caused. The programme itself should therefore check that the data is in the form and of the type it expects and can handle.

5.8 Conclusion

Input and output in PDP10 ALGOL requires more programme text than in FORTRAN - there is no economization to parallel the FORMAT statement with its full capabilities. However, one advantage of ALGOL output is the ability to vary the format dynamically (i.e. as a result of values calculated during programme execution), for example by means of the 'integer expression' in SPACE (integer expression), etc..

5.9 Exercises

Add input and output instructions to the solutions to the exercises of Chapters 3 and 4, where indicated by the comments in those programmes.

CHAPTER 6

ARRAYS

6.1 Arrays

To represent and handle vectors, matrices, etc., we may use an array. This is the means by which ALGOL implements the concept of subscripting a variable. Because the writing of small numbers and symbols, below the level of the others, is inconvenient for computer input or output, array subscripts are enclosed in brackets and separated by commas. So, for example, the vector elements x_1,x_2 and x_3 may be represented by the ALGOL variables X[1], X[2] and X[3]; while the matrix element a_{ij} may become A[I,J]. X, A are array names. An array name is an identifier of the usual form, and each array element may be selected by specifying the appropriate subscripts. Subscripts may be arbitrarily complicated expressions. These are rounded to integer values to select an array element. For example,

HEAD[J, K↑2-3, Z-SIN(X)]

is an element of the three-dimensional array HEAD.

Array elements may be of any type, provided that all elements of any one array are of the same type. They must be declared along with the simple variables.

integer array HAND[-3:10];

declares a one-dimensional array HAND, with integer elements, and with subscripts -3, -2,...,10. This means that 14 successive store locations are reserved, and are named HAND[-3], HAND[-2],...,HAND[10]. Subsequent to this declaration, HAND[J] may be used anywhere that an integer variable is permitted. J is not checked to see whether $-3 \leq J \leq 10$, and chaos can ensue if it is outside these bounds.

The directive CHECKON 1, placed in the programme as a statement, causes all subsequent array bounds to be checked, to the end of the programme, or until a directive CHECKOFF 1 is encountered. With the check facility on, an out-of-bounds subscript causes the programme to cease execution with the error indication ILL MEM REF (illegal memory reference). Use of this facility causes the programme to run more slowly, so it should normally be reserved for debugging runs.

In general, an array declaration is of the form

type array ARRNAME[ℓ_1:u_1, ℓ_2:u_2,..., ℓ_n:u_n].

This declares an n-dimensional array, whose i^{th} subscript may take the values ℓ_i, $\ell_i + 1, \ldots, u_i$ for $i = 1,2,\ldots,n$. The ℓ_i are the lower bounds and the u_i the upper bounds on the subscripts. They must satisfy $\ell_i \leq u_i$ for all i; there is no restriction on the size of n. If the type is omitted, it is assumed to be real.

Obvious abbreviations are permitted in array declarations. For example

array finger, toe [-7:1, 1:5], BOOK[1:10, 1:10, 1:10]

declares 3 real arrays: two two-dimensional ones, finger and toe, each 9 × 5, and BOOK of 3 dimensions, 10 × 10 × 10.

Division by an array element should be avoided, by first making an assignment to a simple variable. Use

```
Z:= A[I] ;
Y:= X/Z ;
```

in place of

```
Y:= X/A[I].
```

6.2 Library Procedures

Three integer procedures, DIM, LB and UB are available for use with arrays.

DIM(ARR) is the number of dimensions of array ARR;

LB(ARR,N), UB(ARR,N) is the {lower / upper} bound of the Nth subscript of array ARR.

6.3 Exercises

1. The ancient Greeks approximated $\sqrt{2}$ by the sequence $\{p_n/q_n\}$, where $p_1 = q_1 = 1$, and

$$p_{n+1} = p_n + 2q_n \ , \ q_{n+1} = p_n + q_n \ , \quad n = 1,2,\ldots$$

(that is, the continued fraction expansion). Print out the first 20 values of p_n, q_n and p_n/q_n, and compare them with the first 20 approximations obtained by using the Newton-Raphson method (Exercise 4.2.1) with $x_1 = 1$.

2. Input data in the form

integer1 integer2 real number

is to be interpreted as indicating that some quantity of a certain product (integer1) has been sold at a particular location (integer2) and the real number gives the value of the sale. The terminal input record contains a single zero. Calculate the total value of sales by product for each point of sale.

CHAPTER 7

FOR AND WHILE STATEMENTS

7.1 Repetition

The ALGOL code for producing the output for Exercise 6.3.2 is very clumsy (page 74).

(i) The statements

```
SPACE(3);write(val[i,loc]);
```

are repeated successively, the only difference between each occurrence being the value of i(= 1,2,3).

(ii) The segment of programme from again: to the end is repeated successively for varying values of loc. This necessitates an initialization (loc:=1;), an incrementation (loc:= loc + 1), and a test (if loc ...).

Such repetitive situations occur so frequently, especially in contexts in which arrays are useful, that special statements exist to implement them. These are the for and while statements, which are the ALGOL equivalent of the FORTRAN DO-loop. They are used to perform a (possibly compound) statement repeatedly, either a specified number of times or until a specified condition is satisfied.

7.2 Forms of the Statements

7.2.1 Form 1.

```
for id := a.e.1 step a.e.2 until a.e.3 do s;
```

where id is an identifier - the control variable - a.e.i is an arithmetic expression, i = 1,2,3, and s is a statement. The effect of this is as follows:

```
        id := a.e.1
L1:     if (a.e.3 - id) * a.e.2 < 0 then goto L2;
        s ;
        id := id + a.e.2 ;
        goto L1
L2:     .
        .
        .
```

Since the test is made before execution, it may be that s is never

executed. Since id is updated before testing then id ≠ a.e.3 when the for loop has been completed in the normal manner.

```
        for  I :=  1 step 2 until 9 do s;
```

executes the statement s for I = 1,3,5,7,9, unless this is affected by assignment of values to I within s. If s is

```
        begin  J := 7*I ; I := 10 end
```

then comparison with the above expansion of the for statement shows that s will be executed only once. Not only id but also any a.e.i may be changed by s, but great caution should be exercised.

```
    real  R,S ; integer I ; array A[1:10] ;
      .
      .
      .
    for  A[J] := 2 step R↑2  until  SIN(s)*10  do ... ;
```

If 'step a.e.2' is omitted, 'step 1' is assumed.

7.2.2 Form 2.

```
        for  id :=  a.e.1,  a.e.2,..., a.e.n  do  s;
```

This is equivalent to

```
                id := a.e.1 ;
                 s ;
                id := a.e.2 ;
                 s ;
                 .
                 .
                 .
                id := a.e.n ;
                 s ;
```

and so s is executed exactly n times. This is useful when we wish to have s executed for a few unrelated values.

7.2.3 Form 3.

Often we do not know exactly how many iterations of s we need, but know some condition which is to be satisfied. This is catered for by

```
        for  id := a.e.  while  b.e.  do  s ;
```

which is equivalent to

```
      L1:    id := a.e.
             if  NOT  b.e.  then  goto  L2 ;
              s ;  goto  L1 ;
      L2:     .
              .
              .
```

In this case, s <u>must</u> change b.e., and will probably change a.e. also. For example:

```
E := 1 ; I := 1 ;
for J := 3*I while E > 1&-4 do
   begin I := J↑2 ;
         A [I] := Ø ;
         E := 1/I
   end;
```

would set A[9] = A[729] = A[6561] = Ø .

Forms 1, 2 and 3 may be combined, for example:

```
for I := 1 step 2 until 9, 13, 21,X↑2 while Z > 2, J*J do s ;
```

Here s is performed for I = 1,3,5,7,9,13 and 21; then repeatedly for I = X↑2 until Z ≤ 2; and then for I = J*J(provided I is not changed in s).

7.2.4 Form 4.

```
while b.e. do s ;
```

For example,

```
T := (1 + X)/2 ; E := 1&5
while E > 1&-5 do begin s := T ;
                        T := (s + X/s)/2 ;
                        E := ABS(s - X/s)
                  end ;
```

finds the square root, T, of X, to within 10^{-5} (cf. the solution to Exercise 4.2.1).

7.2.5 Forms 3 and 4 are useful in reducing the number of labels used. Such a reduction should be the aim of every programmer, since the more transfers of control a programme contains, the harder it is to follow. There are times, of course, when a very intricate piece of programming is needed to avoid the use of a label. Your aim, therefore, should not necessarily be to use <u>no</u> labels, but to use as few as possible. Always think, before inserting a label, "Do I really need it, or would a <u>while</u> statement or conditional statement do the job better?".

7.3 Nesting

The statement s is called the range of the for statement. This may be any statement, including a conditional one. One may transfer control out of the range of a for statement, but not in, as this bypasses the initialization procedure. For statements may be nested:

```
for i := 1 until IMAX do
    for j := 1 until JMAX do s ;
```

The output for Exercise 6.3.2 may be produced thus:

```
for loc := 1 until 5 do
    begin write(loc, 4) ;
          for prod := 1 until 3 do
              begin space(3) : write(val[prod, loc]) end ;
          newline
    end
```

A for statement is not classed as an unconditional statement, and so may appear after then in an if-then conditional, but not an if-then-else conditional.

7.4 Exercises

1. Re-write the general solution to Exercise 3.8.2, using a while statement (cf. Exercises 5.9).

2. Calculate the number of possible captures for either side in a given position in the game of draughts (checkers). Input must specify which side, and the state of the board.

3. Two judges place ten candidates in order in a beauty contest. Find Kendall's Rank Correlation Coefficient, which measures the degree of agreement, and depends on the number of agreements and disagreements between the judges over the ordering of the 45 pairs to be found among ten candidates. It is defined as

   (number of agreements - number of disagreements)/45.

4. Test a given number for primality.

CHAPTER 8

BLOCKS AND BLOCK STRUCTURE

8.1 Blocks

Definition: A block is a compound statement with at least one type declaration after begin; for example,

```
begin integer i ; i := 7 end
```

Blocks may be thought of as the paragraphs of the language; FORTRAN has no corresponding construct.

All ALGOL programmes are blocks, and so far, all programmes used have comprised a single block. However, blocks may be nested, thus:

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

What is the point of doing this? Why not declare everything at the beginning of the programme?

8.1.1 The compiler arranges that, when a new block is encountered, space is allocated for the variables declared at the head of that block. The allocation takes place at block entry, rather than at compile time: this system is referred to as dynamic storage allocation. The space so allocated is relinquished at the end of the block. The scope of a variable is that part of the programme within which it is defined and accessible. Thus the scope of x (above) is the inner block, while the scope of i is the whole programme, including the inner block. A variable is local to the block in which it is declared, and global to any inner blocks.

Now, if blocks were simply successively nested, some point would be within the scope of all the variables, and storage space for all variables would be necessary at that one time.

```
begin integer i ;
      :
      begin real x ;
            :
            begin real z, t ;
                  :
                  begin Boolean f ;      ]
                  :                      ]
                  end ;                  ]
                  :
            end ;
            :
      end ;
      :
end
```

However, if some blocks are distinct from others, we have the following sort of situation:

```
begin integer i ;
      :
      begin real x ;
            :
            begin real z, t ;
            :
            end ;
            :
      end ;
     *
      begin Boolean b ; integer j ;
      :
      end ;
      :
end
```

Here, the storage allocated for x, z and t has been vacated by * , and so can be re-used for b and j. This economization of storage removes the need for an EQUIVALENCE feature. Obviously, a real x could be declared at * without confusion with the previous one. More than this can be said, however. Consider the following:

```
begin integer i, j, k ;
      i := 7 ;  j := i*i ;
        :
      begin integer i, n ;
      i := 4 ;  n := i*i ;
        :
      end ;
        :
      k := i*i ;
        :
end
```

The result of this is that n = 16, while j = k = 49. The reason is that the i declared in the inner block takes precedence over the i of the outer block in, and only in, the inner block. However, it does not represent the same storage space, nor does it destroy the previous i. This becomes available for use again as soon as the inner i is released (that is, at the end of the inner block).

Thus, a local variable takes precedence over a global variable of the same name. The scope of a variable is that block in which it is declared, excluding any internal blocks which have a declared variable of the same name.

Greatest efficiency of storage space and of the time taken to access variables is achieved by declaring identifiers in the innermost block in which they are required.

Any block must be entered at the beginning, or the storage allocation mechanism is by-passed.

8.1.2 The second use of the block structure is that it allows several programmers to divide the work of coding a large problem between them. Intercommunication between the blocks can then be via the identifiers declared at the beginning of the whole programme.

8.1.3 The third, and possibly the main, use of the block structure is that it enables the implementation of dynamic arrays. These are arrays in which the bounds are declared in terms of variables, not constants; for example

```
integer array HAND [1:N, I:J].
```

The variables used to denote bounds must be declared, and have a value assigned in a block outside that containing the array declaration, and not simply prior to the array declaration in the same block. This is to allow storage allocation, and the establishment of the mechanism for accessing the array.

In any block, all the declarations occur at the beginning, in any order, followed by the executable statements.

8.2 Stack

The storage structure used for ALGOL variables is a stack, push-down list, or last-in-first-out store. This is an ordered list of items such that items may only be added or removed at one end. Thus, any new element obscures earlier ones. These become accessible once more when the new element is removed. There is a pointer to the current position of the end of the stack, and this is incremented or decremented as items are added or removed.

When a block is entered, the variables declared therein are added to the stack, being removed on exit from the block. Clearly, some inter-connections must be provided to permit access to global variables located further down the stack. However, if two variables of the same name are on the stack simultaneously, the more recent takes precedence, thereby implementing the 'scope' concept described above.

8.3 Exercises

1. Calculate $\sum_{n=1}^{K} \frac{(-1)^{n+1}}{n^4}$, (i) to t terms ($t \geq 1$); (ii) correct to p decimal places ($p \leq 8$).

   In each case print the sum, and in case (ii) print the number K of terms used.

2. Merge two sets of numbers, stored in order of increasing magnitude in two arrays, into a single, ordered set in a third array. The first two items of data are the numbers of elements in the two arrays.

CHAPTER 9

PROCEDURES

9.1 Introduction

The FORTRAN concept of subprogramme is paralleled by the ALGOL procedure. No specific distinction is drawn to parallel the 'function subprogramme'/'subroutine subprogramme' distinction of FORTRAN but the manner of declaration (i.e. definition) and calling (i.e. use) of a procedure is slightly different in an analogous fashion to FORTRAN.

9.1.1 The purpose of procedures is economization:

(i) If some computation is required at a number of places in a programme, it saves programmer time and machine storage space if a specimen calculation can be written out once only and referred to as necessary - just as you refer to the library procedures (SQRT, COS, etc.) as necessary.

(ii) If another person has written some code to perform a particular operation, it saves programmer time to be able to use this in one's own programme.

9.2 Formal Structure

9.2.1 We must distinguish carefully between the definition of a procedure and what it is to do, and the use of that procedure. A programme which uses procedures is structured thus:

```
begin   declarations of identifiers and procedures

        rest of programme including use of procedures as required

end
```

Note that just like identifiers, procedures need only be declared at the head of the block within which they are required.

9.2.2 A procedure is defined by means of a procedure declaration, which comprises a procedure heading and a procedure body.

The procedure heading contains

(i) the word '<u>procedure</u>' and possibly some information about it;

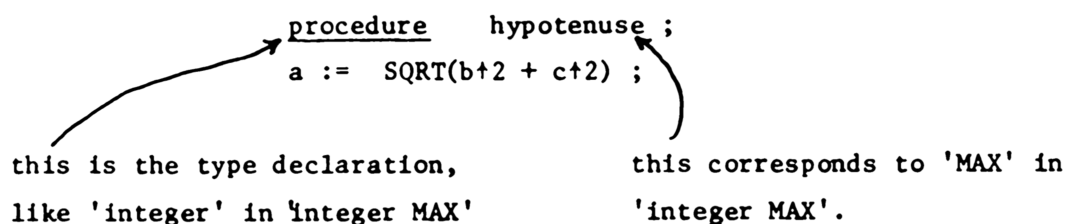
(ii) the name of the procedure;

(iii) the formal parameters or dummy arguments of the procedure, if any (the words 'parameter' and 'argument' will be used interchangeably);

(iv) information about the formal parameters.

(ii) and (iii) are formed according to the usual rules for identifiers.

The procedure body is a single statement, which may be simple, compound, or a block. For example:



This declaration causes a sequence of machine instructions to appear in the programme but does not (even at run time) cause any processing to take place. This only occurs when the procedure is called, which takes place when the procedure name appears in the body of the programme. Thus, if, following the above declaration, the sequence

```
b := 3 ;   c := 7.9 ;
hypotenuse ;
y := 2*a ;
```

appears in the body of the programme, the effect of the statement 'hypotenuse' is to assign to the variable a the value of $\sqrt{(b^2 + c^2)}$, where the a,b,c referred to are understood to be those whose scope includes the declaration of the procedure 'hypotenuse'.

9.2.3 Parameters. It is clear that, without arguments, such a procedure is of only limited value, since to set z := SQRT(x↑2 + y↑2) it would be necessary first to set b := x, c := y, and afterwards to write z := a. We could therefore amend the definition of procedure hypotenuse to:

```
procedure hypotenuse(a,b,c) ;
a := SQRT(b↑2 + c↑2) ;
```

which involves formal parameters a, b and c. When this procedure is called, these must be replaced by actual parameters. The calling example given above would then be

```
b := 3 ; c := 7.9 ;
hypotenuse(a,b,c) ;
y := z*a ;
```

with the same result as before. (Here the actual parameters have the same names as the formal parameters.) Now, if we wish to set z := SQRT(x↑2 + y↑2), we simply write

```
hypotenuse(z,x,y) ;
```

with no auxiliary assignments.

9.2.4 Specifiers

If a procedure has parameters (arguments), the definition of ALGOL provides for the optional specification of these parameters. For most compilers, this specification is essential. Thus, for example, the hypotenuse procedure should be declared thus:

```
procedure hypotenuse(a,b,c) ;
real a,b,c ;
a := SQRT(b↑2 + c↑2) ;
```

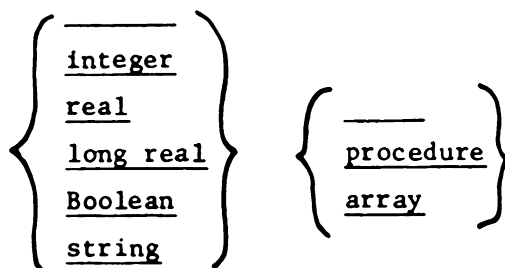
if a,b,c are always to be replaced by real variables.

A more realistic example is:

```
procedure  pivot(A,M,K,Z) ;
comment  this procedure determines the element of largest absolute
value in columns and rows K to M inclusive of the M × M matrix A,
and sets this in Z;
integer  M,K ; real Z ; array A ;
begin  real  Y ; integer  I,J ;
     Z := ABS(A[K,K]) ;
     for  I := K  until  M  do
          for  J := K  until  M  do
               begin  Y := ABS(A[I,J]) ;
                      if  Y > Z  then  Z := Y
               end
end
```

(Note the assignment of ABS(A[I,J]) to Y so as to save time by not recomputing the value of this expression.)

The possible specifiers are label, switch, or



The order of specification of formal parameters is immaterial. No bounds are necessary in specifying arrays.

The purpose of specifiers is to permit the compilation of efficient code. For example, p↑q is compiled as $\underbrace{p*p*\ldots*p}_{q\ \text{times}}$ if q is integral, and $e^{q\log p}$ otherwise. If at compile time, q's type is unknown, inefficient code will result.

Specification allows some errors relating to arrays to be recognized at compile time. However, since the compiler is told only that an identifier refers to an array, and is not told anything about the dimensions of the array, errors involving dimensions are only picked up at run time. For example, if X is specified as a <u>real</u> identifier, the occurrence of X[J] would cause an error indication at compile time; however, if A is a two-dimensional array, and is therefore specified as an array, the occurrence of A[J] would provoke an error response only at run time, as the number of dimensions is unspecified in the procedure.

Although specifications <u>look</u> like declarations, they differ in that no provision for allocating storage space is made as a result of the compiler's encountering them.

9.2.5 Label parameters

The specifier <u>label</u> exists so that labels can be procedure parameters to furnish the possibility of returning from the procedure to different parts of the main programme. (There would be no point in having a label in the procedure body itself as a parameter.) A label parameter must be specified as such. Furthermore, special precautions must be taken to enable the compiler to recognize a label as such when it appears as an actual parameter in a call of the procedure. Consider the following structure:

```
begin real x ;
      procedure fancy (a,ℓ) ;
      real a ; label ℓ ;
      begin Boolean b ;
            :
            if b then goto ℓ else ...
            :
      end ;
  ℓabl: :
      fancy (x, ℓabl) ;
        :
  ℓab:
        :
end
```

When the call of fancy occurs, with actual parameters x and ℓabl, x has been declared, but ℓabl, of course, has not. However, the prior occurrence of ℓabl: has shown ℓabl to be a label, and the compiler will therefore recognize it as such in the procedure call.

If, however, the call were

fancy (x,ℓab)

then ℓab has been neither declared nor used. This would result in a compiler error with message UNDECLARED IDENTIFIER. This may be overcome by making a forward declaration of ℓab in a block containing both the procedure call and the use of the label. In this case, the only such block is the whole programme. The first line should therefore be amended to

```
begin real x ; forward ℓab ;
```

If the destination of the premature exit from the procedure is always the same location (lab:) in the main programme, then lab need not be a parameter, and no forward declaration is needed at all.

9.2.6 Procedure parameters

Finally, procedures may appear as parameters of other procedures, and so must be specifiable. For example,

```
procedure sum(X,M,N,F) ;
procedure F ; real X ; integer M,N ;
begin integer I ;
     X := 0 ;
     for I := M until N do
          X := X + F(I)
```

is an implementation of $\sum_{i=M}^{N} F(i)$. The procedure call

```
sum(Z, 1, 20, SQRT)
```

has the effect of setting $Z = \sum_{i=1}^{20} \sqrt{i}$.

9.3 Functions, or Typed Procedures

In many situations in which the procedure concept is useful, we are most interested in one particular value which results from the call of the procedure; for example the pivot, or the sum, in the above examples. It is most likely that immediately after the call

```
pivot(ARR, LIM, I, PIV) ;
```

we shall have a statement involving the newly calculated value PIV. In this circumstance, it is possible to economize even more by using a _function procedure_ or _typed procedure_. Thus,

```
real procedure  pivot(A, M, K) ;
integer M,K ; array A ;
begin real X, Y ; integer I, J ;
      X := ABS(A[K,K]) ;
      for I := K until M do for J := K until M do
          begin Y := ABS(A[I,J]) ;
                if Y > X then X := Y
          end ;
      pivot := X
end
```

Note that the variable 'pivot', which is the procedure name, is assigned a value within the procedure body. This procedure is used (called) by writing its name, and actual parameters, wherever its value is desired, anywhere that a variable may be used; for example

```
AXE := Z↑3 - PIVOT(ARR, LIM, I) ;
```

Notes

i) The call of a typeless procedure is a statement in its own right.

ii) The call of a function procedure represents a variable.

iii) In each case, the call brings the procedure body into action, with actual parameters replacing formal ones.

iv) Whether typed or typeless, the values of the actual parameters of a procedure are available for use after its call, with new values if these are assigned in the procedure body.

v) Referring to the above examples, pivot(A,M,K,Z) (typeless) is "how to get the pivot", while pivot(A,M,K) (typed) is "the actual value of the pivot".

9.4 Identifiers in procedures

Each identifier appearing in the body of the procedure declaration must be in one of the following categories:

(i) A local variable, declared at the start of the body;
(ii) a formal parameter, specified in the procedure heading;
(iii) a global variable, whose scope includes the procedure declaration.

Identifiers in category (i) will also be in this same category at each call of the procedure.

Identifiers in category (ii) will be replaced at any call by actual parameters, which must be variables whose scope includes that call.

Identifiers in category (iii) will be the same at a call as at the declaration, even if there are more local variables of the same name whose scope includes the call. For example, in

```
begin  real a, p ;
       procedure  test(x) ; real x ;
       begin  x := a↑2 ;
              print(x)  ; newline
       end ;
         :
       a := 5 ;
       test(p) ;
         :
       begin  real  a ;
              a := 8 ;
              test(p) ;
       end
end
```

each call of test uses the identifier a declared in the first line; even the second call does not use the a which is local to the inner block.

Output from the programme is therefore

```
2.5000000&␣␣1
2.5000000&␣␣1
```

Note that the second line of this programme involves an identifier x which has not been declared. This is only correct because it is a formal parameter of a procedure.

9.5 Calling Mechanisms for Actual Parameters

9.5.1 Call by name. The ALGOL report specifies that the method of passing actual parameters to a procedure should be such that the procedure call has exactly the same effect as a copy of the procedure body, within which each occurrence of each formal parameter has been simply replaced by the corresponding actual parameter as it stands. This is referred to as call by name and has the consequence that wherever a formal parameter appears in the procedure body, the corresponding actual parameter is re-evaluated as if it appeared in the procedure body at that point. For example,

```
begin  real  r, s, t, y ;
       real  procedure  max(a,b) ;  real a,b ;
       begin  real  x ;
              x := if  a >= b then  a  else  b ;
              max := x
       end ;
        :
       y := max(r*r, s*t - r*r)
        :
end
```

The call of max is carried out as if, firstly, each occurrence of a were replaced by r*r, and each occurrence of b by s*t - r*r, and secondly, the procedure body were executed. This involves evaluating s*t - r*r twice over, and r*r a further twice, which is clearly an inefficient proceeding. However, in

```
begin  integer i, k;  array  arr[1:100];  real z;
       procedure example(x,y,n);
       real x, y;  integer n;
       begin  real a, b;
              i := 1;  a := x;
              i := 2;  b := x;
       end
        .
        .
        .
     example(arr[i],z,k);
        .
        .
        .
end
```

The call of example is equivalent to

```
   begin  real a, b;
          i := 1; a := arr[i];
          i := 2; b := arr[i];
          .
          .
          .
   end
```

and so a is set equal to arr[1], while b becomes arr[2], and this is presumably the desired effect. Thus, the concept of replacement can be valuable. Note that the identifier i is global.

The reason for the term 'call by name' is as follows. An identifier x specifies a particular memory location, namely the one which is allocated to this identifier when it is declared. At any particular time, this memory location holds a particular _value_ which is a constant. Thus 'x' may be thought of as the _name_ of the variable, and the contents of the memory location designated by x at any time is the current _value_ of that variable.

In the above method of calling an actual parameter, therefore, the procedure is informed of the name of the variable, and the value used at any place in the execution of the procedure body is the value _then_ current.

9.5.2. _Call by value_. This rather sophisticated method of passing parameters can be wasteful. As exemplified above, if the actual parameter is a complicated expression which does _not_ vary throughout the execution of the procedure body, its re-evaluation at each occurrence of the corresponding formal parameter is a waste of time. ALGOL therefore provides the capability of calling an actual parameter _by value_. This is done as follows:

```
procedure  example(x,y,n);
value x, n;  integer  n;  real  x, y;
begin  :
```

namely by a value declaration, which is the first type declaration in the specification part of the procedure heading. Any variables not specified as being called by value are called by name. Following the above declaration, the call

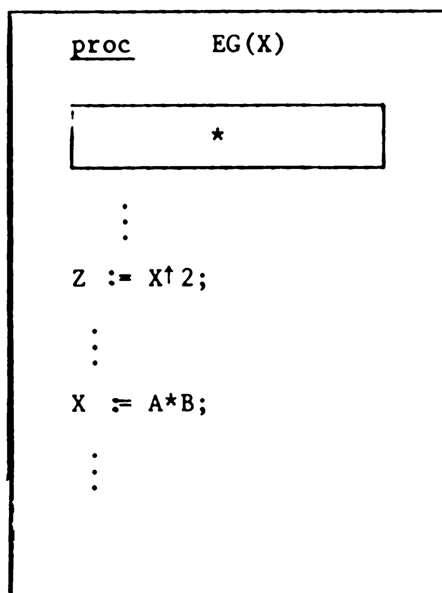
example(SQRT(Z↑2 + T↑2), A+B*C, INDEX[I]);

would be dealt with as follows. SQRT(Z↑2 + T↑2) would be evaluated, the value stored in a temporary location, and the address of this location made available to the procedure for reference and use wherever the name of the formal parameter x appears in the procedure body. A similar comment applies to INDEX[I] and n. The other parameter, y, however, would effectively be replaced by 'A+B*C' at each occurrence. (In practice, code would be generated to evaluate 'A+B*C' and this short 'sub-procedure' called wherever y appears).

Thus, in practice, whether an actual parameter is called by value or by name, the occurrences of the corresponding formal parameter are replaced by reference to an address. At that location will be found: in the case of a call by value, the value to use; in the case of a call by name, either the value to use (if the actual parameter is a single, unsubscripted identifier) or a reference to the rules for computing the value to use.

9.5.3. The following diagrams illustrate the two methods of passing parameters to procedures.

The declaration of a procedure EG(X) causes the compilation of code for the procedure:



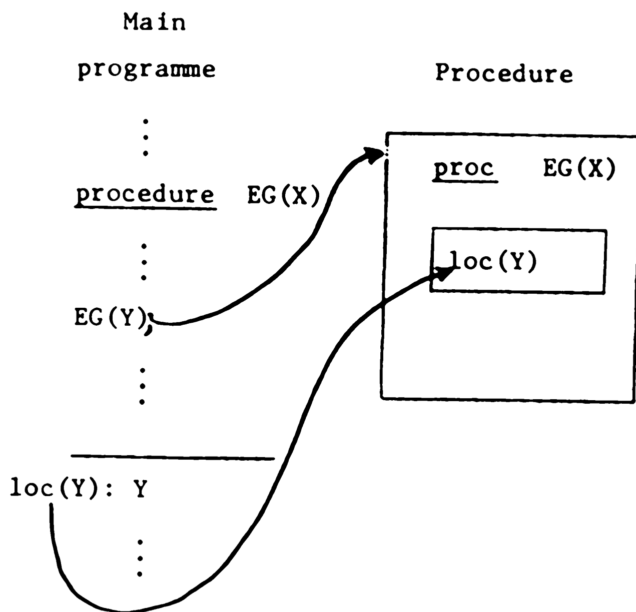
This location is reserved for some reference to the actual parameter to be used to replace X.

Lines such as these are coded to pick up the address of the current replacement for X from location * above.

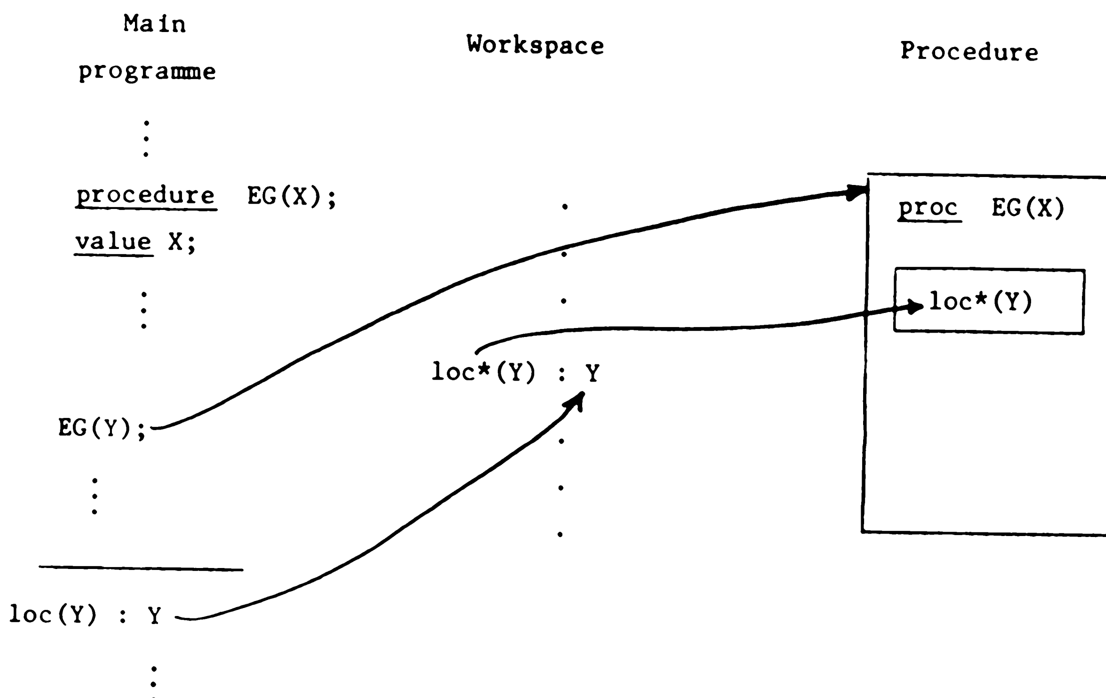
The current value of any identifier Y declared in the main programme is held in some location loc(Y), known to the main programme and reserved for that identifier.

Call by name:

The code compiled for a call, EG(Y);, to this procedure involves storing the address, loc(Y), in location *, followed by a transfer of control to the first statement of the procedure.



Call by value:



In this case, the code compiled for a call, EG(Y);, to this procedure involves obtaining a temporary storage location, loc*(Y), storing the value of Y therein, and storing the address loc*(Y) in location *, before transferring control to the procedure. (The extension to the case in which Y is an expression and not a single identifier is straightforward.)

One result of calling by value is that the actual parameter has the same value after the execution of the procedure call as it did before. Therefore one would not call by value an argument used to transfer the results of the procedure back to the programme (an output value).

A good working rule is:

Call by value all input arguments except i) those which are also output arguments, and ii) arrays (not array elements).

The latter exception is because temporary storage is set up for the whole array, and all its values transferred. This is time- and space-consuming.

Note that the same effect as call by value is obtained by assigning the value of a parameter to a local varaible immediately on call, and then working with that:

```
procedure  example(x,y,n);
value  x;  integer  n;  real  x, y;
begin  real  z;  z: = y;
         .                       }  If  y  is not used in here, we have
         :                       }  effectively 'called it by value'.
end;
```

9.5.4. Call by reference. FORTRAN usually uses neither of these methods of calling, but 'call by reference' or 'call by simple name'. This entails the evaluation of each actual parameter down to a machine address, which, if possible, is not a temporary location set up for the purpose. When this device was originally suggested ([3]) it was only envisaged that it should be used when the actual argument was a single variable name. Therefore a choice must be made of the technique to adopt to deal with an actual parameter which is an expression. The choice is that this is treated as a call by value.

Thus, in calling by reference an expression, the expression is evaluated and the address of its value handed to the procedure: that is, call by value;

in calling by reference a single simple variable, the address of the variable is handed over: that is, call by name;

in calling by reference a single subscripted variable, the address of

the particular array element is handed over: which is neither call by name nor call by value.

For example, if ALGOL also allowed a 'call by reference' (as BCPL does) we could have ([4]):

```
begin  integer  i;  array  A[1:3];
       procedure  report(x,y,z);
       value  x;  ref  y;  name  z;  real  x, y, z;
comment  ref refers to the hypothetical call by reference, and name to
the default call by name;
       begin  i := 3;  A[1] := 5;
              print(x);  print(y);  print(z)
       end;
       i := 1;  A[1] := 2; A[3] := 4;
        report(A[i], A[i], A[i])
end
```

This will print 2, 5, 4 (as real numbers).

x: x is called by value. When report is called, i = 1, A[i] = A[1] = 2, and so this is the value printed since x is unchanged within the procedure.

y: y is called by reference, and so what is handed to the procedure is the address of the actual parameter replacing y, namely A[1] (A[i] where i = 1). But within the procedure a new assignment is made to the global variable A[1], and this therefore changes y to the new value (5).

z: z is called by name, and so it is re-evaluated at each point in the procedure at which reference to it occurs. In particular, in the print statement, reference to z provokes the obtaining of A[i] once more. But, by now, i = 3, and so it is the current value of A[3], namely 4, which is output.

In general, in a call to a procedure EG(ARR[I]), if the parameter is called by

value, then no change of elements in the main programme occurs;
reference, then the one element ARR[I], where I is the value of
the subscript on call, may be changed;
name, any number (≥0) of elements of ARR may be changed.

These sorts of changes in the main programme as a result of the execution of a procedure are called side-effects of the procedure. Their existence necessitates the careful specification of the order of evaluation. For example, since the expressions X+F(N) and F(N)+X are each evaluated from left to right, the assignments

Y := X+F(N); and Y := F(N)+X;

may produce different values of Y.

9.6. Recursion and Iteration

9.6.1 Recursion refers to (i) the call of a procedure using that procedure (implicit recursion), or (ii) a procedure body making use of itself (explicit recursion).

9.6.2(i) In the expression SQRT(X), X may be any arithmetic expression; in particular one involving SQRT, such as

A↑2 + SQRT(B), so we have

SQRT(A↑2 + SQRT(B)).

Basically, there is no problem here. The compiler needs the mechanism to 'unpick' arbitrarily complicated arithmetic expressions not involving function calls, so as to produce a string of machine instructions, ending with a call to the square root function, so it is not difficult to include the facility for 'unpicking' function calls as well.

However, once this sort of nesting of function calls is permitted, it should be allowed to a reasonable depth. Furthermore, it is usually permitted for the programmer to write his own functions of the same name as the standard ones, to replace those, so any such implicit recursion must be available to all functions or none.

The difficulties this caused, led some earlier implementations of FORTRAN (and ALGOL) to forbid this implicit recursion, although it is now available in both languages on the PDP-10. A stack is used in its implementatio
(see section 8.2).

9.6.3(ii) Explicit recursion is not available in FORTRAN, but is available in ALGOL. This refers to the situation in which a procedure body calls itself, or, in which procedure A calls procedure B which calls procedure C which calls ... which calls A.

The problem here is in saving the information as the control passes to the subroutine. At least the current value of the programme counter must be saved. Many subroutine calls save this at the beginning of the subroutine, and the recursive calling of the subroutine would overwrite this information. Furthermore, if such recursive calls are allowed, it is generally impossible to tell at compile time how deep the recursion will go. Again, therefore, the stack is called into play, and the return values of the programme counter placed successively on the top of the stack, and the value of the pointer to the stack adjusted accordingly.

For example (this is the standard example);

```
real procedure factorial(n);
value n; integer n;
if n = 1 then factorial := 1
         else factorial := n * factorial(n-1);
```

The call 'factorial(6);' invokes also the calls factorial(5), and so on, and finally the build-up of the answer. This is a top-down system, in which the quantity we wish to compute is successively broken down into its components.

9.6.4. Iteration, on the other hand, is essentially a bottom-up operation, starting from nothing and building up the result step by step; for example,

```
real procedure factorial(n);
value n; integer n;
begin integer i, j; i := 1;
      for j := 1 until n do i := i*j;
      factorial : := i
end;
```

9.6.5. The main uses of recursion are in handling tree-structured data. Here one can write a procedure to deal with a node, which calls itself when there are other nodes depending from that one, and is 'earthed' otherwise. The factorial function is not of this from, and is much better treated iteratively. In fact, recursion has so far found little application in ALGOL and related languages: it is much more vital in list-processing languages. There are few strictly numerical applications for which recursion can be used, and fewer still for which it is efficient. For most scientific and business problems, there is no reason to use recursion instead of iteration when both are feasible. In fact, almost any computation which can be defined recursively can also be defined iteratively. If, however, a function is doubly recursive, it cannot usually be expressed iteratively.

Example: polynomial evaluation.

$$f(x) = \sum_{i=0}^{n} a_i x^{n-i}$$

(i) Recursion

```
real procedure poly(a,x,n);
value x, n; integer n; array a; real x;
if n = 0 then poly := a[0]
         else poly := x*poly(a,x,n-1)+a[n];
```

(ii) Iteration : synthetic division

The obvious way to evaluate f(x) without recursion is

```
fx := 0;
for i := 0 until n do fx := fx + a[n-i] * x ↑ i;
```

This is clearly inefficient, since x ↑ i is recalculated each time. It is more efficient to save x ↑ i, and calculate x ↑ (i+1) as x * x ↑ i. However, this may be implemented even more efficiently by involving the coefficients in the same scheme, thus:

```
real procedure poly(a,x,n);
value x, n; integer n; array a; real x;
begin real y; integer i;
      y := a[0];
      for i := 1 until n do y := y * x + a[i];
      poly := y
end
```

For all practical purposes this is the best way. (There is a way which is theoretically more efficient, but only if (i) one is performing a lot of calculations for fixed a_i and different x's; and (ii) one ignores the increased roundoff error involved in this other method, due to the greatly increased computation involved.)

Example: Ackerman's function

$$A(0, n) = n + 1, \qquad n \geq 0;$$
$$A(m, 0) = A(m-1, 1), \qquad m \geq 1;$$
$$A(m, n) = A(m-1, A(m, n-1)), \qquad m, n \geq 1.$$

(Although this is doubly recursive, it can be expressed iteratively,)

The recursive facility of ALGOL permits a direct implementation of the recursive definition, thus:

```
integer procedure ACKER(M,N);
value M, N; integer M, N;
if M = 0 then ACKER := N + 1
         else if N = 0 then ACKER := ACKER(M-1, 1)
                       else ACKER := ACKER(M-1, ACKER(M, N-1));
```

However, the depth of recursion uses a very large amount of computer time.

Recursion can be simulated by defining a one-dimensional array to implement the stack 'by hand' rather than leaving it to the compiler.

9.7. Forward and External Declarations

A lot of information is contained in a computer programme, and the compiler attempts to make best use of it all so as to produce a good machine code translation, In order to achieve this, some compilers scan the programme a number of times (multi-pass compilers) to assimilate the

information in context. The PDP-10 ALGOL compiler, however, is a one-pass compiler, aiming for speed of translation through only scanning the programme once, possibly at the expense of some lack of efficiency in the resulting translation.

Since all variables and procedures must be declared before being used, the one-pass compiler knows how to treat identifiers when they are encountered. The one exception to this occurs in the case of procedures which call each other, e.g.

```
begin procedure  P(X, Y);
      begin      .
                 .
                 .
                 Q(Z);
                 .
                 .
                 .
      end;
      procedure  Q(X);
      begin      .
                 .
                 .
                 P(R, S)
                 .
                 .
                 .
      end;
        .
        .
        .
end
```

Here a mention of Q is encountered before its declaration. (Note that this mention is in the declaration of P, not a call of P : Q is of course declared before it is executed in a call of P.) Furthermore, the situation is as bad if the procedure declarations are re-ordered (then P precedes its declaration). To allow the PDP-10 ALGOL compiler to cope with this situation, a FORWARD declaration of Q must be made (cf. section 9.2.4), preceding the first mention of Q, thus:

```
begin forward procedure Q(X);
      procedure P(X, Y);
          .
          .
          .
      procedure Q(X);
          .
          .
          .
end
```

This informs the compiler that the appearance of Q before its declaration is not an error. The forward declaration and actual declaration

must be in the same blocks.

A similar situation arises if a procedure and the programme are compiled separately, as in this case no declaration of that identifier appears in the programme. In the place where this declaration would have been we write

<u>external</u> < type if any > <u>procedure</u> < procedure name > (< arguments >)

9.8. <u>FORTRAN Interface</u>

It is anticipated that it will be possible to incorporate FORTRAN subroutines into ALGOL programmes, but this facility is not working at present (comments in [6] notwithstanding).

9.9. <u>Exercises</u> 1. Calculate Ackerman's function without depending on the compiler for recursion (by using iteration or implementing the stack by hand : see [5]).

2. Solve a polynomial equation by the Newton-Raphson method.

CHAPTER 10

STRING VARIABLES

10.1 Byte Strings

A string variable points to a byte string. The declaration

string S;

allocates two adjacent store locations for S (just like a long real declaration). Subsequent to this declaration, the assignment

S := NEWSTRING(100,7);

creates a string of 100 7-bit bytes, each initialized to zero, and sets S to point to this string. The string is stored right-justified in successive words, with at least one byte per word (so no byte may contain more than 36 bits : c.f. §5.5), but without splitting bytes across words. Thus, a string of 20-bit bytes is stored with 1 byte per word; 10-bit bytes are stored 3 per word; and so on. The number of bytes in a string is limited only by the availability of storage.

The effect of the assignment

T := S;

is to copy the byte string pointed to by S, and then set up in T a pointer to this new string. More discriminating replication of byte strings may be obtained by means of the procedure COPY.

T := COPY(S, m, n);

generates a new string identical to the string which forms the mth, $(m+1)^{th}$, ..., n^{th} bytes of S, and assigns it to T.
COPY(S, n) denotes COPY(S, 1, n), while
COPY(S) denotes COPY(S, 1, n) where n is the number of bytes in S.

If L is an integral expression, S.[L] denotes the integral value of the Lth byte of the string S.

We may compare strings. For example, if

string R, S, T;
R := "ABCD"; S := "ABCDE"; T := "ABCE",

then R < S < T.

10.2 String Arrays

We may have string arrays, for example

string array T[1 : M];

and then T[K].[L] is the Lth byte of string T[K].

10.3 Other Library Procedures

Strings may be joined by means of the string procedure CONCAT, which takes two string parameters. Thus,

```
R := "ABC"; S := "DEF"; T := "GHI";
U := CONCAT(R, S); V := CONCAT(T, U);
```

makes U point to a string "ABCDEF", and V to a string "GHIABCDEF".

I := LENGTH(S) sets I equal to the number of bytes in the string possessed (i.e. pointed to) by S.

DELETE(S) deletes the byte string possessed by S, if this is dynamically created (that is, produced by COPY or NEWSTRING).

10.4 Input and Output (c.f. §§5.2.1, 5.4 and 5.5)

'In-' and 'outsymbol', etc, and 'write' are oriented towards ASCII, 7-bit, bytes. Thus, on input

```
insymbol(J);
```

reads the next ASCII character and attempts to store its 7-bit ASCII value right-justified in J. In particular, if J is S.[L], and S has N-bit bytes, where $N \geq 7$, then the ASCII value is stored as the low-order 7 of the N bits which hold the Lth byte of S. If, on the other hand, $N < 7$, then the right-most N bits of the ASCII value are stored in that byte position.

Similarly, on output,

```
outsymbol(J);
```

picks out the rightmost 7 bits of J, and outputs the corresponding ASCII character. If J is a byte with less than 7 bits, this will fail, and no output will be produced.

The action of write is similar. If S is string of M bytes,

```
write(S);
```

has exactly the same effect as

<u>for</u> I := 1 <u>until</u> M <u>do</u> outsymbol(S.[I]);

If more than 7 bits of a byte are required, this may be obtained in numeric form by using 'print' and in character form by division and the use of outsymbol. The latter method is illustrated in Example 3 of p.28, which is applicable in the string situation if the variable i is replaced by S.[L] for some L, and some string S whose bytes contain at least 21 bits. For the former, observe that after

```
string S; S := newstring(1, 30);
S.[1] := $.ABC.;
```

write(S); gives C

while print(S.[I]) gives 1073475

(which is the decimal integral value of the word which has the ASCII values

of A, B, C in its rightmost 21 bits).

10.5 Byte Manipulation

Two library procedures GFIELD ('get field') and SFIELD ('set field') permit the manipulation of any field (contiguous sequence of bits) within any variable, thus:

```
GFIELD(var, pos, len)
SFIELD(var, pos, len, val)
```

where pos, len and val are of type integer.

var is the variable name;

pos specifies the left hand end of the field. If var is integer, real or Boolean, then $0 \leq pos \leq 35$ and $pos + len \leq 36$; while if var is of type long real or string, $0 \leq pos \leq 71$ and $pos + len \leq 72$;

len is the length of the field in bits;

GFIELD is an integer procedure, so we may write, for example,

```
IND := GFIELD(A,ϕ,7)
```

while SFIELD is a typeless procedure and sets the specified field to the value val:

```
SFIELD(B,10,20,16)
```

10.6 String Parameters

Although the following structure should be correct:

```
begin  string  t;
       procedure  ALPHA(s);   string s;
       begin . . .  end;
       t := newstring(50,7);
       ALPHA(t)
end
```

reference to any byte of s within the procedure body will sometimes (but not always) give an illegal memory reference or a byte subscript out of range. This may be overcome by introducing a string variable which is local to the procedure body, thus:

```
begin  string  r;  r := newstring(50,7);
       :
       s := r
end;
```

if s is an output variable from the procedure; or

```
begin  string  r;  r := newstring(50,7)
       r := s;
```

end

for an input variable.

10.7 Exercises. 1. Write a programme to read two Roman numerals, convert them to Arabic form, and then print their sum in Roman form.

2. Convert an ALGOL programme using only upper case letters to one using upper and lower case letters.

CHAPTER 11

OWN VARIABLES AND SWITCHES

11.1 Own Variables

In Chapter 8 were introduced the concepts of local and global variables. The value of a local variable is normally lost on exit from the block in which it is defined. However, one would sometimes like to preserve the value of a variable after exit from a block for use on a subsequent re-entry to that block, while not wishing to use it outside the block. In this case, it need not be a global variable.

This end may be achieved by means of the <u>own</u> declaration, as in

```
    .
    .
    .
begin  own  integer  I;  own  array  A[1:M, 1:N];  real  Z;
          .
          .
          .
end ;
          .
          .
          .
```

This has the effect of preserving the values of I and the elements of A on exit from the block. These cannot be referred to from outside the block, and may have the same names as other identifiers declared elsewhere.

There is a snag : the variable I can only be assigned a value within its scope, that is, in the block. If this is done, surely the point of preserving the value is lost? PDP-10 ALGOL overcomes this by initializing all <u>own</u> variables to zero before programme execution. We shall mention a more general way later.

11.2. Dynamic Own Arrays

It is quite likely that the values of M and N, the subscript bounds in the array above, will be different on successive entries to the block. What does it <u>mean</u> to preserve the values of, say, a 3 x 5 array for assignment to a 6 x 2 array? The assignment is made thus:

if the pair [i, j] appears in both subscript ranges, then

$A[i, j]_{new} := A[i, j]_{old}$;

if it appears in the new range but not the old, then

$A[i, j]_{new} := 0$;

if it appears in the old but not the new, then the storage location for

$A[i, j]_{old}$ is released.

11.3. Switches

11.3.1. A switch is like a FORTRAN computed GOTO. A switch identifier follows the normal rules for construction of identifiers, and in use is followed by a subscript expression which refers to an element in a switch list. For example,

```
switch SELECT := LAB1, LAB2, LAB3, LAB4;
goto SELECT[2];
```

transfers control to the statement labelled LAB2. The former of these two statements is a declaration and must appear with the other declarations at the head of the block, before any executable statement.

If the value of the subscript expression is ≤ 0 or > the number of entries in the switch list, then control passes to the next statement in sequence after the goto statement.

11.3.2. Labels such as LAB1, and switches like SELECT[int. exp.] are examples of designational expressions. A switch list is a list of designational expressions, for example:

```
switch TIME := NEXT, FIRST, SELECT[1];
```

Then, if I has the value 3, goto TIME[I]; is equivalent to goto SELECT[1]; which in turn is equivalent to goto LAB1;

Since the switch identifier SELECT appears in the above declaration, this must follow the declaration of SELECT. If two switch identifiers each appear in the switch list of the other, a forward declaration must be used.

```
forward switch TIME;
switch SELECT := LAB1, LAB2, LAB3, TIME[2];
switch TIME := NEXT, FIRST, SELECT[1];
```

Designational expressions may be more complex in form; for example:

```
goto if b.e. then LAB1 else LAB2;
```

With the power of the ALGOL conditional and compound statements available, switches are of limited use.

11.4. Example. A switch may be used to initialize an own variable. To calculate the fibonacci sequence, 1, 1, 2, 3, 5 ... we may write:

```
begin integer CHOICE, NEWVAL;
      CHOICE := 1;
FIBON : begin own integer FIB1, FIB2; integer TEMP;
            switch SELECT := FIRST, NEXT;
            goto SELECT[CHOICE];
      FIRST : FIB1 := FIB2 := 1;
              PRINT(FIB1,4);    PRINT(FIB2,4);
              CHOICE := 2;
      NEXT :  TEMP := FIB1 + FIB2;
              FIB1 := FIB2;
              NEWVAL := FIB2 := TEMP;
              PRINT(NEWVAL)
      end;
      if NEWVAL < &10 then goto FIBON
end
```

Notes (i) Whatever device is used to initialize own variables, it must include some global identifier. Here we use CHOICE.

(ii) NEWVAL is introduced to make the current fibonacci term available globally.

(iii) It is typical of the use of own variables that an iterative process is involved in which the output of one iteration gives the input for the next.

(iv) If this were the whole programme the use of own and the switch would be unnecessary. It could be of use if the conditional statement 'if NEWVAL ...' were replaced by a significant amount of programme which used successive terms of the Fibonacci sequence.

11.5. Exercises. 1. Write a random number generating procedure, using an own variable to hold the random number and provide the starting point for successive steps.

2. Use a switch to control the calculation of one of the first 5 Legendre polynomials

$$(p_n(x) = \frac{1}{n!2^n} \cdot \frac{d^n}{dx^n} (x^2-1)^n \quad , \qquad n = 0, 1, 2, 3, 4.)$$

CHAPTER 12

RUNNING AND DEBUGGING

12.1 Running ALGOL Programmes

It is assumed that the reader is familiar with the DEC System-10, and no attempt will be made here to summarize the command language or editing facilities. These are well set out in [7].

12.2 Debugging ALGOL Programmes

12.2.1. Following Dijkstra, we may observe that the best way to proceed here is to avoid the debugging stage by not writing errors into the programme in the first place. If you fail in this, you may use the DDT programme ([8]) in the usual way. However, if you are working on-line, the simple expedient of inserting a number of statements ("snapshots") which cause output to occur to the teletype, can locate errors quickly and efficiently.

When the programme is debugged, these output statements are no longer required. If desired, this effect may be obtained by using the following procedure.

```
procedure   snapshot(x, y, loc, key);
value x, y, loc, key;  real x;  integer  y, loc, key;
if  key ≠ 0 then  begin   output(15, "TTY");  selectoutput(15);
                          write("[N]LOC = ");  print(loc);
                          print(x);  print(y);   newline
                   end;
```

Then, on debugging runs, set key = 1, and when the programme is correct, change this one assignment to key : = 0. Channel 15 should not be used elsewhere; x and y allow the output of real and integer values; loc locates the point reached in the programme.

12.2.2. The array bound checking facility mentioned in section 6.1. is very useful when a programme contains a large number of array references.

12.2.3. <u>Compiler errors</u>. Compilation errors are reported in reasonably intelligible form, together with the ordinal number of the line of the programme in which the error was noted. For example, the message

16 UNDECLARED IDENTIFIER

should cause you to look in line 16 (not <u>statement</u> 16) for a situation which would look to the compiler like an undeclared identifier. This could well be a misspelled identifier or reserved word.

However, the compiler tries to force a translation as far as

possible on the assumption that the programme is correct. Clearly it may do this by interpreting the logic differently from the way the programmer intended it. The translation may finally break down some way past the point at which the 'actual error' occurred. For example, one programme failed to compile and gave the single error message

72 DECLARATIONS MUST BE TERMINATED BY SEMICOLON.

On inspection, one compiler error was found: namely the omission of an end in line 21!

A complete list of compile-time error messages appears in Appendix 3.

12.2.4. Run-time Errors. If a run-time error occurs, an error message is produced, detailing the type of error and its address (not line number). Such errors are either fatal or non-fatal. The latter sort may be trapped and control transferred to a label within the programme; thus after the statement

TRAP(ERR, LAB);

has been executed, any occurrence of error number ERR causes control to be transferred to the statement labelled LAB. ERR is an integer expression. The statement

TRAP(ERR);

turns off the trapping of error number ERR.

The trap numbers are listed below.

TRAP NO.	RUN-TIME ERROR
18	FLOATING POINT OVERFLOW
19	FIXED POINT OVERFLOW
32	INPUT OR OUTPUT DEVICE UNAVAILABLE
33	ILLEGAL MODE FOR INPUT OR OUTPUT DEVICE
34	INPUT OR OUTPUT ON UNDEFINED CHANNEL
35	ATTEMPT TO READ OR WRITE ON DIRECTORY DEVICE WITHOUT FILE OPEN
37	FILE NOT AVAILABLE OR RENAME FAILURE
38	ATTEMPT TO READ OR WRITE OVER END-OF-FILE
39	ERROR CONDITION ON INPUT OR OUTPUT
40	ILLEGAL CHARACTER IN NUMERIC DATA
41	OVERFLOW IN NUMERIC DATA
42	ERROR CONDITION ON CLOSING FILE
43	ILLEGAL INPUT-OUTPUT OPERATION

TRAP NO.	RUN-TIME ERROR
44	I-O CHANNEL NUMBER OUT OF RANGE
48	SQRT ARGUMENT NEGATIVE
49	LN ARGUMENT ZERO OR NEGATIVE
50	EXP ARGUMENT TOO LARGE
51	INVERSE MATHS FUNCTION ARGUMENT OUT OF RANGE
52	TAN ARGUMENT TOO LARGE

12.3. Common Sources of Error

1. If B is Boolean, B and not B may both be true (section 2.4.3).

2. If S and T are strings, setting S:=T and then changing T also changes S (section 10.1).

3. If M and N are integers, M/N produces a real result (section 2.2; FORTRAN programmers take note).

4. The semicolon is unnecessary, but not incorrect, before 'end'. It is incorrect before 'else'.

5. Reserved words must be treated as such.

6. No two arithmetic operators may appear in succession. In particular, n DIV -3 and n REM -5 must be replaced by n DIV (-3) and n REM (-5) respectively.

7. All exponentiation involving variables is compiled into machine code floating point commands (even 3↑5, for example). Thus,

```
I := 3;  J := 7;  K := 8;  Z := I + J * 10 ↑ K;
```

gives Z = 700,000,000, while

```
Z := 3 + 7 * 10 ↑ 8;
```

gives the correct result.

8. Leave a space after the last data item in a file to prevent end-file errors.

Bibliography

[1] P.C. Sanderson, "Computer Languages", Newnes-Butterworths, 1970.

[2] P. Naur, "Report on the Algorithmic Language ALGOL 60", CACM, 3, 1960, 299-314.

[3] C. Strachey & M. V. Wilkes, "Some Proposals for Improving the Efficiency of ALGOL 60", CACM, 4, 1961, 488-91.

[4] B. Higman, "A Comparative Study of Programming Languages", Macdonald/Elsevier: Computer Monographs, #2, 1967.

[5] A. Ralston,"Introduction to Programming and Computer Science", McGraw-Hill, 1971.

[6] DEC System 10 "Mathematical Languages Handbook", 1974.

[7] DEC System 10, "Users' Handbook", 1974.

[8] DEC System 10, "Assembly Language Handbook", 1974.

Solutions to Exercises

Exercises 2.5

1. i), ii), iv), vi) are valid. viii) is valid if delimiter words are not reserved.

2. ii), iii), iv), v), vi), vii), viii) and x) are all valid.

Exercises 3.8

1.
```
begin integer m, n, large,small, quot, rdr;
comment to find the g.c.d. of integers m and n by the
Euclidean Algorithm;
       if m < n then begin large := n; small := m end
                     else large := m; small := n end;
LAB :  quot := large DIV small;
       rdr := large REM small;
       if rdr > 0 then begin large := small;
                             small := rdr;
                             goto LAB
                       end;
comment at this point, the value of small is the g.c.d.;
end
```

2.
```
begin  integer i, sum;
       sum := 0; i := 1;
LAB :  sum := sum + i ↑ 3;
       i := i + 1;
       if i < = 100 then goto LAB;
comment at this point, sum has the required value;
end
```

(A better implementation of this uses the _for_ statement: see Exercise 7.4.1.)

Exercises 3.8 (cont'd)

```
3.  begin real area1, area2, differ, fval, x ; integer n;
            x := area1 := area2 := 0; n := 0;
    LAB1 : fval := 1/(1+x↑2); area1 := area1 + 2*fval;
            x := x + 0.05; n := n + 1;
            if n < = 20 then goto LAB1;
            area1 := (area1-1.5)*0.025;
            x := 0; n := 0;
    LAB2 : fval := 1 / (1 + x ↑ 2);
            if n = 0 or n = 20 then area2 := area2 + fval
                    else if (n DIV 2) * 2 = n then area2 := area2+
                                                        2 * fval
                              else area2 := area2 + 4 * fval;
            x := x + 0.05; n := n + 1;
            if n < = 20 then goto LAB2
    Comment because of round off error in real and long real
    variables, it is better to test whether n=20 than whether x=1;
            differ := area1-area2
    end
```

Exercises 4.2.

```
1.  begin real A, EPS, ROOT, NEXT; integer N;
             read(A, EPS);
    comment this obtains the desired values for A and EPS, for this
    run, as described in Chapter 5;
            ROOT := 1; N := 1;
    LAB :   NEXT := (ROOT + A/ROOT) / 2;
            if ABS(ROOT - NEXT) > = EPS then
                begin ROOT := NEXT;
                      N := N + 1;
                      goto LAB
                end;
    comment here NEXT has the value of the square root of A to within
    EPS, and it has taken N iterations;
            ROOT := SQRT(A); EPS := ABS(ROOT - NEXT);
    comment this compares the calculated value with the library value;
    end
```

(See section 7.2.4 for a better implementation of this algorithm.)

Exercises 4.2. (cont'd)

2.
```
begin real X1, X2, X3, Y1, Y2, Y3, SIDE1, SIDE2, SIDE3, SEMI, AREA;
comment read in values for the X and Y co-ordinates;
        SIDE1  :=  SQRT((X2 - X3)↑2 + (Y2 - Y3)↑2);
        SIDE2  :=  SQRT((X1 - X3)↑2 + (Y1 - Y3)↑2);
        SIDE3  :=  SQRT((X1 - X2)↑2 + (Y1 - Y2)↑2);
        SEMI   := (SIDE1 + SIDE2 + SIDE3) / 2;
        AREA   :=  SQRT(SEMI * (SEMI-SIDE1) * (SEMI - SIDE2) *
                                  (SEMI - SIDE3))
end
```

3.
```
begin real R1, R2, THETA1, THETA2, R3, THETA3, R, THETA,
           X1, X2, Y1, Y2, X3, Y3, X, Y;
comment read in the given values of the data;
        X  := R1 * COS(THETA1) + R2 * COS(THETA2);
        Y  := R1 * SIN(THETA1) + R2 * SIN(THETA2);
        R3  :=  SQRT(X↑2 + Y↑2);
        THETA3  :=  ARCCOS(X/R3);
comment end of first part;
        R  :=  SQRT(X1↑2 + Y1↑2); X  :=  SQRT(X2↑2 + Y2↑2);
        THETA  :=  ARCCOS(X1/R);  Y  :=  ARCCOS(X2/X);
        R  := R*X;  THETA  := THETA+Y;
        X3  := R*COS(THETA);  Y3  := R*SIN(THETA);
comment storage space would be saved by using X3, Y3 as temporary
variables as well as result variables, in place of X, Y.  Note
that none of the input data is overwritten;
end
```

Exercises 5.9

3.8.1.
```
begin integer m, n, large, small, quot, rdr;
comment ... ;
        output(1, "TTY");   output(7, "DSK");
        selectoutput(1);
        WRITE("TYPE INTEGERS FOR PROCESSING. [NB]");
        selectoutput(7);    openfile(7, "RESEUC");
        read(m,n);
comment values of m and n are typed on the teletype, separated
by spaces and terminated by a 'return' character;
        if m < n  ...
```

Exercises 5.9 (cont'd)

```
          write("GCD OF");   print(m);   write(" AND");
          print(n);   write(" IS");   print(small);
          newline
     end

3.8.2.     begin  integer  i, sum, exp, lim;
     comment this is a generalization of the previous version:
           input(1, "CDR");  selectinput(1);
           output(2, "LPT");  selectoutput(2);
           read(exp, lim)
          sum  := 0;  i  := 1.
     LAB :  sum  := sum + i↑exp;
          i  := i + 1;
          if i < = lim then goto LAB;
           write("SUM OF");  print(exp);
           write("TH POWERS OF THE FIRST");  print(lim);
           write(" POSITIVE INTEGERS IS:");  print(sum);
          newline
     end

3.8.3.          .
          .
          .
          output(5, "DSK");  selectoutput(5);
          openfile(5, "INTEG.RES");
          .
          .
          .
          write("[20S]INTEGRALS[N5S]TRAPEZIUM[11S]SIMPSON
               [8S]DIFFERENCE[N3S]");
          print(areal,  6);  space(3);   print(area2,  6);  space(3);
          print(differ, 6);  newline
     end
```

4.2.1, 4.2.2, 4.2.3 may be handled similarly.

Exercises 6.3

```
1.  begin integer n; real ptemp, qtemp, xtemp; array p, q, rat,
                                                        x[1:20];
          output (5, "LPT"); selectoutput(5);
          write ("[9S]P[15S]Q[13S]RATIO[11S]NEWTON[N]");
          n := 1; p[n] := q[n] := rat[n] := x[n] := 1;
    first : space(3);  print (p[n], 6); space(3);  print (q[n], 6);
          space(3);  print (rat[n], 6); space(3);  print (x[n], 6)
          newline;
          n := n+1;
          if n > = 21 then goto last;
          ptemp := p[n-1]; qtemp := q[n-1]; xtemp := x[n-1]
          p[n] := ptemp + 2*qtemp;
          q[n] := ptemp + qtemp;
    comment by using the temporary variables, three array accesses
    are saved. Whenever the same array element is used more than
    once, it is worth introducing a temporary variable to reduce
    execution time. However, do not carry this to such an extent
    that the sense of the programme is hidden from the human reader;
          rat[n] := p[n]/q[n];
          x[n] := (xtemp + 2/xtemp) / 2;
          goto first;
    last :
    end

2.  begin integer prod, loc; real sale, tot;
          array val[1:3,1:5];
    comment each val[i,j] is set to zero at declaration;
          input(1,"CDR");  selectinput(1);
          output(2,"LPT");  selectoutput(2);
    readin : read(prod);
          if prod = ϕ then goto results;
          read(loc, sale);
          val[prod, loc] := val[prod, loc] + sale;
          goto readin;
results : write("[20S]SALES ANALYSIS[NS]LOCATION[35]
              PRODUCT1[9S]PRODUCT2[9S]PRODUCT3[N]");
          loc := 1;
again :   write(loc, 4);  SPACE(3);  write(val[1, loc]);
          SPACE(3);  write(val[2, loc]); SPACE(3);
           write(val[3, loc]); NEWLINE;
```

Exercises 6.3 (cont'd)

```
        loc  := loc + 1;
        if  loc < = 5 then  goto  again
    end
```

Exercises 7.4.

```
1.  begin  integer  i, sum, exp, lim;
        sum  := 0; i  := 0;
        while i < lim do begin i  := i + 1; sum  := sum + i↑exp end
    end

2.  begin  integer i, j, capt, r, s, k, m;
        integer  array  board[1 : 8, 1 : 8];
    comment  this programme calculates the number of possible captures
    for white in a given checker position.  -2, -1, 0, 1, 2 respectively
    represent black king, man, unoccupied, white man, king;
        for i  := 1 until 8 do  begin  r  := if  (i div 2)*2  = i
                                              then  1  else  2;
                                       for j := r step 2 until r+6
                                              do read(board[i,j])
                                 end;

    comment piece locations are listed by ranks:  i. e. board[i, j]
    refers to square on rank i and file j;
        capt  := 0;
        for i:= 1 until 8 do for j  := 1 until 8 do if board[i, j]
                                    > 0 then for k  := -1, 1 do for
                                    m  := -1, 1 do
        begin  r  := i + 2 * k;  s  := j + 2 * m;
              if (r > 0 and r < 9 and s > 0 and s < 9 and board
                [i+k, j+m] < 0 and  board[r, s] = 0 then capt  :=
                capt + 1
        end;
         write("NUMBER OF POSSIBLE CAPTURES FOR WHITE IS"); print(capt)
    end
```

Exercises 7.4 (cont'd)

```
3.  begin  integer  i, r, s, agree, disagree;  real  KRCC;
           integer  array  judge[1:2,1:10];
           Boolean  array  comp[1:2];
    comment    judge[i,j] is the position of the jth candidate accordin
    to judge i;
           agree  := disagree  := 0;
           for r  := 1 until 9 do
               for s  := r + 1 until 10 do
                   begin for i  := 1, 2 do comp[i]   :=  judge[i, r] >
                                                         judge[i, s];
                         if comp[1]  eqv comp[2]  then agree  := agree + 1
                                                  else disagree  :=
                                                       disagree + 1
                   end;
           KRCC  := (agree - disagree) / 45;
           print (KRCC)
    end

4.  begin  integer  num, lim, i;  real x;
    comment tests if num is prime.   num > = 2, or = 0 to terminate;
             output(7, "DSK"); selectoutput(7);  openfile(7, "PRES");
    first : read(num);
            if num # 0 then  begin  print(num);
                                     if num = 2 or num = 3 then
                                    write("IS PRIME [N]")
               else if (num DIV 2)*2 = num then  write("IS NOT PRIME [N]")
               else begin  x  := num; lim  := SQRT(X);
                           for i := 3  step 2 until lim do
                              if (num DIV i)*i = num then
                                 begin  write("IS NOT PRIME [N]");
                                        goto next
                                 end;
                          write("IS PRIME [N]");
                    next;
                    end;
                                  goto first
                             end
    end
```

Exercises 8.3

```
1.  begin  real  sum;  integer n, t, sign;
            input(1, "TTY");  selectinput(1);
           read(t);
    comment the series is summed to t(≥1) terms;
           sign  := 1;  sum  := 0;
           for n  := 1 until  t  do
               begin  sum  := sum + sign / n↑4;
                      sign  := -sign
               end;
            write("THE SUM TO");  print(t);   write("TERMS IS");
           print(sum);  newline;
           begin  real  sumnew;   integer  p, lim;
    comment sum is obtained correct to p (≤8) places, with a cutoff
    point of lim (≥1) terms.  The identifier  p  is not necessary :
    t  could have been re-used;
                  sumnew   := 1;  sum  := 10; n  := sign  := 1;
                  while  abs(sum - sumnew) > .5*10↑(-p)  do
                         begin  n  := n+1; sign  := -sign;
                                sum  := sumnew;
                                sumnew  := sumnew + sign/n↑4
                         end;
            write("THE SUM CORRECT TO");  print(p);
            write(" DECIMAL PLACES IS");  print(sumnew);
            write(". THIS TOOK");  print(n);
            write(" TERMS.[N]");
           if  n = lim then begin  write("MAXIMUM NUMBER OF TERMS WAS
                                          USED.  PENULTIMATE SUM WAS");
                                   print(sum);  newline
                            end
         end
    end
```

(In the printed original, the keywords begin, real, integer, comment, for, until, do, end, while, if and then are underlined.)

Exercises 8.3 (cont'd)

```
2.  begin  integer  m, n, sum;
            input(5, "CDR");   output(6, "LPT");
           selectinput(5);   selectoutput(6);
           read(m,n);    sum :=  m + n;
           begin  real  x, y, z;  integer i, j, k;
                 array a[1 : m+1], b[1 : n+1], c[1 : sum];
    comment  it is a simple matter to select the appropriate a[i] or
    b[j] at each step for inclusion in c.  However, the  a  or  b
    array must be tested at each step to determine whether it has all
    been used, and if so to transfer the remainder of the other array
    directly to c.  This programme adopts an alternative method.
    The size of the largest element in  a  or  b  is determined and a
    number larger than this appended to both  a  and  b.  This ensures
    that the first m+n numbers put in  c  are the desired ones without
    further testing;
           z :=  x :=  0;
           for i :=  1 until  m  do
               begin  read(y);   a[i] :=  y;
                      if y>x  then  x :=  y
               end;
           for i :=  1 until  n  do
               begin  read(y);   b[i] :=  y;
                      if  y>z  then z :=  y;
               end;
           a[m+1] :=  b[n+1] :=  if x>z then x+1 else z+1;
           i := j := 1;
           for k := 1 until sum do
               if a[i] < b[j] then  begin  c[k] := a[i];
                                           i := i+1
                                    end
                              else  begin  c[k] := b[j];
                                           j := j+1
                                    end
         end
    end
```

Exercises 9.9

1.
```
integer procedure acker(m,n);
value m, n; integer m, n;
begin integer ptr, ack; integer array stack[1:200];
comment the array stack is the pushdown list, and the variable
ptr is the stack pointer.  The dimension of stack is arbitrary :
it could perhaps be a parameter of the procedure;
      ptr := 0;
lab : if m = 0 then begin ack := n+1;
                          if ptr ≠ 0 then
                             begin m := stack[ptr];
                                   n := ack;
                                   ptr := ptr-1;
                                   goto lab
                             end
                    end
               else if n = 0 then begin m := m-1;
                                        n := 1;
                                        goto lab
                                  end
                             else begin index := index+1;
                                        stack[index] := m-1;
                                        n := n-1;
                                        goto lab
                                  end;
      acker := ack
end
```

2.
```
begin integer n;
      read(n);
      begin real procedure funct(a,z,m);
            value z, m; integer m; real z; array a;
            begin real y; integer i;
                  y := a[0];
                  for i := 1 until m do y := y*z+a[i];
                  funct := y
            end;
            real procedure deriv(a,z,m);
            value z, m; integer m; real z; array a;
```

Exercises 9.9 (cont'd)

```
                begin real y; integer i;
                      y := m*a[0];
                      for i := 1 until m-1 do y := y*z+(m-1)*a[i];
                      deriv := y
                end;
                array coeff[0:n];    real x, xold, eps;
                read(eps);
                x := 0; xold := 10;
                while abs(x-xold) > eps do
                      begin xold := x;
                            x = x-funct(coeff, x, n) / deriv(coeff,
                                                              x, n);
                      end;
                print(x);
         end;
      comment this programme needs two improvements.  1. An upper
      limit on the number of iterations permitted.  2. The value deriv
      (coeff, x, n) should first be assigned to some identifier y and
      this checked for size before use as a quotient, to prevent overflow;
      end
```

Exercises 10.7

```
   1.  begin integer num1, num2, length; string t;
             integer array decval[67 : 88];
             procedure readroman(s, length);
       comment skips spaces to find a roman numeral delimited by spaces,
       reads numeral into s, and its number of characters into length;
             string s; integer length;
             begin integer  char; string r;
                   r := newstring(100,7);
                   insymbol(char); while char = 32 do insymbol(char);
       comment 32 is the decimal equivalent of the octal number 40, which
       is the ASCII code for the space character;
                   for length := 1, length + 1 while char ≠32 do
                                        begin r.[length] := char;
                                              insymbol(char)
                                        end;
```

Exercises 10.7 (cont'd)

```
                    length := length - 1;
                    s := r
          end;
          procedure romtodec(r, length, num);
          value length; string r; integer length, num;
     comment converts to arabic in num the roman numeral with length
     characters which is in r;
          begin integer k; string s;
                num := 0; k := 1; s := newstring(100,7);    s := r;
                while k <= length do
                      if k = length
                      then begin num := num+decval[s.[length]];
                                 k := k+1
                           end
                      else if decval[s.[k+1]]>decval[s.[k]]
                           then begin num := num+decval[s.[k+1]]
                                             -decval[s.[k]];
                                      k := k+2
                                end
                           else begin num := num+decval[s.[k]];
                                      k := k+1
                                end


          end;
          string procedure dectorom(num);
          value num; integer num;
     comment the value of this procedure is the roman numeral for num;
          begin string r; integer k, i; integer array tens[0:3],
                                                      fives[0:2];
               r := newstring(100,7);   k := 0;
               tens[3] := $.M.; tens[2] := $.C.; tens[1] := $.X.;
               tens[0] := $.I.;
               fives[2] := $.D.; fives[1] := $.L.; fives[0] := $.V.;
               while num>=1000 do begin k := k+1; r.[k] := $.M.;
                                        num := num-1000
                                  end;
```

Exercises 10.7 (cont'd)

```
                for i := 2, 1, 0 do
                    if  num>=9*10↑i  or  5*10↑i>num and num>=4*10↑i
                          then  begin  k := k+2; r.[k-1] := tens [i];
                                       if num>=9*10↑i
                                       then begin r.[k] := tens[i+1];
                                                  num := num-9*10↑i
                                            end
                                       else begin r.[k] := fives[i];
                                                  num := num-4*10↑i
                                            end
                                end
                          else  begin  if  num>5*10↑i
                                       then begin k := k+1;
                                                  r.[k] := fives[i];
                                                  num := num-5*10↑i
                                            end;



                                      while num>=10↑i do begin k := k+1;
                                                              r.[k]:=tens[i];
                                                              num:=num-10↑i


                                                         end
                                end;
            dectorom  := r
        end;
        decval[$.I.] := 1; decval[$.V.]  := 5; decval[$.X.]  := 10;
        decval[$.L.]  := 50; decval[$.C.]  := 100; decval[$.D.]  := 500;
        decval[$.M.]  := 1000;
        t := newstring(100,7)
        readroman(t, length);
        romtodec(t, length num1);
        write(t); print(num1); newline;
        readroman(t, length);
        romtodec(t, length, num2);
        write(t); print(num2); newline;
```

Exercise 10.7 (cont'd)

```
          write(dectorom(num1+num2)); print(num1+num2); newline
     end
```

(This programme translates MIM correctly, but translates 1999 into MCMXCIX.)

2.

```
     begin string s; integer x,y,depth;
          input(0, "dsk");  output(1, "dsk");
          write("source file: [B]"); read(s);
          openfile(0, s);
          write("output file: [B]"); read(s);
          openfile(1, s);
          selectinput(0); selectoutput(1);
          while not iochan(0) and %100 do
               begin insymbol(x);
                    if x>64 and x<91 then x := x+32;
     comment the ASCII codes for the upper case letters are the decimal
     numbers 65-90, while those for the lower case letters are 97-122;
                    outsymbol(x);
     comment letters within brackets will not be lowered;
                    if x=$.[. then begin depth := 1;
                                   while not (x=$.]. and depth = 0 do
                                        begin insymbol(x);
                                             outsymbol(x);
                                        if x=$.]. then depth := depth-1;
                                        if x=$.[. then depth := depth+1
                                        end
                              end;
     comment nor will ASCII constants;
                    if x=$.$. then begin insymbol(x); outsymbol(x); y := x+1;
                                   while y#x do begin insymbol(y);
                                                      outsymbol(y)
                                                end
                              end
               end
     end
```

(Apart from the underlining, and the two occurrences of the acronym 'ASCII' which have been printed in upper case, this programme is in the form of output from itself. It accepts the names of the input and output files from the teletype, in string form, that is, enclosed in quotation marks.)

Exercise 11.5

1.
```
begin  real x;  integer n;
       real procedure rand(a, b, starter);
comment the first call to this procedure should have starter an odd
integer <67,108,864.    To obtain a sequence of random numbers,
subsequent calls should have starter=0.  The numbers are rectangularly
distributed over the interval [a,b];
       value a, b;  real a, b;  integer starter;
       begin own integer m;  integer i;
             if starter≠0 then m := starter;
             for i := 125, 25 do m := i*m rem 67108864;
comment this avoids possible overflow on the PDP10 through multiplying
by 3125 in one step;
             rand := m/67108864*(b-a)+a
       end;
comment the next two statements indicate how to use this procedure;
       print(rand(0, 1, 234567));
       for n := 1 until 10 do  print(rand(0, 1, 0))
end
```

2.
```
begin . . .
       real procedure leg(x, which);
       value x, which; real x; integer which;
       begin real y;
             switch choose := P0, P1, P2, P3, P4;
             goto choose[which];
             y := 0; goto last;
       P0 :  y := 1; goto last;
       P1 :  y := x; goto last;
       P2 :  y := (3*x*x-1)/2;  goto last;
       P3 :  y := (5*x*x-3)*x/2;  goto last;
       P4 :  y := ((35*x*x-30)*x*x+3)/8;
       last : proc := y
       end;
       .
       .
       .
end
```

APPENDIX 1

BACKUS NORMAL FORM

The syntax of ALGOL is specified in metalinguistic formulae. The structure of these is called Backus Normal Form or Backus-Naur Form (BNF).

< digit > :: = 0|1|2|3|4|5|6|7|8|9

is one such formula which defines(::=) the syntactic entity 'digit' (indicated by pointed brackets) to be either 0 or 1 or 2 or ... or 9 (the | denotes alternation). Similarly, we have

< letter > ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|A|B|C|D|
E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

In terms of these, the following formula.

< identifier > ::= <letter>|<identifier><letter>|<identifier><digit>

defines an identifier to be either a letter, or an identifier followed by a letter, or an identifier followed by a digit. (The juxtaposition of two entities on the right hand side of such a formula signifies concatenation of the sequences denoted.) This is a recursive definition which is 'earthed' by the first of the 3 alternatives. It corresponds to the verbal specification: "An identifier is a string of letters and digits, beginning with a letter". (The upper limit of 64 on the length of the string is a machine-dependent factor, and not 'pure' ALGOL.)

Starting in this way, the whole structure may be built up.

For example, assuming the definitions of < unsigned number >, < variable > (which includes subscripted and unsubscripted variables), and < function designator > (i.e. a procedure call), the syntax of arithmetic expressions is as follows:

< adding operator > ::= +|-

< multiplying operator > ::= X|/|÷

< primary > ::= <unsigned number > | < variable > | < function designator > | (< arithmetic expression>)

< factor > ::= < primary > | < factor > ↑ < primary >

< term > ::= < factor > | < term > < multiplying operator > < factor>

< simple arith. exp. > ::= < term > | < adding operator > < term > |
< simple arith. exp.> < adding operator><term>

< if clause > ::= if < Boolean exp. > then

< arithmetic expression > ::= < simple arith. exp.> | < if clause >
< simple arith. exp.> else < arithmetic expression >

Even without the second alternative in the last line, this definition is recursive becuase of the occurrence of < arithmetic expression> in the third line. The implication of the fourth alternative in the third line (which is a sequence of 3 entities, namely a '(', an < arithmetic expression >, and a ')') is that the expression between a left parenthesis and the matching right parenthesis is evaluated by itself, and this value used in subsequent calculations. Also built in to this definition is the hierarchy (↑ ; X, /, ÷; +, -) of arithmetic operators.

APPENDIX 2

'ALGOL-Like' Languages

Some languages are described as 'ALGOL-like'. The major characteristics of ALGOL which should be taken into account when making such a description are ([4]):

(i) use of Backus-Naur Form(BNF) to describe syntax, with semantics in English;

(ii) acceptance of a large amount of mathematical notation, and the elimination of compiler-directed restrictions;

(iii) distinction between the imperative or assignment operator (:=) and the predicative or relational operator (=);

(iv) use of specially distinguished English words for new symbols;

(v) page layout at the discretion of the programmer.

APPENDIX 3

ERROR MESSAGES

0. PROBABLY SEMICOLON OMITTED
1. UNDECLARED IDENTIFIER
2. INCORRECT STATEMENT
3. INCORRECT EXPRESSION
4. PROBABLY OPERATOR OMITTED
5. IDENTIFIER OR CONSTANT MISSING
6. INCORRECT DESIGNATIONAL EXPRESSION
7. INCORRECT OR UNPARENTHESIZED ASSIGNMENT
8. SYMBOL NOT PERMITTED HERE
9. AMBIGUOUS USE OF COLON
10. SEMICOLON PROBABLY SUPERFLUOUS
11. ONLY LETTER STRING ALLOWED
12. THIS DELIMITER IS NOT PERMITTED BEFORE DO
13. WHILE STATEMENT IS NOT ALLOWED BETWEEN THEN AND ELSE
14. THEN MUST NOT BE FOLLOWED BY IF
15. THEN STATEMENT NOT FOUND
16. DECLARATIONS MUST BE TERMINATED BY SEMICOLON
17. THIS IS NOT ALLOWED AFTER END
18. CANNOT BE USED AS ARGUMENT
19. ARGUMENT TOO LARGE
20. PROBABLY END OMITTED
21. COMPLEX ARITHMETIC NOT IMPLEMENTED
22. DECLARATOR LONG MUST BE FOLLOWED BY REAL
23. NOT PERMITTED AS SPECIFIER
24. NOT PERMITTED AS SPECIFIER
25. INCORRECT DECLARATION OR SPECIFICATION
26. NO DECLARATION SHOULD FOLLOW PROCEDURE DECLARATION
27. IMPROPER ARRAY DECLARATION OR SPECIFICATION
28. DELIMITER MUST NOT BE DECLARED OR SPECIFIED
29. PROBABLY LIST ELEMENT MISSING
30. IMPROPER DECLARATION
31. VALUE WAS ALREADY SPECIFIED
32. IMPROPER TYPE OF FORMAL IN VALUELIST
33. NON-FORMALS MUST NOT BE SPECIFIED
34. BOUND PAIR NOT FOUND
35. INCORRECT BOUND PAIR
36. PROBABLY RIGHT BRACKET OMITTED

37. TYPE DOES NOT MATCH FORWARD DECLARATION
38. := MISSING IN SWITCH DECLARATION
39. PROCEDURE NESTING TOO DEEP
40. NOT SIMPLE IDENTIFIER IN FORMAL LIST
41. FORMAL LIST NOT PROPERLY TERMINATED
42. PROBABLY ; MISSING IN PROCEDURE HEADING
43. NOT ALL FORMALS HAVE BEEN SPECIFIED
44. INCORRECT STRUCTURED FORMAL LIST
45. UNTIL EXPRESSION NOT FOUND
46. BYTE SELECTOR NOT PERMITTED AS CONTROLLED VARIABLE
47. ASSIGNMENT DELIMITER NOT FOUND
48. DELIMITER "DO" NOT FOUND
49. ASSIGNMENT HAS NON-MATCHING TYPES
50. PROBABLY DELIMITER OMITTED
51. CANNOT BE USED AS UNARY OPERATOR
52. CANNOT BE USED AS BINARY OPERATOR
53. THEN EXPRESSION NOT FOUND
54. CONDITIONAL EXPRESSION MUST HAVE ELSE PART
55. CONDITIONAL EXPRESSION MUST BE PARENTHESIZED
56. IMPROPER SEQUENCE OF DELIMITERS
57. WRONG NUMBER OF DIMENSIONS
58. TOO MANY PARAMETERS FOR STANDARD FUNCTION
59. NON-TYPE PROCEDURE AS EXPRESSION
60. MATCHING CLOSE PARENTHESIS NOT FOUND
61. INCORRECT NUMBER OF ACTUAL PARAMETERS
62. FORWARD HAS NO MATCHING DECLARATION IN SAME BLOCK
63. FORWARD DECLARATION NOT ALLOWED FOR THIS TYPE
64. FORWARD DECLARATION WAS REQUIRED
65. ASSIGNMENT HAS NON-MATCHING TYPES
66. STACK ADDRESS OVERFLOW
67. NON-INTEGER OPERAND FOR DELIMITER REM OR DIV
68. COMPLEX ARITHMETIC NOT IMPLEMENTED
69. NON-ARITHMETIC OPERAND FOR ARITHMETIC OPERATOR
70. NON-ARITHMETIC OPERAND FOR RELATIONAL OPERATOR
71. NON-BOOLEAN OPERAND FOR BOOLEAN OPERATOR
72. DELIMITER NOT REQUIRES BOOLEAN OPERAND
73. UNARY + AND - REQUIRE ARITHMETIC OPERAND
74. OVERFLOW WHILE COMBINING CONSTANTS
75. OVERFLOW IN LONG REAL OPERATION ON CONSTANTS
76. UNDEFINED RESULT FOR POWER OPERATION ON CONSTANTS
77. PARAMETER FOR STANDARD FUNCTION MUST BE ARITHMETIC

78. STANDARD FUNCTION "INT" REQUIRES BOOLEAN PARAMETER
79. STANDARD FUNCTION "BOOL" REQUIRES ARITHMETIC PARAMETER
80. PARAMETER MUST BE OF ARITHMETIC TYPE
81. FOR STATEMENT NOT ALLOWED BETWEEN THEN AND ELSE
82. SWITCH IDENTIFIER NOT ALLOWED HERE
83. EXPRESSION TOO LONG
84. IDENTIFIER DECLARED IN THIS BLOCK NOT PERMITTED
85. INCORRECT FILE OR BLOCK STRUCTURE
86. INCORRECT BLOCK STRUCTURE; TOO MANY ENDS
87. PROCEDURE INCORRECTLY TERMINATED
88. INCORRECT FILE STRUCTURE
89. EMPTY SOURCE FILE
90. INVALID CONSTANT
91. IDENTIFIER EXCEEDS 64 CHARACTERS
92. IMPROPER WORD DELIMITER
93. CHARACTER NOT LEGAL IN ALGOL
94. EXPONENT TOO SMALL
95. EXPONENT TOO LARGE
96. DECLARATION FOLLOWS STATEMENT
97. IMPROPER USE OF PERIOD
98. INTEGER CONSTANT CONVERTED TO TYPE REAL
99.
100. EXPECTED WAS AN EXPRESSION
101. EXPECTED WAS A STATEMENT
102. EXPECTED WAS A BOOLEAN EXPRESSION
103. EXPECTED WAS A DESIGNATIONAL EXPRESSION
104. EXPECTED WAS A LABEL IDENTIFIER
105. EXPECTED WAS A LABEL IDENTIFIER
106. IDENTIFIER MUST NOT BE DECLARED OR SPECIFIED TWICE
107. BOUND PAIR EXPRESSION MUST BE ARITHMETIC
108. IDENTIFIER USED TWICE IN FORMAL LIST
109. STEP EXPRESSION MUST BE ARITHMETIC
110. UNTIL EXPRESSION MUST BE ARITHMETIC
111. INITIALIZING EXPRESSION MUST BE ARITHMETIC
112. CANNOT BE USED AS CONTROLLED VARIABLE
113. IMPROPER LEFT PART OF ASSIGNMENT
114. EXPECTED WAS A STRING EXPRESSION
115. EXPRESSION OF IMPROPER TYPE
116. EXPRESSION OF IMPROPER TYPE

117. EXPECTED WAS AN ARRAY IDENTIFIER

118. SUBSCRIPT MUST BE ARITHMETIC EXPRESSION

119. EXPECTED WAS A SWITCH IDENTIFIER

120. IMPROPER ACTUAL PARAMETER

121. EXPECTED WAS A PROCEDURE IDENTIFIER

122. SUBSCRIPT OF SWITCH DESIGNATOR MUST BE ARITHMETIC

123. VARIABLE OF IMPROPER TYPE

124. EXPECTED WAS A STRING VARIABLE

125. PARAMETER IN STANDARD FUNCTION HAS INCORRECT TYPE

APPENDIX 4

ASCII CHARACTER CODES

The following table lists the octal values of the characters used. For example, N has the value $116_8 = 1001110_2 = 78_{10}$

Character	ASCII 7-Bit	Character	ASCII 7-Bit	Character	ASCII 7-Bit
Horizontal Tab	011	:	072	Z	132
Line Feed	012	;	073	[	133
Form Feed	014	<	074	\	134
Carriage Return	015	=	075	]	135
Space	040	>	076	↑	136
!	041	?	077	←	137
"	042	@	100	`	140
#	043	A	101	a	141
$	044	B	102	b	142
%	045	C	103	c	143
&	046	D	104	d	144
'	047	E	105	e	145
(	050	F	106	f	146
)	051	G	107	g	147
*	052	H	110	h	150
+	053	I	111	i	151
,	054	J	112	j	152
-	055	K	113	k	153
.	056	L	114	l	154
/	057	M	115	m	155
0	060	N	116	n	156
1	061	O	117	o	157
2	062	P	120	p	160
3	063	Q	121	q	161
4	064	R	122	r	162
5	065	S	123	s	163
6	066	T	124	t	164
7	067	U	125	u	165
8	070	V	126	v	166
9	071	W	127	w	167
		X	130	x	170
		Y	131	y	171

Character	ASCII 7-Bit
z	172
{	173
|	174
}	175
~	176
Delete	177

INDEX

(Where a topic occurs on successive pages of a chapter, only the first page is listed.)

ABS, 19
Ackerman's function, 57
ALGOL, 1,2,88
    characters, 3
    hardware language, 3
    publication language, 3
    reference language, 3
<u>And</u>, 5,9
ARCCOS, 19
ARCSIN, 19
ARCTAN, 19
Arguments, 43
<u>Array</u>, 3,31,45,60
    , dynamic, 40,64
    , own, 64
ASCII code, 22,93
    constant, 7,28
Assignment, 11,88
Autocode, 1

Backus, 86,88
<u>Begin</u>, 3,12,17
Block, 16,38,64
<u>Boolean</u>, 3,24,45
    constant, 7
    expression, 8,13,29,35,65
    variable, 10,24
Bracket, 3,7
BREAKOUTPUT, 27
Buffer, 22,27
Byte processing, 28
    string, 60

Call, 49
Channel, 21
Checkoff, 31
Checkon, 31
CLOSEFILE, 23
<u>Comment</u>, 3,5,17
Compiler, 67
Compound statement, 16,38
Conditional expression, 8,15
Conditional statement, 13,37
Control variable, 34
COPY, 60
COS, 19
COSH, 19

Debugging, 67
Declarations, 31,42,46,57
Declarator, 3
DELETE, 61
Delimiter, 3,4
Designational expressions, 65
Devices, 21
DIM, 32
<u>Div</u>, 5
<u>Do</u>, 3,34
Dummy statement, 16
Dynamic storage allocation, 38

<u>Else</u>, 3,13,17
<u>End</u>, 3,12,17
End-of-file, 29
ENTIER, 19
<u>Eqv</u>, 5,9
EXP, 19
Expressions, 8
External, 57

False, 3,4,7,9,24
Field manipulations, 61
File devices, 23
names, 23
specification, 23
For, 3
statement, 16,34
FORTRAN, 1,5,8,13,16,30,34,39,42,55
Forward references, 46, 57
Functions, 19,47

GFIELD, 61
Global, 38,48,64
Goto, 3,12
Identifier, 4,6,8,12,48,86
If, 3,13,17
Image binary mode, 22
Image mode, 22
IMAX, 20
IMIN, 20
Imp, 5,9
INPUT, 21
INSYMBOL, 28,61
Integer, 3,6,24,45
IOCHAN, 29
Iteration, 55

Label, 3,12,36,44, 65
LARCTAN, 20
LB, 32
LCOS, 20
LENGTH, 61
LEXP, 20
LINK, 61
LINKR, 61
LLN, 20
LMAX, 20
LMIN, 20
LN, 19

Local, 38,48,62,64
Long real, 6,7,24,45
LSIN, 20
LSQRT, 20

Mode, 22

Name, 49
, simple, 53
Nesting, 37,38
NEWLINE, 26
NEWSTRING, 60
NEXTSYMBOL, 28
Not, 5,9

Octal constant, 7,8,9
OPENFILE, 23
Operator, 3
hierarchy, 8,87
precedence, 8
, relational, 9
Or, 5,9
OUTPUT, 21
OUTSYMBOL, 28
Own variables, 3,64

PAGE, 26
Parameter, 43
PRINT, 24
Procedure, 3,42
body, 42
call, 42
declaration, 42
heading, 42
Programme layout, 16, 88
Pushdown list, 41

READ, 24
Real, 3,6,24,45

Recursion, 55
Reference, 53, 86
RELEASE, 22
Rem, 5
Reserved word, 4
RMAX, 20
RMIN, 20

Scope, 38
SELECTINPUT, 22
SELECTOUTPUT, 22
Separator, 3
SFIELD, 61
Side-effect, 54
SIGN, 19
SIN, 19
SINH, 19
SKIPSYMBOL, 28
SPACE, 26
Spacing, 16,26, 88
Specifier, 44
SQRT, 19
Stack, 41,55
Statement, 11
Step, 3,34
String, 3,6,24,26,45,60
    constant, 7
    variable, 28
Subscript, 32
Switch, 3,44,64
Syntax, 86

TAB, 26
TAN, 19
TANH, 19
Teletype handling, 27
Then, 3,13,17
TRAP, 68
True, 3,4,7,9,24
Type, 6,11,31,47

UB, 32
Until, 3,34

Validation of data, 29
Value, 3,50

While, 3
    statement, 16,29,34
WRITE, 26,61