Edinburgh Regional Computing Centre

Edinburgh ALGOL Language Manual

A description of the ALGOL 60 Language as implemented by ERCC

First Edition

June 1976

EDINBURGH REGIONAL COMPUTING CENTRE

Edinburgh ALGOL Language Manual

This manual describes the algorithmic language ALGOL 60 as implemented by ERCC. It is primarily intended as a reference manual, not as a beginner's guide to programming.

© 1976 Edinburgh Regional Computing Centre

PREFACE

This Manual describes the algorithmic language ALGOL 60 as implemented on the Edinburgh Multi-Access System (EMAS). It is intended as a reference manual, not as a beginner's guide to programming. The book falls into four parts:

Chapter one gives an introduction to the language - its origins and basic concepts.

Chapters two to six describe the elements of the language.

Chapters seven to eleven deal with input and output of data and other areas which are specific to the hardware environment of the language compiler.

Chapter twelve consists of the full ALGOL 60 report as published by the originators of the language, preceded by some explanatory notes.

This Manual is the work of many people in the Edinburgh Regional Computing Centre. Particular mention should be made of Anne Tweeddale who typed the manuscript, and the staff of the Reprographics section who printed the Manual.

Felicity Stephens,
September 1975.

CONTENTS

Chapter	Title	Page
1	Basic Concepts	3
2	Simple Arithmetic or Boolean Expressions	9
3	Statements	15
4	Block Structure and Declarations	25
5	Procedures	35
6	Standard Functions	51
7	Hints on Program Optimisation	53
8	Input and Output	55
9	Hardware Representation	67
10	Program Segmentation	71
11	Compiler Messages and Diagnostics	75
12	Qualifications to the ALGOL Report for EMAS ALGOL	85
Appendix	The ALGOL Report	

CHAPTER 1 BASIC CONCEPTS

ALGOL (ALGOrithmic Language) was developed between 1957 and 1960 as a language suitable for scientific and mathematical data processing. A definition of the language was produced by the originators, but was found to contain a number of ambiguities and obscurities. A revised definition, therefore, was published in 1963, entitled 'Revised Report on the Algorithmic Language ALGOL 60' and is reproduced in this manual in Appendix 1. In the course of time, further clarifications and refinements of the language have become customarily accepted and a description of some of these was published in 'ALGOL Bulletin' in 1974.

The language defined by the report is known as the 'reference language'. A 'hardware representation' of the language must also be distinguished; that is, the way in which the basic symbols of the language are presented to the compiler. The hardware representation depends on the character set of the computer in use and may contain any restrictions to the full range of characters specified in the reference language. For example, the EMAS hardware representation of keywords is of the form %NAME, where NAME stands for any keyword and is always in upper case. This manual follows the notation of the reference language as far as typographically possible. The hardware representation is described in Part 3.

A SIMPLE ALGOL PROGRAM

```
begin integer X, Y, Z;
      Y:=READ;
      Z:=READ;
      X:=Y + Z;
      PRINT(X, 3, 0)
end
```

The simple program above reads in two numbers, adds them together, and prints out their sum. The body of the program is enclosed by the brackets begin and end. The program is made up of a series of statements separated by semi-colons, and is headed by a declaration.

The declaration

```
integer X, Y, Z
```

states that three variables X, Y and Z can take any integer value during the running of the program.

The variables Y and Z are given values by the statements

```
Y:=READ
Z:=READ
```

which read two numbers from the default input device, normally the on-line terminal. The first value read is assigned to Y, and the next to Z.

The statement:

```
X:=Y + Z
```

is read as 'X becomes equal to Y plus Z'. This statement is an algorithm used to solve the problem of adding two numbers together.

The statement:

```
PRINT(X, 3, 0)
```

causes the value of X to be printed on the on-line terminal.

BASIC SYMBOLS

ALGOL programs are written using the following basic symbols:

1. Alphanumeric characters, which are used to form identifiers.

2. Delimiters, which are subdivided into:-

   Operators
   Separators
   Brackets
   Declarators
   Specificators

   For instance, in the above example of an ALGOL program, the operator '+' was used to denote addition.

3. The logical values true and false.

DECIMAL NUMBERS

All numerical variables used in ALGOL programs must be either of type real or of type integer. Signed and unsigned numbers using the digits 0 to 9 may be written in ALGOL and have their ordinary meanings. Signed numbers may be used in EMAS ALGOL for the input of data and for the output of results.

INTEGER NUMBERS

Numbers of type integer are integer numbers in the ordinary sense. An integer number used in EMAS ALGOL must lie in the range -2147483648 to 2147483647 inclusive and must not contain a decimal point or a comma.

The examples which follow show the correct use of integer numbers.

Number	Notes
0	The plus sign is optional for a positive integer
16	
+16	
-16	
10 287 132	Commas must not be inserted; if spaces are inserted for clarity they are ignored by the compiler

REAL NUMBERS

All numbers which are not of type integer must be of type real. A real number will thus, in general, consist of a fractional part and an integral part. Real numbers may be written using one of the two methods described below:

1. A decimal point may be used to separate the integral and fractional parts. If a decimal point is used, it must be followed by at least one digit. If the fractional part of the number is zero, both the decimal point and the fractional part may be omitted but the number then becomes of type integer. If the integral part is zero, it may be written as 0 or omitted.

2. A decimal number and an integer exponent separated by the symbol $_{10}$ may be used as in the example below:

   $$2.0_{10}\ -4$$

   The decimal number and integer exponent must conform to the rules described above. The decimal number may be omitted if required, whereupon 1 will be assumed.

Numbers of type real must lie in the range $-7*10^{75}$ to $7*10^{77}$. A number whose modulus is smaller than $7*10^{-77}$ is taken as zero.

The maximum working accuracy using EMAS ALGOL is sixteen significant decimal figures.

e.g. 0, 0.0 — Alternative ways of specifying zero

73.5, +73.5 — Plus sign is assumed if not given

.5, 0.5 — Zeros before the decimal point are not required

10^{10}, $1_{10}10$, $1.23_{10}-15$ — Examples of the use of real numbers with exponents, the first two examples being identical

IDENTIFIERS

ALGOL identifiers are used to name simple variables, arrays, labels and procedures.

The rules governing the choice of identifiers are:

1. A letter followed by any sequence of alphanumeric characters may be used. Spaces will be ignored by the compiler.

2. Identifiers may be of any length but only the first 255 characters are significant.

   e.g. ALPHA, B3, TEMPI, J, K2NUM, L5J6

NAMING SIMPLE VARIABLES

The programmer must declare each variable in his program to be of type real, integer or Boolean. For example:

```
real ALPHA, BETA
```

IDENTIFIERS USED AS ARRAY NAMES

An array is simply an ordered set of variables of the same type, one identifier being used to name the whole array. Array identifiers must follow the rules for naming simple variables described above. Particular variables (or 'elements')

of an array are referred to by writing the array identifier followed by one or more subscripts enclosed in square brackets. Such variables are called subscripted variables and may appear in the program wherever the use of a simple variable is permitted.

Arrays can have one or more dimensions. For example, the algebraic variables R_1, R_2,....,R_n could be represented in ALGOL by the one-dimensional array R[1], R[2],....,R[n].

The 3 x 3 matrix

$$\begin{matrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ X_{31} & X_{32} & X_{33} \end{matrix}$$

is a two-dimensional array with nine elements. The ALGOL subscripted variables corresponding to the elements of this array are as follows:

```
X[1,1]    X[1,2]    X[1,3]

X[2,1]    X[2,2]    X[2,3]

X[3,1]    X[3,2]    X[3,3]
```

The subscripts used to reference elements of a given array need not necessarily be positive. For example:

```
A[-1,-2,-3]
A[-1,0,6]
```

could refer to elements of a three-dimensional array A.

Indeed, any valid arithmetic expression may be used for subscripts. For example:

```
A[I,J]
X[I+K,J*L]
```

LABELS AND SWITCH DESIGNATORS

Normally, statements in an ALGOL program are obeyed sequentially, but it is sometimes necessary to transfer control to another point in the program. Any statement can be labelled by being prefixed with an identifier, hitherto undeclared, followed by a colon. For example:

```
LABEL1: X:= Y + Z
```

The goto statement can be used to transfer control to any labelled statement. For example:

```
goto LABEL1
```

Switches, which are in effect arrays of labels, can also be used for

transferring control. These are described in Chapter 4.

COMMENTS

Comments should be used by the programmer to record information which will enable others (and perhaps himself) to understand the workings of an ALGOL program.

The following are treated as comments:

1. Any sequence of basic symbols commencing with comment and terminated by a semicolon when comment follows either begin or a semicolon.

2. Any sequence of basic symbols following end and terminated by a semicolon, end or else.

Note: Certain special rules apply to comments in the specification parts of procedure headings (see Chapter 5), where a comment immediately following the specification has a particular function.

The simple program at the beginning of this chapter is reproduced here with some illustrative comments included:

```
begin comment FIRST DECLARE THREE VARIABLES;
      integer X, Y, Z;
      Y:= READ; comment READ IN 1st NUMBER;
      Z:= READ; comment AND THE 2nd NUMBER;
      X:= Y + Z;
      PRINT(X,3,0); comment PRINT THE ANSWER
      WITH THREE PLACES BEFORE THE DECIMAL POINT AND NO FRACTIONAL PART
end OF A VERY SIMPLE PROGRAM;
```

STRINGS

It is sometimes desirable to annotate output with headings or titles. To facilitate this, one of the elements provided in ALGOL is the string. A string is defined as any sequence of basic symbols enclosed by the string quotes [and]. For example:

```
[THE_ANSWER_EQUALS]
[EMAS_ALGOL]
```

N.B. Since spaces are of no significance in an ALGOL program, it is necessary to use the symbol _ to indicate a space required in a string.

CHAPTER 2 SIMPLE ARITHMETIC & BOOLEAN EXPRESSIONS

In ALGOL, two types of expression can be distinguished:

1. Arithmetic expressions

2. Boolean expressions

An arithmetic expression is a rule for computing a numerical value: for example X + Y.

A Boolean expression is a proposition which can take one of the logical values true or false. The evaluation of arithmetic and Boolean expressions forms a major part of any algorithm represented by an ALGOL program.

SIMPLE ARITHMETIC EXPRESSIONS

Numbers, identifiers and arithmetic operators may be combined to form simple arithmetic expressions. Simple arithmetic expressions may also include conditional arithmetic expressions enclosed in parentheses, and function designators.

ARITHMETIC OPERATORS

Any of the operators +, -, *, ÷, / and ↑ may be used in simple arithmetic expressions. + and - have their normal mathematical meanings, while * represents the arithmetic multiplication operator × .

DIVISION

Two operators are used in ALGOL for division. The operator / may be used with variables and constants of type real and type integer. The operator / produces a result of type real irrespective of the types of dividend and divisor; for example:

$$X/Y \text{ signifies } \frac{X}{Y}$$

$$5.7/P \text{ signifies } \frac{5.7}{P}$$

The operator ÷ may only be used for division with variables of type integer, and produces a result of type integer. The formal definition of the process is:

```
X ÷ Y = sign (X/Y)* entier (abs (X/Y))
```

where sign, entier and abs are ALGOL standard functions as described in Chapter 6. The result of the division consists of a quotient whose sign is determined algebraically, and a remainder which is discarded.

e.g. $9 \div 2 = 4$
$100 \div 15 = 6$
$(-9) \div 2 = -4$

EXPONENTIATION

The operator ↑ is the sign of exponentiation. The base must precede the sign and both the base and the exponent may be either a number or an identifier. No values of base or exponent are allowed which would lead to infinite, indeterminate or imaginary results. When the exponent is of type real, the value of the base may not be negative.

The result of exponentiation is always real, unless the base is of type integer and the exponent is a positive constant, when the result is of type integer. For example:

A↑B signifies A^3 and will be of type real even if A and B are integers.

I↑3 signifies I^3 and will be of type integer if I is of type integer.

I↑3.5 signifies $I^{3.5}$ and will be of type real even if I is of type integer.

Note that this is slightly at variance with the definition given in the ALGOL Report (see Chapter 12).

OPERATOR PRECEDENCE

Simple arithmetic expressions are formed by combining identifiers, constants and arithmetic operators.

Arithmetic operations are executed in order of occurrence from left to right, observing the order of precedence below:

↑	Highest precedence
* / -	
+ -	Lowest precedence

Thus the sequence of evaluation of an arithmetic expression is as follows:

1. The expression is scanned from left to right. Each exponentiation is executed as it is encountered, the operators * / ÷ + and - being ignored.

2. The expression is again scanned from left to right. Operations involving * / or ÷ are executed in the order in which they are encountered, the operators + and - being ignored.

3. The expression is scanned once more, again from left to right. Operations involving either + or - are executed in the order in which they are encountered.

As could be inferred from the sequences given above, two arithmetic operators may not appear next to each other. Wherever this would otherwise occur parentheses must be used. Parentheses can also be introduced into an expression to enforce a specific order of evaluation.

Note: The introduction of parentheses into an expression does not alter the precedence of the operators involved, but merely the order of evaluation. An expression containing parentheses is evaluated from the innermost parentheses outwards, the contents of each pair of parentheses being considered as a separate arithmetic expression.

It is strongly recommended that parentheses be used in all cases where there is any doubt as to the order of evaluation, since redundant parentheses are ignored.

The following examples have a twofold purpose:

1. To compare ALGOL expressions with ordinary algebraic expressions.

2. To show how the order of evaluation of simple arithmetic expressions is altered by the use of parentheses.

ALGOL Expression	Algebraic Expression
A - B + C	(A - B) + C
A - (B + C)	(A) - (B + C)
A/B * C	(A/B) * C
A/(B * C)	(A)/(B * C)
A↑B * C	(A^{B}) * C
A↑(B * C)	$A^{(B*C)}$

SIMPLE BOOLEAN EXPRESSIONS

The result of a Boolean expression is either the logical value true or the logical value false. Simple Boolean expressions are formed using the relational and logical operators described in the following two sections.

USE OF RELATIONAL OPERATORS

The relational operators are as follows:

<	Less than
≤	Less than or equal to
=	Equal to
≥	Greater than or equal to
>	Greater than
#	Not equal to

A simple Boolean expression may be one of the following:

1. Either of the logical values true or false.

2. A variable or function designator (see Chapter 5) of type Boolean, which may have either of the logical values true or false (type declarations are described in Chapter 4).

3. A single relation, where a relation is defined as two simple arithmetic expressions separated by a relational operator.

4. A more complex form involving relations between variables of type Boolean and logical values. The values are operated on by the logical operators described below. For example:

   assuming the declaration

   integer A,B,C,P; real X; Boolean AB

false	A logical value
A	A Boolean variable which may be either true or false
B = 0	true if B = 0, otherwise false
X↑2<4	true for -2<X<2, otherwise false
A-P>6*C	true if A>6*C+P, otherwise false

LOGICAL OPERATORS

The logical operators listed below may be used in simple Boolean expressions:

equiv	is equivalent to
impl	implies
and	and
or	or
not	not

If B and C are any two Boolean primaries, the effect of these operators is as follows:

not The value of not B is false if B is true and true if B is false.

and The value of B and C is true if both B and C are true; otherwise, it is false.

or The value of B or C is true if either B or C is true; otherwise, it is false.

impl The value of B impl C is true if B is false (regardless of C), or if C is true (regardless of B); otherwise, it is false.

equiv The value of B equiv C is true if both B and C have the same value; otherwise, it is false.

A summary of these effects is given in the following table:

B	false	false	true	true
C	false	true	false	true
not B	true	true	false	false
B and C	false	false	false	true
B or C	false	true	true	true
B impl C	true	true	false	true
B equiv C	true	false	false	true

ORDER OF EVALUATION OF SIMPLE BOOLEAN EXPRESSIONS

1. Arithmetic and relational operations are evaluated in accordance with the rules given above.

2. Logical operators are evaluated in the order given by the list of operators below:

   not Highest precedence

   and

   or

   impl

   equiv Lowest precedence

As with arithmetic expressions, parentheses can be inserted for clarity or to alter the order of evaluation of Boolean expressions given by the above list. Note that, in general, two logical operators must not appear next to each other. The one exception to this rule concerns the operator not, which may immediately follow one of the other logical operators:

e.g. A and not B

Other examples are:-

A > 4 * B↑2 - 1 true if A > 4 * B↑2 - 1

not (F > P and P > Z) true if F ≤ P or P ≤ Z

CHAPTER 3 STATEMENTS

ALGOL programs consist of a series of statements separated by semi-colons. The ALGOL Report distinguishes three different kinds of statement, as follows:

1. Unconditional statements, which can be further subdivided into

   (a) Unlabelled basic statements, which can be

      (i) Assignment statements
      (ii) goto statements
      (iii) Procedure statements

   (b) Compound statements

   (c) Blocks

2. Conditional statements

3. for statements

Semi-colons are used in ALGOL to separate statements; they are not themselves part of the statements. They are only needed when it is necessary to separate statements from each other: they are not, for example, required immediately following begin or immediately preceding end, since begin and end can be thought of as brackets, which do not therefore need to be separated from what they enclose.

Note however that it is not necessarily wrong to insert a redundant semi-colon: depending on the context it may be understood to be followed by a dummy statement. On the other hand it is necessary to understand what constitutes a statement, since a semi-colon inserted before the end of a statement will in general be wrong. In particular, for statements (see below) do not terminate at do, but after the ALGOL statement which follows do.

In the partial programs given as examples in this manual, in each case a semi-colon is not appended to the last statement of the example since it might not be necessary, depending on how the program continued.

The next section describes how statements are labelled. The sections following this describe assignment statements, conditional statements, conditional arithmetic and Boolean expressions, goto statements, for statements and compound statements in that order.

LABELS

Any undeclared identifier may be used as a label to prefix any ALGOL statement. The label and statement must be separated by a colon. A statement prefixed by a label is known as a basic statement. Any basic statement may be further prefixed by a label, so that the general form of a basic statement is:

```
lab1: lab2: ... :labn: statement
```

where lab1, lab2, ..., labn are labels and statement is any ALGOL statement.

Since labels may only be prefixed to statements it appears incorrect to place a

label on end. However ALGOL permits a dummy or null statement and the construction

```
    statement;
labl: end
```

is valid as labl is considered to be attached to a dummy statement after statement but before end.

ASSIGNMENT STATEMENTS

Assignment statements are used to assign the value of an expression to a variable. The variable may be of type integer, real or Boolean and the expression may have an integer value, a real value or a Boolean value. The rules for the compatibility of variables and expressions are given in the next section. The assignment statement

```
X := Y + Z
```

assigns the value of Y + Z to the variable X.

More complex forms of assignment statement are allowed, the general form being

```
var1 := var2 := ... := varn := expression
```

where var1, var2, ... ,varn represent variables of type real or type integer or type Boolean and expression is an appropriate arithmetic or Boolean expression. For example:

```
P:=Q:=0;
B1:=B2:=true
```

COMPATIBILITY OF VARIABLES AND EXPRESSIONS

Boolean expressions may only be assigned to variables of type Boolean. Integer expressions may be assigned to variables of type integer or of type real. In the latter case, the integer value of the expression is changed internally to a floating point number. Real expressions may be assigned to variables of type real or type integer. In the latter case, the real value of the expression is rounded to the nearest integer.

SUBSCRIPTED VARIABLES AND MULTIPLE ASSIGNMENT STATEMENTS

The sequence of evaluation of multiple assignment statements can become important when subscripted variables are used. Consider the following:

```
ARRAY [J] := J := 25
```

In all assignment statements of this type, the subscripts on the left hand side of a := are evaluated first, in sequence from left to right, then the value of the right-hand expression is evaluated and assigned to the left-hand variables. Thus, if J is initially 5, ARRAY [J] becomes ARRAY [5] and then J and ARRAY [5] are assigned the value 25.

CONDITIONAL STATEMENTS

A conditional statement, which may be labelled, must take one of the following forms:

1. (a) if bool then uncon

   (b) if bool then uncon else statement

2. if bool then forstatement

where bool is a Boolean expression, uncon an unconditional statement (which may be a compound statement - see later in this chapter), statement is any ALGOL statement, and forstatement is a for statement. For example:

1. if X = 0 then Y := 1

   This is the simplest form of conditional statement. If X = 0 then Y will be assigned the value 1. If X has any value other than 0, control will pass to the next statement in the program, and the statement Y := 1 will not be executed.

2. if X = 0 then Y := 1 else Y := 2

   In this case Y will be assigned the value 1 if X = 0. If X has any other value, Y will be assigned the value 2.

3. if X = 0 then Y := 1 else if X = 1 then Y := 2 else Y := 3

   Here, a further conditional statement has been added. If X = 0, Y will be assigned the value 1. If X = 1, Y will be assigned the value 2. For any other value of X, Y will be assigned the value 3.

4. if X < 0 then for Y := 1 step 1 until 3 do T [Y] := X + 1

   if X < 0 the for statement will be executed. If X has any other value, the for statement will be ignored and control will pass to the next statement in the program. for statements are described later in this chapter.

CONDITIONAL ARITHMETIC AND BOOLEAN EXPRESSIONS

Conditional arithmetic and Boolean expressions take a form similar to the conditional statement described above:

i.e. if booll then sarith else arith

if booll then sbool else bool2

where booll and bool2 stand for Boolean expressions and sbool for a simple Boolean expression; arith and sarith are an arithmetic expression and a simple arithmetic expression respectively. Simple arithmetic and Boolean expressions were described in Chapter 2. An arithmetic expression must be either a simple arithmetic expression or a conditional arithmetic expression. Similarly a Boolean expression must be either a simple Boolean expression or a conditional Boolean expression.

Consider the following examples:

1. Y := if X = 0 then 1 else 2

In this case, Y will be assigned the value 1 if X = 0. If X has any other value, Y will be assigned the value 2.

2. Y := if X = 0 then 1 else if X = 1 then 2 else 3

Here, if X = 0, Y will be assigned the value 1; if X = 1, Y will be assigned the value 2. For any other value of X, Y will be assigned the value 3. Note that these two examples demonstrate a more elegant programming technique than was shown earlier (examples 2 and 3 of Conditional Statements) whilst carrying out the same operations and producing identical results.

The conditional Boolean expression is very rarely required.

A more complex example, using subscripted variables, is given below; such complex examples are rarely used in practice:

3. S := A[I, if A[I,J] = 0 then 1 else B[L,M]]

goto STATEMENTS

The statements which form a program are normally executed in the order in which they are encountered. However, transfer of control from one statement to a non-consecutive statement can be accomplished using the goto statement. The statement to which control is transferred may be earlier or later in sequence than the goto statement which refers to it. The general form of the goto statement is:

goto desexp

where desexp stands for a designational expression, which is simply a rule for finding a label. The designational expression may be:

1. A simple designational expression, of the form

   (a) a label
   (b) a switch designator
   (c) a designational expression enclosed in parentheses.

2. A conditional designational expression, exactly comparable to the conditional arithmetic expression:

   if bool then sdesexp else desexp

   where bool, sdesexp and desexp stand for a Boolean expression, a simple designational expression, and a designational expression respectively.

The use of goto statements and labels is illustrated below:

```
LAB10: if N = 1 then goto LAB1
       else if N = 2 then goto LAB2
       else if N = 3 then goto LAB3
       else goto ERROR;
```

Also:

```
LAB10: goto if N=1 then LAB1 else if N=2 then LAB2 else if N=3 then LAB3
       else ERROR;
```

Whilst the use of a conditional statement in this way is perfectly acceptable, a better method uses a switch designator. The use and declaration of switches is described in Chapter 4.

for STATEMENT

The for statement enables the execution of any statement or combination of statements to be repeated a specified number of times.

The general form of the for statement is:

for var := flist do statement

where var stands for a variable, known as the controlled variable, flist for a for list and statement for any ALGOL statement, including another for statement or a compound statement. The controlled variable var must be a simple variable - it may not be a subscripted variable.

The for list is made up of a series of for list elements separated by commas. The general form of a for list is as follows:

fle,fle, ..., fle

where fle is a for list element.

A for list element may take one of three forms below:

1. An arithmetic expression

2. arith step arith until arith

3. arith while bool

where arith and bool stand for arithmetic and Boolean expressions respectively. The use of the separators step, until and while is described in more detail below. When the for statement is executed, the values of the expressions in the for list are consecutively assigned to the controlled variable, and the statement following do is then executed using the current value of the controlled variable.

Examples using the various types of for list element are given below:

1) arithmetic expression element

Suppose the sum of the expression

$$x + \sin(\text{abs}(\log(ax^2)))$$

is to be evaluated for X = -.75, -.37, .16, .39, .74.

The following for statement would achieve this provided Y were zero initially:

```
for X := -.75, -.37, .16, .39, .74 do
    Y := Y + X + SIN (ABS (LN (A*X*X)))
```

Any arithmetic expressions may be used as for list elements of this type.

2) step-until element

The two statements below compute the sum of ten elements of STO:

```
SUM := 0;
for I := 1 step 1 until 10 do
    SUM:= SUM + STO[I]
```

The statements are executed as follows:

1. SUM is assigned the value 0.

2. The value 1 is assigned to the controlled variable I and a check is made to ensure that the value of I is not greater than 10.

3. SUM is now assigned the value SUM + STO[1].

4. The value of I is now increased by one step: thus I now has the value 2. A check is again made to ensure that I is not greater than 10, and SUM is assigned the value SUM + STO[2].

5. I successively takes the values 3, 4, 5, 6, 7, 8, 9, 10 and SUM is assigned the value SUM + STO[I].

6. I is increased by one more step: I now has the value 11. The test shows

that I > 10; thus SUM := SUM + STO[I] is not evaluated and the execution of the statement is completed. Control now passes to the next statement in the program. The value left in the controlled variable I is undefined.

Any arithmetic expression may be used for the initial value of the controlled variable, the value of the step and the value of the limit in the step-until element. Negative steps are allowed, as are steps which change sign during the execution of the for statement. A complete specification of the actions of for statements is given in 'THE EXACT ACTION OF for STATEMENTS', below.

3) while element

It is often convenient to perform a loop until some function of the controlled variable goes outside certain bounds. This facility is provided by the third type of for list element, which has the general form:

arithmetic expression while Boolean expression

Suppose, for example, it is required to evaluate

$$\sum_{r=2}^{\infty} \frac{1}{(r^2+b)\,(r^2-1)}$$

neglecting all terms $<_{10} -10$. The section of program below, which forms a block, satisfies these requirements:

```
begin real TERM; integer R, R2;
      R := TERM := 1;
      SUM := 0;
      for R := R + 1 while TERM > 10 -10 do
      begin R2 := R*R;
          TERM := 1/((R2 + B) * (R2 - 1));
          SUM := SUM + TERM
      end
end
```

Here, the compound statement (see below) which forms the range is performed for R = 2, 3, 4, ... until TERM $<_{10} -10$.

Note the following points:

1. The values of expressions used in for list elements may be affected by a change of value of the controlled variable or by the execution of the range. Here, TERM is altered at each performance of the range.

2. The variable TERM must be given an initial value which satisfies the Boolean expression TERM $>_{10} -10$ or the range will not be executed at all.

3. The statement R2 := R*R is introduced to avoid the necessity of evaluating R*R twice in the calculation of TERM.

4. SUM and B are assumed to be declared elsewhere in the program.

THE EXACT ACTION OF for STATEMENTS

This section defines the action of EMAS ALGOL when executing a for statement, in terms of simpler ALGOL statements. This definition is only required when evaluation of the expressions may have side effects. The reader is warned that the ALGOL report does not completely define the action of for statements, and other compilers may act differently. Some users may prefer to omit this section on first reading.

The execution of the for statement:

```
for V := A step B until C do STAT
```

where A,B and C are arithmetic expressions and STAT is any statement, may be described by the following group of ALGOL statements:

```
        V := A;
        TEMP := B;
L:      if (V - C) * SIGN(TEMP) > 0 then goto NEXT;
        STAT;
        TEMP := B;
        V := V + TEMP;
        goto L;
NEXT:   V:= UNDEFINED; comment V becomes undefined at this point;
```

where SIGN is an ALGOL standard function; TEMP is a simple variable of the same type as V, used to ensure that B is evaluated only once during each traverse of the cycle; X and A, B and C are evaluated in the correct order. V is the controlled variable of the for statement. NEXT points to the next element in the for list, or, if the for list is exhausted, to the next statement in the program.

The execution of the for statement:

```
for V := A while B do STAT
```

where A is any arithmetic expression, B is any Boolean expression, and STAT any statement, may be described by the following group of ALGOL statements:

```
L3:     V := A;
        if not B then goto NEXT;
        STAT;
        goto L3;
NEXT:
```

When exit from a for list is made via a goto statement, the controlled variable retains its current value. The value of the controlled variable, however, is undefined after the completion of a step-until element or when the for list is exhausted. Thus:

```
for I := 1 step 1 until 10, I + 1 while
        A [I] > 0.001 do ....
```

is invalid as I is not defined on entering the while clause.

COMPOUND STATEMENTS

When writing a program, it is frequently necessary to treat a series of statements as a logical unit. To achieve this the statements are enclosed by the basic symbols begin and end, as in the following example:

```
begin LARGE := X;
      X := 0;
      if LARGE < 6 then goto LAB1
end
```

The resulting unit is known as a compound statement and has the following general form:

```
lab1: lab2: ...: labn: begin statement1 ;
                            statement2 ;
                            ......
                            statementn
                       end
```

where lab1, lab2, ..., labn are labels and statement1, statement2, ..., statementn are statements. Note that no semi-colon is necessary after statmentn as there is no further statement from which it need be separated.

The statements which form the compound statement may be any of the types of statement listed at the beginning of this chapter. In particular, each statement may itself be a compound statement, leading to a nested structure. This nesting of compound statements can be continued indefinitely, giving the following general structure:

```
begin S1;
      S2;
      begin S3;
            S4;
            ...
            begin S6;
                  ...
                  Sn;
                  ...
            end
      end
end
```

where S1, S2, ..., Sn are statements.

Compound statements are most frequently used after the do of for statements and after the then or else of conditional statements. Note that a semi-colon is not required to terminate the last statement before end.

Like any other statement, a compound statement may be labelled. A goto statement outside a compound statement may refer to a label within that compound statement, so long as the compound statement does not follow the do of a for statement. For example:

```
goto L1;
      ....
begin S := 1;
      L1:....
end
```

Note that there are no declarations following the begin of a compound statement. If declarations are present the resulting construction is not a compound statement but a block. Blocks are described in Chapter 4.

If a compound statement follows the do of a for statement, labels within the compound statement may only be referenced by goto statements within the same compound statement, although exits from the compound statement by means of a goto statement are allowed. For example:

```
goto ENTER;
.....
for I := 1 step 1 until 10 do
begin .....
      ENTER:
      .....
end
```

is incorrect, but

```
for I := 1 step 1 until 10 do
begin
      LAB:
      ...
      goto LAB;
      ...
end
```

is permissible.

CHAPTER 4 BLOCK STRUCTURE & DECLARATIONS

BLOCK STRUCTURE

Many ALGOL programs can be written as a single block, that is, a series of statements headed by one or more declarations and enclosed by begin and end. For example:

```
begin real A, B;
      integer C, D;
              A := B*C;
              .....
              C := C/D
end
```

The general form of a block is as follows:

```
labl: lab2:....: labn:
begin declarl; declar2; declarn;
      statementl;
      .
      .
      .
      statementn
end
```

where labl,...,labn are labels

declarl,...,declarn are declarations

statementl,...,statementn are statements, which may themselves be compound statements or blocks.

Note that declarations are separated by semi-colons, in the same way as statements, and that no semi-colon is necessary immediately before the end symbol.

Blocks can be nested; that is, each of the statements which is part of a given block can itself, by definition, be a block. This process of nesting can continue indefinitely, giving the following general structure:

```
begin comment block1;
      ....
      begin comment block2;
            ...
            ...
            begin comment blockn;
            ....
            end of blockn;
            ....
      end of block2;
      ....
end of block1
```

There are two main advantages to block structure:

1. The scope of variables and labels is restricted so that blocks correspond to logical units, enabling parts of a program to be written independently

of each other.

2. Economy of storage allocation is possible, as the storage for a block is allocated only when that block is entered, and is deallocated when that block is left. This means, for example, that the size of an array can be made to depend on the data which is input to a program, and also that several variables can use the same physical storage locations at different times during the exection of a program.

DECLARATIONS

Each identifier used to name a variable in a program must be declared at the head of a block. The declaration defines the type of variable that the identifier will represent within the block.

The rules governing the declaration of identifiers are as follows:

1. A declaration must be placed at the head of the block to which it applies.

2. All identifiers used must be declared except the following:

   (a) Those used as labels

   (b) Those which represent
   (i) ALGOL standard functions
   (ii) standard Input/Output functions

   If the user writes his own procedures for standard functions or for Input/Output, then these procedures must be declared even if they have the same names as the procedures normally supplied from the standard procedure library.

The declarators integer, real, Boolean, array, integer array, real array, Boolean array, own and switch are described in the sections below; the declarator procedure is described in Chapter 5.

DECLARATION OF SIMPLE VARIABLES

The declaration of simple variables is made by writing a list of identifiers separated by commas and preceded by one of the declarators real, integer or Boolean. The general form is

```
declar var1, var2, ..., varn;
```

where declar is one of the declarators real, integer or Boolean, and var1, var2, ..., varn represent variables.

The order in which type declarations are made and the order in which variables of each type are listed are not significant.

Some examples of the declaration of simple variables are given below:

```
integer A, B, C

real E, F, G

Boolean BOOL1, BOOL2

integer ALPHA, BETA, GAMMA
```

DECLARATION AND USE OF ARRAYS

The array declaration defines the name by which the array is to be known, the number of dimensions and the upper and lower limit of each dimension. The number and size of the dimensions are specified by a bound pair list.

Subsequently, each element of the array is referred to, as a variable, by its name and a subscript list, the value of each subscript lying within the corresponding bounds defined by the bound pair list of the declaration. In common with simple variables, arrays must be declared at the head of the outermost block in which they will be used.

The general form of the array declaration is as follows:

```
declarray arr1,arr2,...,arrn
```

where declarray represents one of the declarators array, real array, integer array or Boolean array; array and real array are equivalent and indicate that the elements of the array so declared are of type real; integer array and Boolean array indicate that the elements of the arrays so declared are of type integer and Boolean respectively.

arr1,arr2,...,arrn take the following form:

$$var[l_1 : u_1 , l_2 : u_2 , \ldots , l_n : u_n]$$

where u_1, u_2,...,u_n and l_1, l_2,...,l_n represent the permitted upper and lower bounds respectively for the n subscripts of the n-dimensional array var. The individual pairs of bounds (for example l_2 : u_2) are known as bound pairs. Individual bound pairs are separated by commas and the whole bound pair list is enclosed in square subscript brackets.

Note that in EMAS ALGOL an array can have up to 12 dimensions: that is, up to 12 bound pairs can appear in any one array declaration.

l_1, l_2,...,l_n and u_1, u_2,...,u_n are generally specified as positive, negative or zero integers or as integer variables, but they can take the value of any arithmetic expression. If the arithmetic expression has a real value, the appropriate bound will be rounded to the nearest integer.

The following examples illustrate array declarations more fully:

1. real array X [1:20,0:5]
   This declaration defines an array of 120 real variables which will be referred to as X[1,0],...,X[20,5]. Such subscripted variables as X[0,0] and X[0,6] would have no meaning in a block headed by the above

declaration.

2. Boolean array BOOL[1:2]
This declaration defines a one-dimensional array BOOL which has 2 elements, either of which can take the value true or false.

3. In all the above examples, the arrays bounds are integers. It is also possible for the array bounds to be any arithmetic expression, as in the following example:

```
real array MATRIX[1:N,1:SUB[I],0:M*Q]
```

Here the number of elements of the three-dimensional array MATRIX depend on the values of N, M, Q and the value of the subscripted variable SUB[I]. All these variables must be global to the current block and must have been assigned a value before the declaration of MATRIX was encountered.

4. Several arrays of the same type may be declared at the same time by separating successive items by commas.

If two or more arrays of a given type have the same bound pair list, the bound pairs need only be specified once; the individual identifiers are separated by commas.

The following declarations illustrate the above points:

```
integer array A, B, C[4:N,1:6], D[0:20]

real array PI, PSI[-1:4], THETA, PHI[1:2,1:9]
```

The following example illustrates array use:

If 100 numbers are stored as an array declared as:

```
real array ALPHA[1:100]
```

then the means

$$\frac{1}{100}\sum_{I=1}^{100} ALPHA_I$$

and

$$\frac{1}{100}\sum_{I=1}^{100} (ALPHA_I)^2$$

can be obtained by the series of statements

```
SUM:=SUMSQ:=0;
for I:=1 step 1 until 100 do
begin SUM:=SUM+ALPHA[I];
      SUMSQ:=SUMSQ + ALPHA[I]↑2
end;
AMEAN:=SUM/100;
BMEAN:=SUMSQ/100
....
```

At the first performance of the loop, SUM is set to ALPHA[1] and SUMSQ to ALPHA[1] ↑ 2; at the second performance, SUM is set to ALPHA[1] + ALPHA[2] and SUMSQ to ALPHA[1]↑2 + ALPHA[2]↑2, and so on.

SWITCH DECLARATIONS

Switches are used to generalise the goto statement and, in common with variables, must be declared at the head of the outermost block in which they are used.

A switch is effectively a one-dimensional array of labels, and the switch declaration associates each element of the switch with a label. The general form of a switch declaration is as follows:

```
switch sw:=desexp1, desexp2, ..., desexpn
```

where sw is an identifier and desexp1, desexp2, ..., desexpn are designational expressions forming a switch list. The elements desexp1, desexp2, ..., desexpn of the switch list are referred to from the program by using sw[1], sw[2], ..., sw[n] respectively, in conjunction with a goto statement. For example:

```
goto sw[2]
```

is equivalent to

```
goto desexp2
```

As with array subscripts, the switch subscripts may take the value of any arithmetic expression, real values being rounded to the nearest integer. If the subscript value is <1 or >N then no branch occurs. This is in accordance with the ALGOL Report but the reader is warned that many compilers regard a switch out of range as a program error.

The following example illustrates the use of switches and a conditional designational expression:

```
switch TEST1:=LAB1,LAB2,FAIL[M];
switch TEST2:=LAB3,LAB4,FAIL[M];
switch FAIL:=EXIT1,EXIT2,EXIT3;
                ...
goto if I # 1 then TEST1[N] else TEST2[N];
                ...
```

With reference to the goto statement, if I is not equal to 1, the switch list for TEST1 is examined. If N has the value 1 or 2, a branch is made to LAB1 or

LAB2 respectively. If N has the value 3, reference is made to the switch list for FAIL. A branch will be made to EXIT1, EXIT2 or EXIT3 depending on whether the value of M is 1, 2 or 3 respectively. If I is equal to 1, the switch list for TEST2 is examined. A branch is made to LAB3 or LAB4 if N has the value 1 or 2 respectively, and the switch list for FAIL is referred to if N has the value 3. In the latter case, a branch will be made to EXIT1, EXIT2 or EXIT3 depending on whether the value of M is 1, 2 or 3 respectively. If N<1 or N>3 or (N=3 and (M<1 or M>3)) then no branch is made.

SCOPE OF IDENTIFIERS

It was stated earlier that the use of block structure has many advantages, not least of which is the facility given to the programmer to write and test his program as a series of separate logical units. In so doing, the programmer might well inadvertently have a clash of identifiers and/or labels when his units are finally assembled to make a complete program. To avoid such clashes, certain restrictions are applied to limit the scope of each identifier used in a program. These restrictions are listed below:

1. (a) A given identifier can only be used to declare one type of variable in a given block.

   (b) An identifier may not be declared at the head of a block and appear as a label in the same block.

   (c) An identifier may not be used as a label twice in the same block.

2. A variable represented by an identifier declared at the head of a block does not exist outside that block.

3. A variable represented by an identifier declared at the head of a block is only accessible in an inner block if the same identifier does not appear as a label or has not been redeclared in the inner block or in any intermediate block.

4. A label is not accessible outside the block in which it occurs. In other words, all entries to a block must be through the first begin and, in contrast to a jump into a compound statement, a jump to a label within a block from outside the block is forbidden.

5. A label occurring in a block is not accessible from an inner block if the relevant identifier has been declared as an identifier for a variable at the head of the inner block or any intermediate block, or appears as a label within the inner block or any intermediate block. In other words, exit from a block to a label via a goto statement is only possible if the label has not been re-used or declared in the block containing the goto statement or in any block within the block containing the label that itself encloses the block containing the goto.

The examples which follow illustrate the application of the rules listed above.

EXAMPLES OF THE RESTRICTED SCOPE OF IDENTIFIERS

In the following examples, the begin and end symbols for nested blocks have been indented. The indentation has no significance other than to make the program structure clear to the reader. The reader is, however, strongly recommended to follow this practice when writing his own programs.

Example 1

```
begin real X;
      .....
      begin integer X;
            .....
            begin integer Y;
                  .....
            X:
                  .....
                  goto X;
                  .....
            end;
            .....
      end;
      .....
end
```

In this example, X may take a real value in the outermost block, an integer value in the first inner block, and is used as a label in the innermost block.

Example 2

```
begin real Z;
      integer I;
            .....
      LAB:
            .....
      begin real array A[0:6,1:7];
                  .....
                  goto LAB;
                  .....
            LAB: I:=I+Z;
                  .....
      end;
      .....
      goto LAB;
      .....
end
```

In this example, the goto statement in the inner block causes a jump to the statement labelled LAB in the same block. At this point, I is assigned the value I + Z; I and Z are still accessible in the inner block as neither has been redeclared or used as a label within this block. The goto statement in the outer block causes a jump, not to the label LAB within the inner block, but to the label LAB in the outer block.

OWN VARIABLES AND ARRAYS

It has already been stated that the values of variables or arrays are lost after exit from the block in which they are declared. In some applications, it would be advantageous for a variable or array to have the same value or values on re-entry to a block that it had on the previous exit from that block. This facility is obtained by prefixing the appropriate declarations with the symbol own. In EMAS ALGOL own arrays must have constant bounds.

For example:

```
own integer I;
own real array LIST[1:5,1:10];
own Boolean array TRUTH[1:8,1:6]
```

On the first entry to a block, the own variables or arrays may be given some values by assignment statements. Then on subsequent entries to the block, these statements can be by-passed leaving the variables and arrays with the values they had at the last exit from the block.

An example of a block with own variables is given below. This block is designed to develop successive Fibonacci terms, the Fibonacci series being 1,1,2,3,5,8,13, etc., where each term except the first two is the sum of the two preceding terms.

```
FIB: begin own integer CURRENT, PREVIOUS;
           integer SUM;
           switch S:=FIRST, NEXTFIB;
           goto S[N];
     FIRST: CURRENT:=PREVIOUS:=1;
            N:=2;
     NEXTFIB: SUM:=CURRENT + PREVIOUS;
              PREVIOUS:=CURRENT;
              TERM:=CURRENT:=SUM
     end
```

Before the first entry to the block, the variable N must be declared and set to 1 so that goto S[N] causes a branch to FIRST. The succeeding statements are performed to obtain the third Fibonacci term, 2, as the variable TERM (declared outside the block).

Subsequent entries to the block will develop successive Fibonacci terms, provided N is left unchanged at 2. In the second entry, for example, goto S[N] becomes goto NEXTFIB. As the values of CURRENT and PREVIOUS have been retained, SUM becomes 3; PREVIOUS becomes 2; and TERM and CURRENT become 3, the fourth Fibonacci term. In the third entry, goto S[N] again becomes goto NEXTFIB; SUM becomes 5; PREVIOUS becomes 3; and TERM becomes 5, the fifth Fibonacci term.

Thus, at each entry, this block forms the current Fibonacci term from the previous two Fibonacci terms. On the next entry to the block, this current term and the term before it will still be available for calculating the next term.

It is important to note that identifiers appearing in own declarations are, like other identifiers, local to the block in which they are declared. Consequently, even though any values assigned to own variables are retained throughout the program, it is possible to refer to these variables only in the block in which

they are declared.

Note also that own variables, like other variables, initially have no defined values.

CHAPTER 5 PROCEDURES

It is frequently necessary to perform the same calculation at different points in a program. If the programmer merely repeats the same series of statements each time the calculation is required then he is wasting his own time, compilation time and, frequently, storage space at run time. It is far better to write the series of statements once in the form of a simple procedure and to call this procedure at the required points in the program.

Simple procedures are of considerable value but they form only part of ALGOL procedure facilities. It is also possible to write generalised procedures with formal parameters which may be used for a variety of similar rather than identical calculations. Each time one of these procedures is called, actual parameters are supplied to make the procedure suitable for the current calculation.

The use of procedures also facilitates the interchange of problem-solving techniques between different programmers. A technique may be coded as a procedure with formal parameters, and then incorporated into an existing program, and be made relevant to that program by giving appropriate actual parameters at the time of the call.

There are two stages in the use of a procedure: the declaration and the call. A procedure call can be made within the main program by means of a procedure statement, consisting of the procedure identifier, possibly followed by actual parameters enclosed in parentheses. However, it should be noted that a procedure identifier, like any other identifier, is defined only within the block in which it is declared. Thus, a procedure should be declared in the outermost block in which it is to be called, or in some block enclosing this, and its identifier must not be used for other purposes in any inner block which will make use of the procedure, or which contains a block which will make use of it. A procedure name is valid throughout the block in which it is declared and hence one procedure in a block can call any other procedure in the same block regardless of the order in which the procedure declarations appear.

SIMPLE PROCEDURES

Suppose that the value of Y from the equation

$$Y = X^3 + 2X^2 + 1.3$$

is required for different values of the variable X. A possible procedure declaration is:

```
procedure FINDY;
      Y:= (X + 2) * X↑2 + 1.3
```

The symbol procedure followed by the identifier FINDY forms the procedure heading. This statement is separated from the procedure body, in this case a single statement, by a semi-colon. As with the declaration of arrays and simple variables, this declaration must be made at the head of a block within the scope of the variables X and Y. The following might be the start of a block containing the procedure FINDY:

```
begin real X, Y, Z;
      integer array A[1:9,1:3];
      procedure FINDY;
      Y:= (X + 2) * X↑2 + 1.3
      .....
```

Once the declaration has been made, Y may be calculated anywhere within the scope of X and Y by the procedure statement:

```
FINDY
```

Examples of the call of FINDY are as follows:

1.
```
X:= 3.016↑3;
FINDY
```

2.
```
if X < 10 then FINDY else Y:=0
```

The procedure FINDY above has only one assignment statement as its procedure body. It is possible to write a procedure body consisting of a compound statement or an ALGOL block, as in the following examples:

1.
```
procedure FINDY 1 TO 3;
begin Y[1]:= 1 + X;
      Y[2]:= 1 + X↑2;
      Y[3]:= 1 + X↑3
end
```

2.
```
procedure ASSIGN Z;
begin real P;
      P:= X↑2;
      Z[1]:= 1 + P;
      Z[2]:= 1 + P * X;
      Z[3]:= 1 + P * P
end
```

In the first example, the procedure call

```
FINDY1 TO 3
```

will assign values to the subscripted variables Y[1], Y[2] and Y[3]. In the second example, P is declared at the head of the block forming the procedure body. P is then assigned the value of X and values are assigned to the three subscripted variables Z[1], Z[2] and Z[3]. Note that P is local to the block forming the procedure body, and that it is inaccessible outside this block.

Procedure bodies which are blocks and compound statements follow the normal rules for such statements. Labels may be used within the procedure body if desired. Such labels are inaccessible from outside the procedure body, although a goto statement in a procedure body can refer to a label outside the procedure body, subject to the scope rules given earlier in this chapter.

PROCEDURES WITH FORMAL PARAMETERS

All the procedures given so far are comparatively inflexible since they always operate on the same variables and always assign results to the same variables. Of more general use are procedures which specify an algorithm without indicating the values involved, the appropriate values being supplied at the time of the call.

For example, suppose that the value of the general expression:

$$X^3 + 2X^2 + 1.3$$

must be calculated for many values of X, and that the result of the calculation must be assigned to a variety of variables. To achieve this, the first example of this chapter could be modified as follows:

```
procedure FINDY (X, Y); real X, Y;
          Y:= (X + 2) * X↑2 + 1.3
```

The procedure body is identical to that of the first example but the procedure heading now consists of the identifier followed by a formal parameter part and a specification part.

The formal parameter part (X, Y) defines X and Y as formal parameters, to be replaced by actual parameters at each call. In the procedure body, X and Y may be considered as dummy quantities to be replaced by actual quantities each time the procedure is called. Note that the identifiers of formal parameters are undefined outside the procedure body.

The specification part real X, Y indicates that the formal parameters X and Y are of type real. Although this particular specification part has the same form as a type declaration defining X and Y as real variables, it is not such a declaration. It merely indicates that X and Y will each be replaced by a real quantity when the procedure is called.

Examples of calls on this procedure are:

1. FINDY (A,B)

2. FINDY (L * M↑2, N)

These two calls specify that the procedure FINDY is to be evaluated with X and Y replaced respectively by

1. A and B

2. the expression L * M↑2 and the variable N.

In the example procedure, FINDY, both formal parameters are called by name. This means that whenever a formal parameter appears in the procedure body, it is replaced, when a call takes place, by the corresponding actual parameter.

Thus the second call of FINDY above has the same effect as

```
N:= (L * M↑2 + 2) * (L * M↑2)↑2 + 1.3
```

with the expression L * M↑2 being evaluated twice. In this case, a better programming technique is to call the formal parameter X by value. To do this, a value part is incorporated into the declaration as follows:

```
procedure FINDY(X,Y); value X; real X,Y;
      Y:= (X + 2) * X↑2 + 1.3
```

The value part, value X, indicates that an actual parameter, corresponding to X, will be evaluated once at the time of the call, and that this value (rather than the actual parameter itself) will be substituted whenever the formal parameter is encountered in the procedure body. Thus, if L * M↑2 has the value 5, the call

```
FINDY (L * M↑2, N)
```

has the same effect as

```
N:= (5 + 2) * 5↑2 + 1.3
```

Formal parameter parts, specification parts, actual parameters, value parts, calling by name and calling by value will now be considered separately in the sections which follow.

FORMAL PARAMETER PART

The formal parameters of a procedure are listed in parentheses immediately after the procedure identifier. Individual parameters may be delimited either by a comma or by the following:

```
) string : (
```

where string is any string of one or more letters.

This string of letters is equivalent to a comma, and enables the programmer to insert comments defining the purpose of each of the parameters in his procedure. As this form of delimiter and the comma are not distinguished, there is no necessity to use the same form of delimiter in both declaration and call.

The parameters of the procedure may represent variables, arrays, labels, identifiers of other procedures etc., and the associated specification part indicates what each parameter represents.

For example:

The procedure SIGMA may be declared as

```
procedure SIGMA (SUM, TERM, I, A, B)
```

This declaration could be replaced by the declaration

```
procedure SIGMA (SUM) of array: (TERM) from: (I) equals: (A) to: (B)
```

which would have an identical effect.

SPECIFICATION PART

In EMAS ALGOL, the specification part of the procedure heading must be present if formal parameters are used and it must contain a reference to each of the formal parameters which precedes it.

The specification part appears immediately before the procedure body and is made up of one or more specifiers, each followed by a list of one or more of the formal parameters. Successive entries in this list are separated by commas and each list is terminated by a semi-colon.

The full list of specifiers is:

real	integer	Boolean	
array	real array	integer array	Boolean array
procedure	real procedure	integer procedure	Boolean procedure
switch	label	string	

These specifiers are used to indicate the kind and type of the actual parameters which are to replace the formal parameters at the time of a call.

Note: The specifiers can be preceded by a value part denoted by the basic symbol value. Use of value is described later in this chapter.

If a formal parameter is specified as real, integer or Boolean, then a quantity of type real, integer or Boolean is expected as an actual parameter.

If a formal parameter is specified as an array, then an array identifier is expected as an actual parameter. No bounds need be quoted in conjunction with an array specifier, the bounds being determined by the actual parameter.

If a formal parameter is specified as a procedure, then a procedure identifier is expected as an actual parameter, i.e. the procedure contains a reference to another procedure, to be provided at the time of the call. The use of the specifiers real procedure, integer procedure and Boolean procedure will become apparent when functions are considered.

If a formal parameter is specified as a switch, then a switch identifier is expected as an actual parameter.

If a formal parameter is specified as a label, then a designational expression is expected as an actual parameter.

If a formal parameter is specified as a string, then a string is expected as an actual parameter. The string can be used only as an actual parameter for other procedures calls within the procedure body.

Several examples of the use of these specifiers are given throughout the remainder of this chapter.

SWITCH IDENTIFIERS AND DESIGNATIONAL EXPRESSIONS AS PARAMETERS

If parameters appearing in the formal parameter list correspond to actual parameters which are switch identifiers or labels, then these formal parameters must appear in the specification part, preceded by switch or label respectively.

LABELS AS FORMAL PARAMETERS

The use of formal parameters which are specified as labels enables the programmer to jump out of a procedure to one of several labels, depending on the actual parameters used when the procedure is called.

For example:

```
procedure TEST(A,B,C,D,OUT); value A,B,C;
                             real A,B,C,D;
                             label OUT;
if (A↑2 + B↑2) < C then goto OUT else D:= A + B + C
```

SWITCHES AS FORMAL PARAMETERS

When the specifying symbol switch is used, a complete switch list is transferred via the parameter list. The example below uses an own variable:

```
procedure CHOOSE (SW,BOOL); switch SW; Boolean BOOL;
          begin own integer I;
                I:= if BOOL then I + 1 else 1;
                goto SW[I]
          end
```

The procedure CHOOSE causes a jump to a label in the list of the actual switch supplied in place of SW. If BOOL is false then the first label in the switch list is used; if BOOL is true the position of the label chosen from the switch list is incremented by one.

Note: as I is an own variable, its previous value is accessible each time the procedure is called. On the first call of procedure CHOOSE, BOOL must have the value false, so that I, previously undefined, can be assigned a value.

PROCEDURE IDENTIFIERS AS PARAMETERS

Consider the following call of the procedure FINDY on page 37:

```
FINDY (SIN(A),B)
```

Here the formal parameters X and Y have been replaced by SIN(A) and B respectively. SIN(A) is itself a procedure call, and the identifier SIN is

supplied complete with its own actual parameter A.

It is sometimes useful to use, as an actual parameter, a procedure identifier without parameters, the appropriate procedure being supplied as an actual parameter. In this case, the specification part of the calling procedure declaration must be followed by a specially constructed comment, as in the following example:

```
procedure P(X,Q);
        value X; real X;
        real procedure Q;
        comment (R,S): value R,S: real R,S;
        begin real Z;
              .....
              Z:= Q(X,X);
              .....
        end
```

In this example, the comment specification indicates that any actual procedure corresponding to the parameter procedure Q will have two real parameters which will be called by value. The assignment statement within the procedure body indicates one possible call of the parametric procedure Q.

It is also possible for one of the parameters of the parametric procedure to be a procedure identifier. In this case, no further comment is required. For example:

```
procedure P(X,Q);
        value X; real X;
        real procedure Q;
        comment (R,S,T): value R,S: real procedure T: real R,S;
        begin real Z;
              .....
        end
```

In this example, the parameter T will be replaced at the time of call by the identifier of a real procedure. If the comment specification is omitted, it will be assumed that the parametric procedure has no parameters. If several parametric procedures are used as formal parameters, each should be followed by its own comment specification, as follows:

```
procedure P1; comment ...;
procedure P2; comment ...;
...
procedure PN; comment ...;
```

if the comment specification appeared as follows:

```
procedure P1, P2, ..., PN; comment...
```

it would be assumed that the comment specification referred to all the procedures P1 to PN.

ACTUAL PARAMETERS

When a procedure with formal parameters is called, a list of actual parameters enclosed in parentheses must follow the identifier. Successive actual parameters are separated by commas and there must be an actual parameter for each formal parameter. Actual parameters must be in the same order and, in general, must be of the same type as the corresponding formal parameters.

The substitution of actual parameters (or the value of actual parameters) for formal parameters must form a valid ALGOL statement. In particular, arithmetic expressions may be used as actual parameters which correspond to formal parameters on the right-hand side of assignment statements in the procedure body. Such expressions may not, however, be used in this way for formal parameters which appear on the left-hand side of assignment statements in the procedure body. Thus, consider the procedure FINDY described in this chapter as:-

```
procedure FINDY(X,Y); real X,Y;
          Y:=(X + 2) * X↑2 + 1.3
```

then the following call is valid

```
FINDY (A * A + B, S)
```

but the call

```
FINDY (A * A + B, S + R)
```

is not permissible, as the replacement of the formal parameter by the actual parameter results in the statement

```
S + R:=((A * A + B) + 2 * (A * A + B)↑2 + 1.3
```

which is meaningless.

In EMAS ALGOL, actual parameters must correspond in kind and type to the formal parameters they replace, with one exception: if a formal parameter that does not occur on the left hand side of any assignment statement in the procedure body is specified as real, a corresponding actual parameter of type integer is allowed. Similarly, and subject to the same conditions, an integer formal parameter can be replaced by a real quantity.

Note that it is permissible to use the same identifier for an actual parameter and for a corresponding or non-corresponding formal parameter. Thus, the following call is permissible for procedure FINDY:

```
FINDY (A,X)
```

Here, X replaces the formal parameter Y even though there is a formal parameter X.

VALUE PART

A value part immediately after the list of formal parameters indicates that some or all of these parameters are to be called by value. It consists of the symbol value followed by a list of the formal parameters to be called by value.

Each such parameter is separated from the next by a comma and the whole list is terminated by a semi-colon. If the value part is present, parameters are said to be called by value; otherwise the parameters are said to be called by name.

CALLING BY NAME AND CALLING BY VALUE

Calling by Name

Section 4.7.3.2 of the ALGOL report summarises the operation of call by name as follows:

> 'Name replacement (call by name). Any formal parameter not quoted in the value list is replaced, throughout the procedure body, by the corresponding actual parameter, after enclosing this latter in parentheses wherever syntactically possible. Possible conflicts between identifiers inserted through this process and other identifiers already present within the procedure will be avoided by suitable systematic changes of the formal or local identifiers involved'.

The reason for assuming that actual parameters are enclosed in parentheses on replacement can be seen by considering the following procedures:

```
procedure FINDP (P,Q); real P,Q;
          P:= EXP (Q *10 -6)
```

A call such as

```
FINDP (A, C + D)
```

is equivalent to

```
A:= EXP ((C + D) *10 -6)
```

If the actual parameter C + D had not been considered as being in parentheses, the incorrect result

```
A:= EXP (C + D *10 -6)
```

might be expected.

The automatic changes of identifiers in a call by name avoid ambiguity when the same identifier happens to be used for a formal and actual parameter.

For example, a procedure to evaluate the general summation

$$\sum_{i=a}^{b} t(i)$$

could be declared as:

```
procedure SIGMA (SUM,TERM,I,A,B); real TERM,SUM;
      integer I,A,B;
      begin SUM:=0;
            for I:=A step 1 until B do
            SUM:=SUM + TERM
      end
```

This general procedure may now be used in the main program to evaluate a variety of different sums; for example:

1. The call

   ```
   SIGMA(ROOTSUM, X↑(1/2), X, 1, 100)
   ```

   evaluates the sum

   $$\sum_{x=1}^{100} x^{1/2}$$

   and assigns the result to ROOTSUM.

2. The call

   ```
   SIGMA(TVECTOR,A * V [R] + B * V [2 * R] + C * V [3 * R], R, I, J * K)
   ```

   evaluates the sum

   $$\sum_{r=i}^{j \times k} (a.\ v\ [r] + b.\ v\ [2r] + c.v\ [3r])$$

   where v is an array. The results are assigned to TVECTOR.

The second call is equivalent to the compound statement:

```
begin TVECTOR:=0;
      for R:=I step 1 until J * K do
      TVECTOR:=TVECTOR + A * V [R] + B * V [2 * R] + C * V [3 * R]
end
```

This call leads to inefficiency because J * K must be calculated at each performance of the range. An improved procedure declaration is given in the next section.

Calling by value

Section 4.7.3.1 of the ALGOL report summarises the operation of call by value as follows:

> 'Value assignment (call by value). All formal parameters quoted in the value part of the procedure declaration heading are assigned the values of the corresponding actual parameters, these assignments being considered as being performed explicitly before entering the procedure body. The effect is as though an additional block embracing the procedure body were created in which these assignments were made to variables local to the fictitious block with types as given in the corresponding specifications. As a consequence, variables called by value are to be considered as non-local to the body of the procedure, but local to the fictitious block'.

The general summation procedure in the previous section can be improved by calling A and B by value, as follows:

```
procedure SIGMA (SUM,TERM,I,A,B);
          value A,B;
          real TERM, SUM;
          integer I,A,B;
          begin SUM:=0;
               for I:=A step 1 until B do
               SUM:=SUM + TERM
          end
```

If the actual parameters which correspond to A and B are expressions, these expressions will be evaluated once only, not at every repetition of the for statement. For example, if J * K = 50, then the call

```
SIGMA (TVECTOR, A * V [R] + B * V [2 * R] + C * V [3 * R],R,I,J * K)
```

is equivalent to

```
begin TVECTOR:=0;
      for R:= I step 1 until 50 do
      TVECTOR:= TVECTOR A * V [R] + B * V [2 * R] + C * V [3 *R]
end
```

Now only one calculation of J * K is performed; the earlier procedure would require 50 such calculations.

If an assignment is made to a value parameter the effect is to assign to the local copy of the parameter. The actual parameter is not affected. It follows that calling by value cannot be used for any parameter used to return a result to the calling block or procedure.

ADVICE ON THE USE OF CALLING BY NAME AND BY VALUE

Care should be taken to ensure that the correct alternative is chosen when deciding between a call by name and a call by value. The following paragraphs indicate the circumstances in which each should be used. When a variable is called by value, a working copy, which may be considered local to the procedure

body, is made of the current value of the variable. Thus, in the case of an array called by value, a copy is made of the whole array.

Consider the following:

1. A simple variable which is a data item for a procedure should be called by value. Simple variables which are used by a procedure to return a result, or which are employed in Jensen's device (see below), must be called by name.

2. An array should normally be called by name, whether it is used as a data item for the procedure or to return results from the procedure. However an array whch is not used to return results may be called by value, in which case a copy of the whole array is made. The procedure may then use the copy as working storage without corrupting the original array.

FUNCTIONS

If a procedure identifier is to return a value so that it may be used as a term in an arithmetic or Boolean expression, the procedure identifier must appear on the left-hand side of an assignment statement in the procedure body. The procedure declaration must also specify the type of value which will be returned.

Thus, the form of a function procedure declaration differs in two respects from the form of the general procedure declarations already considered:

1. The identifier chosen to name the function is also used for the variable to which the single result of the procedure is assigned.

2. The type of this result must be declared by preceding the symbol procedure by a type symbol: real, integer or Boolean.

   For example, suppose $(x^2 + y^2 + z^2)^{\frac{1}{2}}$ is required for various values of x, y and z. A possible function procedure is:

```
real procedure MOD(X,Y,Z);
        value X,Y,Z;
        real X,Y,Z;
        MOD:=SQRT (X↑2 + Y↑2 + Z↑2)
```

This function might be used in statements such as:

```
M:=MOD(P,Q,R)

THETA:=P/MOD(P,Q,R)

A:=MOD(B + C, A + C, C) + MOD(D + K, E + K, F + K)
```

A function procedure identifier followed by a list of actual parameters in parentheses is termed a function designator. Function designators may appear in expressions wherever simple variables are permitted. When a function designator is encountered in an expression, evaluation of the expression is suspended whilst the procedure is executed; the value returned by the procedure is then

used to complete the evaluation of the expression.

A function procedure identifier, like any other procedure identifier, is local to the block in which it is declared.

Finally, note that while the identifier of a function procedure must appear on the left-hand side of at least one assignment statement in the procedure body, it should not appear on the right-hand side of a statement in the body unless recursion (see the next section) is intended.

RECURSION

The ALGOL language has an hierarchical structure. Each element in the language is defined in terms of the elements at a lower level of the structure, but certain elements are allowed to include themselves, directly or indirectly, in their own definitions. For example, a block is defined in terms of statements but a statement may itself be a block.

The ALGOL language is said to be defined recursively, and therefore ALGOL programs can be written using recursion. In EMAS ALGOL, no additional overheads are associated with recursive procedure calls and recursion may be used extensively. An ALGOL procedure may make reference to itself within its own procedure body. Whenever the right-hand side of an assignment statement within a procedure body contains the procedure identifier, then a recursive definition of the procedure is implied.

Simple examples of recursive definitions are rare but the following example, whilst it is an inefficient method of obtaining factorials, illustrates the general principles of this form of definition.

```
integer procedure FACTORIAL (N);
        value N; integer N;
        FACTORIAL:= if N=0 then 1 else N * FACTORIAL (N-1)
```

Note that FACTORIAL appears in the function definition in the normal way as the output variable on the left-hand side of the assignment statement, and that it also appears on the right-hand side of this statement, where it represents a recursive call.

If an actual parameter of 0 replaces N at a call, then the statement FACTORIAL:=1 is performed.

If an actual parameter of 3 replaces N at a call, then

```
FACTORIAL:= 3 * FACTORIAL (2)
```

is performed. As the right-hand side of this statement contains a function designator, evaluation is suspended while FACTORIAL(2) is executed. Of course, the execution of this function is itself suspended while FACTORIAL(1) is executed. To evaluate factorial(n), n procedure calls are required. A more efficient procedure to compute factorial(n) would use a for statement to avoid repeated procedure calls.

The Game of Hanoi

The following procedure is the classic example of recursion and demonstrates the power of the concept in a most elegant way.

In this game, one is given three pegs, and slotted onto one of these pegs are a number of circular discs of different diameters, graded so that the largest is at the bottom and the smallest at the top. The aim of the game is to transfer all the discs to one of the other pegs (making use of the third as required) in such a way that there is never a larger disc above a smaller one. Only one disc at a time may be moved.

If the solution to the game for (N-1) discs is known, then the solution for N can easily be obtained. Let the pegs be numbered 1, 2 and 3 and let it be required that the N discs on 1 be transferred to 3. This can be done by transferring the first (N-1) to 2, the last to 3, and then the first (N-1) from 2 to 3. In the following program, N is the number of discs which have to be moved from peg1 to peg2. If peg1 and peg2 are two pegs, the third is 6-peg1-peg2.

```
begin integer NDISCS,FROMPEG,TOPEG;
      procedure HANOI (N,PEG1,PEG2);
      value N,PEG1,PEG2;
      integer N,PEG1,PEG2;
      if N # 0 then
      begin integer PEG3;
            PEG3:= 6 - PEG1-PEG2;
            HANOI (N-1,PEG1,PEG3);
            PRINTSTRING ([MOVE]);
            PRINT (PEG1,1,0);
            PRINTSTRING ([->]);
            PRINT (PEG2,1,0);
            NEWLINE;
            HANOI (N-1,PEG3,PEG2)
      end;
      NDISCS:=READ;
      FROMPEG:=READ;
      TOPEG:=READ;
      HANOI (NDISCS,FROMPEG,TOPEG)
end
```

The output for the case NDISCS=2, FROMPEG=1, TOPEG=3, is:

```
MOVE 1 -> 2
MOVE 1 -> 3
MOVE 2 -> 3
```

JENSEN'S DEVICE

Consider a procedure designed solely to form the sum of the diagonal elements of real, two-dimensional square arrays:

```
real procedure DIAGSUM (A,M,N);
     value M,N;
     array A;
     integer M,N;
     begin integer I; real S;
           S:=0;
           for I:=M step 1 until N do
           S:=S + A [I,I];
           DIAGSUM:=S
     end
```

A possible call is

```
TRACE:=DIAGSUM(MATRIX, 1, 10);
```

which forms the sum MATRIX [1,1] + MATRIX [2,2] + ... + MATRIX [10,10].

This procedure is adequate if nothing but diagonal sums is required, but it is possible to write a procedure of more general use by means of a programming technique known as Jensen's device.

This consists of using a for statement within a procedure and making the controlled variable a formal parameter called by name. If the above procedure is further modified by making A a real parameter and by giving the procedure the name SUM then the following is obtained:

```
real procedure SUM (A,I,M,N);
     value M,N; real A; integer I,M,N;
     begin real S;
         S:=0;
         for I:=M step 1 until N do
         S:=S + A;
         SUM:=S
     end
```

The diagonal sum can still be obtained by the call

```
TRACE:=SUM (MATRIX [R,R],R,1,10)
```

which is equivalent to

```
for R:= 1 step 1 until 10 do S:= S + MATRIX [R,R]
```

However, the procedure may now be used for a variety of operations by calls such as:

1. ADD:= SUM(A [I],I,7,19)
   which forms the sum A [7] + A [8] + ... + A [19].

2. ODDADD:= SUM(A [2*I+1],I,3,9)
   which forms the sum A [7] + A [9] + ... + A [19].

3. NSUM:= SUM(I,I,1,N)
   which forms the sum of the first N integers.

4. TOTAL:= SUM(SUM(SUM(TENSOR[J,K,L],L,1,10),K,1,15),J,1,12)
   which uses recursive calls to calculate the sum of all the elements of a

three-dimensional array declared as:

```
array TENSOR [1:12, 1:15, 1:10]
```

This call is equivalent to:

```
for J:= 1 step 1 until 12 do
    S:=S + SUM(SUM(TENSOR [J,K,L],L,1,10),K,1,15)
```

The function call in the range of the for statement is equivalent to:

```
for K:=1 step 1 until 15 do
    S:=S + SUM(TENSOR[J,K,L],L,1,10)
```

The function call in the range of this for statement is equivalent to:

```
for L:= 1 step 1 until 10 do
    S:= S + TENSOR [J,K,L]
```

The first evaluation obtains the sum of the elements TENSOR [1,1,1] to TENSOR [1,1,10]. Then K is stepped to 2 to add to the sum TENSOR [1,2,1] to TENSOR [1,2,10], and so on until the final element, TENSOR [12,15,10], is accumulated.

Thus Jensen's device makes it possible to write a for statement which performs some general operation such as summation upon dummy quantities, and then to provide the actual factors and limits at the time of call. However this great generality is obtained at the cost of execution time in the compiled program and should only be used when the generality is really necessary.

CHAPTER 6 STANDARD FUNCTIONS

Certain standard functions are available to the EMAS ALGOL programmer without declaration. These functions are in all respects equivalent to user defined functions except that they may be used without declaration. As recommended in the ALGOL Report, these functions are declared within a hypothetical begin ... end block which surrounds the user program. It follows from the rules governing block structure that the names of these functions may be redeclared for other purposes within the program, although, of course, the corresponding standard functions then become inaccessible.

The formal declarations of the standard functions provided are given below.

ABS

```
real procedure ABS(E);
     value E; real E;
     comment GIVES THE ABSOLUTE VALUE, OR MODULUS, OF E;
```

ARCTAN

```
real procedure ARCTAN(E);
     value E; real E;
     comment GIVES THE PRINCIPAL VALUE IN RADIANS OF THE ARCTANGENT OF E;
```

COS

```
real procedure COS(E);
     value E; real E;
     comment GIVES THE COSINE OF E, WHERE E IS EXPRESSED IN RADIANS;
```

ENTIER

```
real procedure ENTIER(E);
     value E; real E;
     comment GIVES  THE  LARGEST  INTEGER  NOT GREATER THAN THE VALUE OF E:
             THUS ENTIER (2.7) EQUALS 2 AND ENTIER (-3.1) EQUALS -4;
```

EXP

```
real procedure EXP(E);
     value E; real E;
     comment GIVES THE EXPONENTIAL OF E;
```

LN

```
real procedure LN(E);
     value E; real E;
     comment GIVES THE NATURAL LOGARITHM OF E;
```

SIGN

```
real procedure SIGN(E);
     value E; real E;
     comment GIVES THE SIGN OF THE VALUE OF E AS AN INTEGER:
             SIGN(E) = +1 IF E>0, -1 IF E<0 and 0 IF E=0;
```

SIN

```
real procedure SIN(E);
     value E; real E;
     comment GIVES THE SINE OF E, WHERE E IS EXPRESSED IN RADIANS;
```

SQRT

```
real procedure SQRT(E);
     value E; real E;
     comment GIVES THE POSITIVE SQUARE ROOT OF E FOR E>0;
```

CHAPTER 7 HINTS ON PROGRAM OPTIMISATION

The following notes are offered for those programmers who are interested in writing efficient ALGOL programs. It is necessary to keep a sense of proportion in these matters: if the program is only to be run a few times then it is most unlikely that the time saved by following these hints will be sufficient to justify the effort involved in altering a working program. In any event, the most important and easiest way of obtaining faster execution times is to ensure that the program is compiled in optimising rather than diagnostic mode (see Chapter 11).

(1) The parts of a program in which efficiency is particularly important are those which are executed many times.

```
for I := 1 step 1 until 100 do
for J := 1 step 1 until 100 do
begin
    <code>
end
```

In the above example, any minor improvement made in <code> will result in a total saving of 10,000 times that small amount.

(2) Whenever some part of an expression which is a constant factor is required many times, it should be evaluated once and stored as shown in the second version of the example below.

```
for I := 1 step 1 until 100 do
      A[I] := A[I] *K* 22/7
```

```
K := K* 22/7
for I := 1 step 1 until 100 do
      A[I] := A[I] * K
```

(3) It takes longer to move numbers in and out of array elements than to access simple variables. The second of the following two examples is the more efficient.

```
A[I,J] := 0;
for K := 1 step 1 until 15 do
A[I,J] := A[I,J] + B[K]
```

```
X := 0;
for K := 1 step 1 until 15 do
X := X + B[K];
A[I,J] := X
```

(4) The rules of ALGOL require step and until expressions of a for loop to be evaluated on every traverse of the loop. Thus where expressions occur with step and until which do not involve variables changing within the loop, it is more efficient to assign the expressions to local variables first. For example:

```
for I := 1 step 1 until J*K do
A [I] := A [I] + 1
```

would be more efficiently written as

```
U := J*K;
for I := 1 step 1 until U do
A [I] := A [I] + 1
```

(5) The user is referred to Chapter 5 for the differences between calling by name and calling by value. Much time can be wasted if parameters which are required by value are passed by name.

(6) Since real to integer conversion is implicit in ALGOL, the programmer may not realise how costly these operations are. Variables of type integer should always be used in array bounds and subscripts. Where mixed integer and real expressions are required, it is often worthwhile to use brackets to separate out integer sub-expressions. For example:

```
real X;                          real X;
integer I,J,K;                   integer I,J,K;
.....                            .....
X := X + I + J - K               X := X + (I + J - K)
```

(7) Output should be reduced to the minimum which is really useful to the programmer, in the interest of saving machine time and money. To this same end, answers should be printed across the full width of the line printer page where possible, as the cost is proportional to the number of lines printed.

CHAPTER 8 INPUT, OUTPUT

All input media are considered to provide, and all output media to accept, strings of visible* characters; these serial strings of characters are referred to as streams.

The user program has at any instant a current input stream from which any requested input will be taken, and a current output stream to which any generated output will be sent. Input is taken from the current stream unless and until a call of SELECT INPUT (STREAM) is used to bring input from another stream. When a new stream is selected the old one is left open and if the old stream is reselected reading resumes at the next line of input. A number of streams may be open at any one time for input and output and these will be closed automatically at the end of program execution. No stream may be open for both input and output simultaneously. However, it is possible, though seldom desirable, to output to a stream, then close the stream and reopen it for input.

The actual mechanics of input and output at the programming level are thus totally independent of devices and are usually performed by using some or all of the input and output functions defined below.

The very basic symbol routines (READSYMBOL, PRINTSYMBOL, etc) enable the programmer to write more complex procedures for input and output of peculiar data if he wishes. Most programmers will find the high level routines (READ, PRINT, etc) adequate for their requirements.

There are also available a few procedures for input and output of binary (unformatted) data. These will be very seldom required by a normal programmer as the virtual memory in which an EMAS ALGOL program runs is very large. Those interested should consult the User Manual for the appropriate machine.

The Input/Output procedure headings are as follows, an explanation of each procedure being given in the form of comment.

THE READ PROCEDURE

```
real procedure READ;
comment READS NUMERICAL DATA FROM THE CURRENTLY SELECTED INPUT STREAM,
        IGNORING MULTIPLE SPACE, NEWLINE OR NEWPAGE CHARACTERS BEFORE THE
        START OF THE NUMBER, AND TERMINATES ON READING THE FIRST CHARACTER
        WHICH IS NEITHER A DIGIT NOR A CORRECTLY PLACED EXPONENT OR SIGN.
        THE SYMBOLS & AND @ ARE USED AS REPRESENTATIONS OF    ;
                                                             10
```

An example of a simple call is:

```
X := READ; comment READS THE NEXT NUMBER FROM THE SELECTED INPUT
STREAM AND ASSIGNS IT TO THE VARIABLE X;
```

*The visible characters are defined on page 69.

Since the procedure is of type real, it provides values in real form irrespective of the format in which numbers are punched on the input medium. However, as READ is a function designator, a number may be read and stored in integer form by a statement such as:

```
I := READ
```

where the number provided by the READ procedure is converted to integer form before assignment to I, an integer variable.

Numbers provided for input to the procedure may be recorded on the input medium in any of the forms used for writing numerical constants in the program.

For example:

```
     1          +538491          -0.003568
    &12          0.005&15          32&-3
```

Another example of use of the procedure follows. Suppose a series of values are to be read and assigned to an array at several points in the program. As a procedure call can be included within the body of another procedure, the operation of reading a set of values and assigning them to the elements of an array may be defined by means of a procedure declaration, as follows:

```
comment THIS PROCEDURE READS 25 VALUES AND ASSIGNS THEM TO THE
        ELEMENTS OF A 5 * 5 ARRAY;
procedure ARRAY READ(X); array X;
begin integer I,K;
      for I := 1 step 1 until 5 do
      for K := 1 step 1 until 5 do
        X [I,K] := READ
end
```

THE PRINT PROCEDURE

```
procedure PRINT (QUANTITY, M, N); value QUANTITY, M,N;
real QUANTITY; integer M, N;
comment OUTPUTS TO THE CURRENTLY SELECTED OUTPUT STREAM THE VALUE OF
        QUANTITY, PRECEDED BY A SIGN.  M AND N DETERMINE THE FORMAT OF THE
        NUMBER.  QUANTITY MAY BE REAL OR INTEGER;
```

The value of M and N determine the format of the number as follows:

M=0 N≠0 The number is output in the floating-point form D&E where the mantissa D is written to N + 1 significant digits such that $1 \leq D < 10$, and the exponent E consists of two digits with a negative sign or a space representing +. The number always occupies N + 7 character positions in all.

EXAMPLE
M=0 and N=5:

```
-1.23456& 10
 3.45678&-12
 1.00000&  2
```

M#0 N#0 The number is output in fixed-point form with M digits to the left of the decimal point and N digits to the right. The number occupies M+N+2 character positions in all.

```
EXAMPLE
M=3 and N=2:
 123.45
  22.25
  -1.00
```

M#0 N=0 The number is output in integer form occupying M+1 character positions in all.

```
EXAMPLE
M=4 and N=0:
55555
-1245
   10
```

In each format, the total count of character positions includes a sign character which is minus or a space.

If M#0 and the number has more than M digits in the integral part, further character positions will be used to accommodate the number. This may cause following numbers to be displaced to the right when the results are printed.

If M#0 and the number has less than M digits in the integral part, spaces are inserted before the sign character to fill up to the required number of character positions.

For numbers whose modulus is less than 1, the zero before the decimal point is printed; that is, it is not replaced by a space. For example, 0.001. The fractional part of a number is rounded, in the decimal form, to the appropriate number of decimal places.

THE SPACE PROCEDURE

There are two procedures used to output one or more spaces:

1. procedure SPACE;
   comment OUTPUTS TO THE CURRENTLY SELECTED OUTPUT STREAM ONE HORIZONTAL SPACE;

2. procedure SPACES(N); value N; integer N;
   comment OUTPUTS TO THE CURRENTLY SELECTED OUTPUT STREAM N HORIZONTAL SPACES, EACH SPACE BEING EQUIVALENT TO ONE CHARACTER. IF N<=0 THEN THE PROCEDURE HAS NO EFFECT;

An example of the use and effect of these procedures is given after the description of procedure NEWPAGE.

THE NEWLINE PROCEDURE

There are two procedures used to output one or more newlines:

1. `procedure NEWLINE;`
   `comment OUTPUTS ONE NEWLINE TO THE CURRENTLY SELECTED OUTPUT STREAM;`

2. `procedure NEWLINES(N); value N; integer N;`
   `comment OUTPUTS N NEWLINES TO THE CURRENTLY SELECTED OUTPUT STREAM. IF N<=0 THEN THE PROCEDURE HAS NO EFFECT;`

With a line printer, a newline of print is started if the first of these procedures is called, or if N is 1; if N is 2, one blank line is left; if N is 3, two blank lines are left, and so on.

With a paper tape punch, N newline characters are output.

THE NEWPAGE PROCEDURE

```
procedure NEWPAGE;
comment OUTPUTS TO THE CURRENTLY SELECTED OUTPUT STREAM A COMMAND TO THROW
        TO THE HEAD OF A NEW PAGE;
```

EXAMPLE

The following program prints a table of sines and cosines of angles at one-degree intervals, to five-figure accuracy.

```
begin
      integer ANGLE; real SINE, COSINE, FACT, Y;
      FACT:=3.14159/180;
      NEWPAGE;
      for ANGLE:= 0 step 1 until 45 do
      begin
      Y:= ANGLE * FACT;
      SINE:= SIN(Y);
      COSINE:= COS(Y);
      PRINT (ANGLE, 2, 0);
      SPACES (6);
      PRINT (SINE, 1, 5);
      SPACES (6);
      PRINT (COSINE, 1, 5);
      NEWLINES (2)
      end
end
```

The results will be printed as follows:

```
  2 spaces  7 spaces      7 spaces
     ↓        ↓            ↓
              0        0.00000      1.00000
1 blank line →
              1        0.01745      0.99985
              2        0.03490      0.99939
              3        0.05234      0.99863
              4        0.06976      0.99756
              5        0.08716      0.99619

             etc
```

THE PRINTSTRING PROCEDURE

```
procedure PRINTSTRING (STRING); string STRING;
comment OUTPUTS  TO  THE CURRENTLY SELECTED OUTPUT STREAM THE GIVEN STRING:
        LAYOUT EDITING CHARACTERS ARE SPECIALLY INTERPRETED;
```

Text, such as headings and titles, etc. which the programmer wishes to output with his results can be included in the program and output by means of this procedure.

Since spaces and newlines are ignored in ALGOL, it is necessary to use special characters within the string to represent space and newline. The space is represented by _ and newline by ¬. The string is stored with _ and ¬ represented by their ISO internal code and the substitution is done by the PRINTSTRING procedure when the string is output.

EXAMPLE

The program given to illustrate the layout procedures can be amended to include column headings as follows:

```
begin
      integer ANGLE; real Y;
      FACT:= 3.14159/180;
      NEWPAGE;
      PRINTSTRING ([ANGLE ____SINE__________COSINE]);
      NEWLINES(2);
comment SOME UNNECESSARY STEPS HAVE BEEN ELIMINATED FROM  THIS  VERSION  OF
        THE PROGRAM;
      for ANGLE:= 0 step 1 until 45 do
      begin
      Y:= ANGLE*FACT;
      PRINT (ANGLE,2,0);
      SPACES (6);
      PRINT (SIN(Y),1,5);
      SPACES (6);
      PRINT (COS(Y),1,5);
      NEWLINES (2)
      end
end
```

Results will be printed as follows:

```
          2 spaces  7 spaces      7 spaces
             ↓         ↓            ↓
               ANGLE     SINE          COSINE
1 blank line →
               0         0.00000       1.00000
1 blank line →
               1         0.01745       0.99985

               2         0.03490       0.99939

               3         0.05234       0.99863

               4         0.06976       0.99756

               5         0.08716       0.99619


                  etc.
```

THE READ SYMBOL PROCEDURE

```
procedure READSYMBOL (I); integer I;
comment   READS ONE SYMBOL FROM THE CURRENTLY SELECTED INPUT STREAM AND
          ASSIGNS IT TO THE PARAMETER I, WHICH MUST BE A VARIABLE;
```

If this procedure is applied at the end of a card, then I is set to an end-of-line code, the same code being used when a newline character is read from the paper tape.

Only symbols visible at the terminal can be read by the READ SYMBOL procedure.

THE NEXTSYMBOL PROCEDURE

```
integer procedure NEXTSYMBOL ;
        comment   READS ONE CHARACTER FROM THE CURRENTLY SELECTED INPUT
                  STREAM BUT LEAVES THE INPUT STREAM POSITIONED AT THAT
                  CHARACTER;
```

Any number of consecutive calls of NEXTSYMBOL on the same channel will all obtain the same character.

THE CODE PROCEDURE

```
integer procedure CODE (X); string X;
        comment   PROVIDES THE CODE VALUE OF THE CHARACTER X, WHICH MUST BE
                  A SINGLE CHARACTER AND MUST BE ENCLOSED IN STRING QUOTES;
```

This procedure enables the programmer to manipulate the character information without needing to know the internal codes employed for characters, and greatly facilitates the transfer of ALGOL programs between computers using different internal codes. As in PRINTSTRING, _ and ¬ will give the internal code for space and newline respectively.

THE PRINTSYMBOL PROCEDURE

```
procedure PRINTSYMBOL (I); value I; integer I;
comment   OUTPUTS ONE CHARACTER TO THE CURRENTLY SELECTED OUTPUT STREAM;
```

THE SELECT INPUT PROCEDURE

```
procedure SELECT INPUT (N); value N; integer N;
comment   SELECTS INPUT STREAM N: SUBSEQUENT CALLS OF INPUT PROCEDURES TAKE
          INFORMATION FROM STREAM N UNLESS AND UNTIL A FURTHER CALL OF THIS
          PROCEDURE SELECTS A DIFFERENT STREAM;
```

Only one input stream may be selected at any time, and it may not be the same as the output stream. If a call of SELECT INPUT is made part way through reading a line of input the remainder of that line is lost.

THE SELECT OUTPUT PROCEDURE

```
procedure SELECT OUTPUT (N); value N; integer N;
comment   SELECTS OUTPUT STREAM N: SUBSEQUENT CALLS OF OUTPUT PROCEDURES
          SEND INFORMATION TO STREAM N UNLESS AND UNTIL A FURTHER CALL OF
          THIS PROCEDURE SELECTS A DIFFERENT STREAM;
```

Only one output stream may be selected at any time and it may not be the same as the input stream. If a call of SELECT OUTPUT is made part way through a line of output that partial line will be output as though the call of NEWLINE immediately preceded the call of SELECT OUTPUT.

THE CLOSE STREAM PROCEDURE

```
procedure CLOSE STREAM (N); value N; integer N;
comment   CHECKS THAT STREAM N IS NOT CURRENTLY IN USE FOR EITHER INPUT OR
          OUTPUT AND THEN RESETS THE STREAM;
```

If the stream is subsequently reselected for input the stream will be read (or reread) from the beginning. If it is selected for output the current contents will be destroyed.

Binary Input/Output Procedures

Binary files can be used for storing intermediate results of computations, either during the execution of one program, or for use by subsequent programs. The data stored is a direct copy of the contents of the core store. The file handling procedure calls all make explicit references to the logical channel being used, there is no equivalent to the currently selected stream used for character I/O. The variables accessed by binary I/O calls must be held in arrays and the read and write procedures require a specification of such.

```
Example:  array X(1:500);
          PUTSQ(2,X);
```

In this example the 500 elements of array X will be written out to the sequential file defined for channel 2.

Sequential Access Binary Files

Sequential Access Files (SQFILES) may have either fixed or variable length records.

THE OPENSQ PROCEDURE

```
procedure  OPENSQ(CHANNEL); value CHANNEL; integer CHANNEL;

comment    OPENS A BINARY SEQUENTIAL I/O CHANNEL.  IT MUST BE CALLED BEFORE
           GETSQ OR PUTSQ IS CALLED FOR THAT CHANNEL.  IF A FILE IS CLOSED IN A
           PROGRAM AND THEN RE-OPENED IT WILL BE RESET TO THE START;
```

THE CLOSESQ PROCEDURE

```
procedure  CLOSESQ(CHANNEL); value CHANNEL; integer CHANNEL;

comment    ALL FILES ARE CLOSED AUTOMATICALLY AT THE END OF A JOB.  CLOSESQ IS
           ONLY REQUIRED IF IT IS NECESSARY TO RELEASE RESOURCES BEFORE THE END
           OF A JOB;
```

THE PUTSQ PROCEDURE

```
procedure  PUTSQ(CHANNEL,ARRAY); value CHANNEL; integer CHANNEL;
           real/integer/boolean array ARRAY;

comment    EACH CALL OF PUTSQ OUTPUTS ONE LOGICAL RECORD, THE LENGTH OF WHICH IS
           DETERMINED BY THE LENGTH OF THE ARRAY TO BE OUTPUT.  IF A MINIMUM
           AND/OR MAXIMUM RECORD SIZE FOR THE SQFILE HAS BEEN DEFINED AND THE
           LENGTH OF THE ARRAY TO BE WRITTEN IS INCOMPATIBLE WITH THIS RECORD
           SIZE THEN THE CALL ON PUTSQ WILL BE FAULTED.  THE PROCEDURE TAKES TWO
           PARAMETERS, THE CHANNEL NUMBER AND THE NAME OF THE ARRAY IN WHICH THE
           DATA WILL BE FOUND.  THE ARRAY MAY BE OF ANY TYPE;

Example:   PUTSQ(2,FRED)
```

THE GETSQ PROCEDURE

```
procedure  GETSQ(CHANNEL,ARRAY); value CHANNEL; integer CHANNEL;
           real/integer/boolean array ARRAY;

comment    READS A COMPLETE RECORD, NORMALLY WRITTEN OUT BY A CALL OF PUTSQ.
           THE FIRST PARAMETER DEFINES THE CHANNEL TO BE USED AND THE SECOND IS
           AN ARRAYNAME OF ANY TYPE INTO WHICH THE RECORD IS READ.  IF THE
           RECORD READ IS LARGER THAN THE ARRAY THEN THE RECORD WILL BE
           TRUNCATED.  IF THE RECORD IS SMALLER THEN THE CONTENT OF THE
           REMAINDER OF THE ARRAY IS UNDEFINED;

Example:   GETSQ(2,FRED)
```

THE RWNDSQ PROCEDURE

```
procedure  RWNDSQ(CHANNEL); value CHANNEL; integer CHANNEL;

comment    RESETS THE FILE TO START OF FIRST RECORD;
```

Direct Access Binary Files

Direct Access Files (DAFILES) are used for similar purposes to SQFILES. The main difference in the method of use is that records can be accessed in a random order.

DA Files have fixed length records. Records are referenced by their position in the file. The first record is record 1. The record size must be declared by the user in the relevant job control statement.

THE OPENDA PROCEDURE

```
procedure  OPENDA(CHANNEL); value CHANNEL; integer CHANNEL;

comment    THE PROCEDURE TAKES ONE PARAMETER, THE CHANNEL NUMBER OF THE FILE
           BEING OPENED.  IT MUST BE CALLED BEFORE A CALL OF GETDA OR PUTDA FOR
           THE FILE;
```

THE CLOSEDA PROCEDURE

```
procedure  CLOSEDA(CHANNEL);

comment    ALL DAFILES ARE CLOSED AUTOMATICALLY AT THE END OF A JOB SO THIS
           PROCEDURE IS RARELY NEEDED.  IT MAY BE REQUIRED TO MINIMISE THE
           NUMBER OF FILES WHICH ARE CONCURRENTLY OPEN;
```

THE PUTDA PROCEDURE

```
procedure  PUTDA(CHANNEL,SECT,ARRAY); value CHANNEL;
           integer CHANNEL,SECT; real/integer/boolean array name ARRAY;

comment    THIS PROCEDURE IS USED TO WRITE DATA FROM AN ARRAY IN STORE TO A
           FILE.  A CALL OF PUTDA WILL RESULT IN ONE OR MORE RECORDS BEING
           WRITTEN, DEPENDING ON THE NUMBER OF BYTES IN THE ARRAY DEFINED.  IF
           THE ARRAY WRITTEN IS NOT AN EXACT MULTIPLE OF THE DEFINED RECORD SIZE
           THEN THE CONTENTS OF THE END OF THE LAST RECORD WRITTEN WILL BE
           UNDEFINED.  THE FIRST PARAMETER IS AN INTEGER EXPRESSION WHICH
           DEFINES THE CHANNEL NUMBER, THE SECOND IS THE NAME OF AN INTEGER
           VARIABLE.  ON ENTRY THIS SHOULD CONTAIN THE NUMBER OF THE FIRST
           RECORD TO BE WRITTEN.  ON EXIT IT WILL CONTAIN THE NUMBER OF THE
           RECORD AFTER THE LAST RECORD WRITTEN BY THIS CALL.  THE THIRD
           PARAMETER IS THE NAME OF THE ARRAY TO BE WRITTEN AND CAN BE OF ANY
           TYPE;
```

THE GETDA PROCEDURE

```
procedure GETDA(CHANNEL,SECT,ARRAY); value CHANNEL;
          integer CHANNEL,SECT; real/integer/boolean array name ARRAY;

comment   THIS PROCEDURE IS USED TO READ DATA WRITTEN BY PUTDA. THE PARAMETERS
          ARE USED IN THE SAME WAY AND IF AN ATTEMPT IS MADE TO READ INTO AN
          ARRAY WHICH IS NOT AN EXACT MULTIPLE OF THE RECORD SIZE DEFINED, THE
          ARRAY IS FILLED AND THE REST OF THE LAST RECORD IS IGNORED;
```

THE IFIP PROCEDURES

The International Federation for Information Processing has proposed a set of I/O routines for ALGOL and the compiler recognises the following selection of these. Calls on these procedures are translated into calls on the appropriate standard procedures as far as possible. This should suffice to enable existing programs containing calls on these routines to be run on EMAS with the minimum of alteration.

Since the routines are quite restricted and provide no format control, it is assumed that programmers will not wish to use them, and thus only a summary of each is provided below.

THE ININTEGER PROCEDURE

```
procedure ININTEGER (CHANNEL, I); value CHANNEL;
integer   CHANNEL, I;
comment   THIS PROCEDURE USES THE SELECT INPUT PROCEDURE TO SELECT THE
          INPUT STREAM, CHANNEL, IF NOT ALREADY SELECTED, AND THEN USES
          PROCEDURE READ TO INPUT THE NEXT NUMBER AND ASSIGN IT TO I;
```

THE OUTINTEGER PROCEDURE

```
procedure OUTINTEGER (CHANNEL, I); value CHANNEL, I;
integer   CHANNEL, I;
comment   THIS PROCEDURE USES THE SELECT OUTPUT PROCEDURE TO SELECT THE
          OUTPUT STREAM, CHANNEL, IF NOT ALREADY SELECTED, AND OUTPUTS I IN
          THE FORM PRINT (I, 10, 0) FOLLOWED BY A SEMI-COLON AND A NEWLINE;
```

THE INREAL PROCEDURE

```
procedure INREAL (CHANNEL, X); value CHANNEL;
integer   CHANNEL; real X;
comment   THIS PROCEDURE USES THE SELECT INPUT PROCEDURE TO SELECT THE
          INPUT STREAM, CHANNEL, IF NOT ALREADY SELECTED, AND THEN USES
          PROCEDURE READ TO INPUT THE NEXT NUMBER AND ASSIGN IT TO X;
```

THE OUTREAL PROCEDURE

```
procedure OUTREAL (CHANNEL, X); value CHANNEL, X;
integer   CHANNEL; real X;
comment   THIS PROCEDURE USES THE SELECT OUTPUT PROCEDURE TO SELECT THE
          OUTPUT STREAM, CHANNEL, IF NOT ALREADY SELECTED, AND OUTPUTS X IN
          THE FORM PRINT (X,0,10) FOLLOWED BY A SEMI-COLON AND A NEWLINE;
```

THE INSYMBOL PROCEDURE

```
procedure INSYMBOL (CHANNEL, STR, INT);
value     CHANNEL; integer CHANNEL, INT;
string    STR;
comment   THIS PROCEDURE USES SELECT INPUT TO SELECT INPUT STREAM, CHANNEL,
          UNLESS IT IS ALREADY SELECTED, AND THEN SETS INT TO THE VALUE
          CORRESPONDING TO THE FIRST POSITION IN STR OF THE CURRENT INPUT
          CHARACTER. THUS INT IS SET TO 1 IF THE FIRST CHARACTER OF THE
          STRING CORRESPONDS TO THE CURRENT CHARACTER ON THE INPUT STREAM.
          IF THE CURRENT CHARACTER IS NOT GIVEN IN THE STRING, STR, THEN
          INT IS SET TO ZERO. THE STREAM POINTER IS UPDATED TO POINT TO
          THE NEXT CHARACTER;
```

THE OUTSYMBOL PROCEDURE

```
procedure OUTSYMBOL (CHANNEL, STR, INT);
value     CHANNEL, INT; integer CHANNEL, INT;
string    STR;
comment   THIS PROCEDURE USES SELECT OUTPUT TO SELECT OUTPUT STREAM,
          CHANNEL, IF NOT ALREADY SELECTED, AND THEN OUTPUTS THE CHARACTER
          IN THE STRING, STR, TO WHICH THE VALUE OF INT POINTS. THUS IF
          INT HAS VALUE 1, THE FIRST CHARACTER OF THE STRING, STR, IS
          OUTPUT;
```

THE LENGTH PROCEDURE

```
integer procedure LENGTH (STR);
      string      STR;
      comment     LENGTH := NUMBER OF CHARACTERS IN THE OPEN STRING, STR,
                  EXCLUDING THE ENCLOSING STRING QUOTES;
```

THE OUTSTRING PROCEDURE

```
procedure OUTSTRING (CHANNEL, STR);
value     CHANNEL; integer CHANNEL;
string    STR;
comment   THIS PROCEDURE USES SELECT OUTPUT TO SELECT OUTPUT STREAM,
          CHANNEL, IF NOT ALREADY SELECTED, AND THEN OUTPUTS THE STRING,
          STR, FOLLOWED BY A SEMI-COLON AND A NEWLINE;
```

THE OUTTERMINATOR PROCEDURE

```
procedure OUTTERMINATOR (CHANNEL);
value     CHANNEL; integer CHANNEL;
comment   THIS PROCEDURE USES SELECT OUTPUT TO SELECT OUTPUT STREAM,
          CHANNEL, IF NOT ALREADY SELECTED, AND THEN OUTPUTS A SEMI-COLON
          FOLLOWED BY A NEWLINE;
```

THE STOP PROCEDURE

```
procedure STOP;
comment   CAUSES EXECUTION TO CEASE BY ARRANGING TO BRANCH TO THE 'END'
          CORRESPONDING TO THE FIRST 'BEGIN' OF THE PROGRAM;
```

THE MAXREAL PROCEDURE

```
real procedure    MAXREAL;
    comment       GIVES THE MAXIMUM ALLOWABLE POSITIVE REAL NUMBER;
```

THE MINREAL PROCEDURE

```
real procedure    MINREAL;
    comment       GIVES THE MINIMUM ALLOWABLE POSITIVE REAL NUMBER THAT IS
                  DISTINGUISHABLE FROM ZERO;
```

THE MAXINT PROCEDURE

```
integer procedure MAXINT;
        comment   GIVES THE MAXIMUM ALLOWABLE POSITIVE INTEGER;
```

THE EPSILON PROCEDURE

```
real procedure    EPSILON;
    comment       GIVES THE SMALLEST POSITIVE REAL NUMBER SUCH THAT THE
                  COMPUTATIONAL RESULT OF 1.0+EPSILON IS GREATER THAN 1.0
                  AND THE COMPUTATIONAL RESULT OF 1.0-EPSILON IS LESS THAN
                  1.0;
```

THE CPUTIME PROCEDURE

```
real procedure    CPUTIME;
    comment       GIVES THE CPU TIME IN SECONDS FROM AN ARBITRARY POINT
                  BEFORE THE START OF PROGRAM EXECUTION;
```

1900 ALGOL-COMPATIBLE PROCEDURES

The following routines are provided for the convenience of programmers with working 1900 programs.

```
real procedure    READ 1900;
        comment   THIS VERSION OF THE READ PROCEDURE ALLOWS A SINGLE SPACE
                  WITHIN THE NUMBER.  IT ALSO IGNORES NON-NUMERIC
                  CHARACTERS BEFORE THE START OF THE NUMBER AND ACCEPTS
                  '10' AND E IN PLACE OF & OR @ TO INTRODUCE AN EXPONENT;

Boolean procedure READ BOOLEAN;
        comment   SEARCHES THE INPUT STREAM FOR ONE OF THE BASIC SYMBOLS
                  'TRUE' OR 'FALSE';

procedure         PRINT 1900 (QUANTITY,M,N); value QUANTITY,M,N;
                  real QUANTITY; integer M,N;
        comment   THIS VERSION OF THE PRINT PROCEDURE IS IDENTICAL TO PRINT
                  EXCEPT THAT IT ADDS TWO SPACES AFTER THE NUMBER;

procedure         OUTPUT(QUANTITY); value QUANTITY; real QUANTITY;
        comment   OUTPUTS THE VALUE OF QUANTITY IN FLOATING POINT FORM WITH
                  10 SIGNIFICANT FIGURES FOLLOWED BY A SEMI-COLON AND A
                  NEWLINE;
```

```
procedure       WRITE BOOLEAN(QUANTITY); value QUANTITY;
                Boolean QUANTITY;
      comment   OUTPUTS THE VALUE OF THE BOOLEAN EXPRESSION TO THE
                CURRENTLY SELECTED OUTPUT STREAM AS 'TRUE' FOLLOWED BY
                TWO SPACES, OR 'FALSE' FOLLOWED BY ONE SPACE;

procedure       WRITE TEXT (STRING); string STRING;
      comment   OUTPUTS THE STRING TO THE CURRENTLY SELECTED OUTPUT
                STREAM. THE EDITING CHARACTERS S FOR SPACE, C FOR
                NEWLINE AND P FOR NEWPAGE ARE ACCEPTED, ENCLOSED IN
                FURTHER STRING QUOTES. AN EDITING CHARACTER MAY BE
                PRECEDED BY AN UNSIGNED INTEGER N TO INDICATE N SUCH
                CHARACTERS;

procedure       COPYTEXT (STRING); string STRING;
      comment   TRANSFERS SYMBOLS FROM THE INPUT STREAM TO THE OUTPUT
                STREAM UNTIL STRING IS ENCOUNTERED. STRING ITSELF IS NOT
                OUTPUT;

integer procedure READ CH;
      comment   READS ONE CHARACTER FROM THE CURRENTLY SELECTED INPUT
                STREAM. THE PROCEDURE READS ANY CHARACTERS FROM THE
                INPUT, INCLUDING THE 'INVISIBLE' CHARACTERS WHICH READ
                SYMBOL IGNORES;

integer procedure NEXT CH;
      comment   READS ONE CHARACTER AS FOR READ CH BUT LEAVES THE INPUT
                STREAM POSITIONED AT THAT CHARACTER SO THAT THE CHARACTER
                MAY BE READ AGAIN WITH A CALL OF READ CH OR A FURTHER
                CALL OF NEXT CH;

procedure       SKIP CH;
      comment   SKIPS OVER ONE CHARACTER OF THE INPUT STREAM WITHOUT
                READING IT;

procedure       PRINT CH (CH); value CH; integer CH;
      comment   OUTPUTS ONE CHARACTER (DEFINED BY THE LEAST SIGNIFICANT 7
                BITS OF CH) TO THE OUTPUT STREAM;
```

The above 1900 procedures, together with those of the standard Input/Output procedures, provide all the facilities of the basic 1900 ALGOL Input/Output. To avoid changing 1900 programs it is sometimes desirable to rename the 1900 routines so that they have the same names as the standard routines. This can be done by the following ALGOL statements.

```
real procedure  READ; external READ1900;

     procedure  PRINT(X,M,N); value X,M,N; real X; integer M,N;
                external PRINT1900;

     procedure  PAPERTHROW;
                NEWPAGE;

     procedure  NEWLINE(N); value N; integer N;
                NEWLINES(N);
```

```
procedure    SPACE(N); value N; integer N;
             SPACES(N);
```

The observant reader will note that there are no facilities for temporary storage of data on magnetic tape as are found in some ALGOL implementations. This is because EMAS ALGOL is designed to run on a paging system and the user program has a very large allocation of store in which to execute. The system paging routines will automatically remove unused data areas to backing store and bring them back when required. Users who have specialised Input/Output requirements or who wish to access unusual devices can take advantage of the ability of EMAS ALGOL to call routines written in IMP, and thus can make use of the very wide range of Input/Output facilities available to the IMP user.

CHAPTER 9 HARDWARE REPRESENTATION

EMAS ALGOL will accept two quite separate hardware representations of the ALGOL language. The default, and preferred, representation is based on that used for the Edinburgh IMP Language. This uses % as a special character to denote underlining. The underlining terminates at the first character which is not an upper case letter. This includes newlines and spaces. For example:

integer array may be represented either as
%INTEGER %ARRAY or as %INTEGERARRAY

The second is the ECMA (European Computer Manufacturers Association) hardware representation and involves placing the keywords between quotes.

For example:

integer array must be represented as
'INTEGER' 'ARRAY'

This latter alternative must be specifically requested by the Job Control Language before starting the compilation. On System 4 EMAS, this can be done by the foreground command PARM(BCD). Further information on Job Control Language is available in the appropriate User's Manual.

In both representations, lower case letters are accepted and recognised as distinct from the corresponding upper case letters. Thus label, LABEL and Label would be recognised as three distinct identifiers. However, keywords in either representation, standard function identifiers, and Input/Output function identifiers must be in upper case.

The full selection of hardware representations of the various basic symbols is given in the following table.

ALGOL basic symbol	IMP Representation	ECMA Representation
0 - 9	0 - 9	0 - 9
A - Z	A - Z	A - Z
a - z	a - z	a - z
10	@ or &	'10' or @ or &
.	.	.
real	%REAL	'REAL'
integer	%INTEGER	'INTEGER'
Boolean	%BOOLEAN	'BOOLEAN
array	%ARRAY	'ARRAY'
procedure	%PROCEDURE	'PROCEDURE'
switch	%SWITCH	'SWITCH'
label	%LABEL	'LABEL'
string	%STRING	'STRING'
comment	%COMMENT	'COMMENT'
if	%IF	'IF'
for	%FOR	'FOR'
goto	%GOTO	'GOTO'
begin	%BEGIN	'BEGIN'
own	%OWN	'OWN'
then	%THEN	'THEN'
while	%WHILE	'WHILE'

ALGOL basic symbol	IMP Representation	ECMA Representation
end	%END	'END'
value	%VALUE	'VALUE'
else	%ELSE	'ELSE'
step	%STEP	'STEP'
until	%UNTIL	'UNTIL'
false	%FALSE	'FALSE'
do	%DO	'DO'
true	%TRUE	'TRUE'
↑	** or ↑	'POWER' or ** or ↑
>	>	'GT' or >
<	<	'LT' or <
not (¬)	%NOT	'NOT'
(	(	(
)	)	)
[	[	'<' or [
]	]	'>' or]
[(')	{ or <	'('
] (')	} or >	')'
div (÷)	%DIV	'/'
≤	<=	<= or 'LE'
≥	>=	>= or 'GE'
and (∧)	%AND	'AND'
:	:	:
/	/	/
=	=	= or 'EQ'
or (V)	%OR	'OR'
'	'	'
×	*	*
impl (⊃)	%IMPL	'IMPL'
equiv (≡)	%EQUIV	'EQUIV'
:=	:=	:=
+	+	+
-	-	-
#	# or ¬=	# or 'NE' or¬ =

The following table gives the internal codes of all the characters; this is based on the ISO seven-bit code. It is strongly recommended that programmers do not make use of their knowledge of the internal codes if there is any possibility that their programs may be run on other machines. Characters marked with a * are non-printing characters, that is, characters which cannot be seen when the output is listed on a normal output device. Please note that the normal EMAS Input/Output routines remove all these characters on input.

INTERNAL CHARACTER CODE

0	NUL	*	32	SPACE		64	@		96	`	*
1	SOH	*	33	!(|)		65	A		97	a	
2	STX	*	34	"		66	B		98	b	
3	ETX	*	35	£(#)	(1)	67	C		99	c	
4	EOT	*	36	$	(2)	68	D		100	d	
5	ENQ	*	37	%		69	E		101	e	
6	ACK	*	38	&		70	F		102	f	
7	BEL	*	39	'		71	G		103	g	
8	BS	*	40	(		72	H		104	h	
9	HT	*	41	)		73	I		105	i	
10	LF(NL)		42	*		74	J		106	j	
11	VT	*	43	+		75	K		107	k	
12	FF	*	44	,		76	L		108	l	
13	CR	*	45	-		77	M		109	m	
14	SO	*	46	.		78	N		110	n	
15	SI	*	47	/		79	O		111	o	
16	DLE	*	48	0		80	P		112	p	
17	DC1	*	49	1		81	Q		113	q	
18	DC2	*	50	2		82	R		114	r	
19	DC3	*	51	3		83	S		115	s	
20	DC4	*	52	4		84	T		116	t	
21	NAK	*	53	5		85	U		117	u	
22	SYN	*	54	6		86	V		118	v	
23	ETB	*	55	7		87	W		119	w	
24	CAN	*	56	8		88	X		120	x	
25	EM		57	9		89	Y		121	y	
26	SUB		58	:		90	Z		122	z	
27	ESC	*	59	;		91	[		123	{	
28	FS	*	60	<		92	\(¬)		124	|	
29	GS	*	61	=		93	]		125	}	
30	RS	*	62	>		94	↑ (^)		126		
31	US	*	63	?		95	←		127	DEL	*

1) # is an alternative to £ here.

2) This is a currency symbol in ISO and other characters are possible.

CHAPTER 10 PROGRAM SEGMENTATION

The ALGOL Report assumes implicitly that an ALGOL program is a complete entity which is presented to the compiler together with all the functions that it requires, excepting the standard functions. It is, however, convenient if procedures which do not require access to global variables can be compiled independently, since they can then be called by more than one program without recompilation. EMAS ALGOL will accept for compilation a procedure that is not enclosed within a begin end block. The object module is given a name corresponding to the name of the outermost procedure. For example:

```
procedure JAMES (I,J);
value I; integer I,J;
begin
  .
  .
  .
end
```

would be compiled and known as JAMES. To access JAMES from a normal program, it is necessary to give a procedure heading and follow this with the word algol (instead of the procedure body). For example:

```
begin
procedure JAMES (I,J);
value I; integer I,J;
algol;
integer array P[1:10];
  .
  .

JAMES(23,K)
  .
end
```

As a further refinement it is possible to add the identifier of the called procedure after the pseudo basic symbol algol. If this is present, this name, rather than the dummy procedure name, is used to identify the required object module. Thus an alternative way to access JAMES is as follows:

```
begin
procedure JIM (I,J);
value I; integer I,J;
algol JAMES;
comment ANY CALL ON JIM IN THE PROGRAM FOLLOWING WILL NOW BE ROUTED
        TO THE EXTERNALLY COMPILED PROCEDURE JAMES;
```

N.B. (1) It is most important that the order and type of the formal parameters of the dummy procedure (JIM in the above example) correspond exactly with those of the separately compiled procedure. The compiler is unable to make any check.

(2) Any separately compiled procedure may, of course, access any other separately compiled procedure, provided the appropriate dummy heading is given.

(3) Several procedures not enclosed in a begin end block may be compiled together in one file, and they may reference each other without requiring dummy procedure headings.

MIXED LANGUAGE WORKING

It is possible for the ALGOL programmer to call IMP routines or Edinburgh FORTRAN subroutines if he so desires. A dummy procedure heading must be provided (see above) for each routine, and the pseudo basic symbol algol used above must be replaced by the pseudo basic symbol external.

The following table indicates the correspondence between IMP formal parameters and ALGOL formal parameters.

ALGOL	IMP
integer by value	%INTEGER
real by value	%LONGREAL
Boolean by value	%INTEGER
integer	%INTEGERNAME
real	%LONGREALNAME
Boolean	%INTEGERNAME
integer array by value	No equivalence
Boolean array by value	No equivalence
real array by value	No equivalence
integer array	%INTEGERARRAYNAME
array or real array	%LONGREALARRAYNAME
Boolean array	%INTEGERARRAYNAME
procedure	%ROUTINE
real procedure	%LONGREALFN
integer procedure	%INTEGERFN
Boolean procedure	%INTEGERFN
string	%STRINGNAME
switch	No equivalence
label	No equivalence

Notes on the above table:

(1) ALGOL Booleans appear to IMP as integers taking the value of 0 for false and -1 for true.

(2) Procedures can only be passed to IMP if all the parameters of the procedure can be represented in IMP.

(3) Actual parameters corresponding to formal parameters being passed to IMP by name must be variables or array elements of the correct type. Expressions can only be passed to IMP by value.

The following table indicates the correspondence between FORTRAN formal parameters and ALGOL formal parameters.

ALGOL	FORTRAN
integer	INTEGER*4
real	REAL*8
Boolean	INTEGER*4
anything else	no equivalence

Notes on the above table:

(1) ALGOL Booleans appear to FORTRAN as INTEGER*4 taking the value of 0 for false and -1 for true.

(2) Actual parameters being passed to FORTRAN must be local variables or array elements of the correct type. Expressions must be assigned to a local variable before being passed, i.e. there is no call by value.

(3) It is possible to pass one-dimensional arrays to FORTRAN if and only if the lower subscript is 1. In this case the passing of the first element by name to FORTRAN will have the required effect.

CHAPTER 11 COMPILER MESSAGES AND DIAGNOSTICS

The ALGOL Compiler can operate in Diagnostic or Optimising Mode. The former, which is the default, attempts to give helpful diagnostics at compile or run time, whilst the latter aims to produce the most efficient object program. A program should be thoroughly tested before being compiled in optimising mode, since the diagnostics produced on failure are likely to be unhelpful.

COMPILE TIME DIAGNOSTICS

The Compiler normally operates in three passes. The first pass reads in the program, produces the source listing and, where appropriate, translates from ECMA to IMP representation. Pass two checks the syntax of the program against that specified in the ALGOL Report and comments on any discrepancies. Pass three is only entered if pass two has found no syntactic errors. This pass performs extensive semantic checks on the program and generates the object program.

SYNTACTIC ERRORS

All syntactic errors are reported by a message of the form:

```
FAILED TO ANALYSE STATEMENT nnn:-
```

The statement is then printed out with a pointer (!) under the character which caused the analysis to fail. This normally gives a very clear indication of the error: For example:

```
%INTEGERARRY N[1:10]
```

would produce the error report

```
FAILED TO ANALYSE STATEMENT nnn:-
%INTEGERARRY N[1:10]
           !
```

In one or two cases, the consequences of the ALGOL syntax may not be immediately obvious. For example, it is clear from the syntax described in Section 3.1.1 of the ALGOL Report that subscript lists may only follow array identifiers. Thus, if subscripts follow an identifier which has not been declared as an array, they will be rejected with a pointer to the opening square bracket. Also, it is a consequence of the syntax described in section 4.2.4 of the Report that all constituents of a left part list must have been successfully declared as being of the correct type. If an incorrect or undeclared name appears, it will be faulted with the pointer set to the character after the last character of the erroneous name.

If no syntactic errors have been found, pass two terminates with the message

nnn STATEMENTS ANALYSED

Otherwise compilation terminates with the message

CODE GENERATION NOT ATTEMPTED

SEMANTIC WARNINGS AND ERRORS

Semantic messages, are of three grades of severity:

1. Warning messages, for information only.
2. Error messages, which do not stop compilation but cause the program to be marked as faulty.
3. Catastrophic errors, which cause compilation to cease immediately.

A warning message takes one of the following four forms:

1. WARNING:- NAME name NOT USED AT STMNT nnn

   This message occurs on an end statement. The indicated name has been declared in the block just terminated by the end statement, but has never been used.

2. WARNING:- LABEL label NOT USED AT STMNT nnn

   This message also occurs on an end statement. The indicated label has been found in the block just terminated by the nominated end statement but has never been referenced by a goto or switch statement, or passed as a parameter.

3. WARNING:- DUMMY STMNT COMPILED AT STMNT nnn

   A dummy statement has been used other than to place a label. Dummy statements often inadvertently enable erroneous statements to be compiled.

   For example

   ```
   for i:= 1 step 1 until 10 do; begin
   ```

   is a perfectly valid for loop controlling a dummy statement. It is, however, highly probable that the programmer meant

   ```
   for i:= 1 step 1 until 10 do begin
   ```

4. WARNING:- LABEL label PASSED BY NAME AT STMNT n
   This warning occurs on a procedure heading where label 'label' occurs in the value list. Value labels are not supported in this implementation and the program will be compiled as if the label were passed by name.

Compile time error messages take the form:

```
* sss FAULT nn (message)
```

where sss is the statement number of the offending statement
nn is the fault number
message is an abbreviated description of the fault.

The fault number is only provided for easy reference to the following list:

FAULT 2 (LABEL SET TWICE) LABEL

The current statement bears a label which has already been used to identify a statement in the current block.

FAULT 4 (SWITCH NAME NOT SET)

A statement of the form

goto NAME [I]

has been encountered where NAME is not currently declared as of type switch.

FAULT 5 (LABEL NAME IN EXPRSSN)

A name which is currently declared as a label has been found in an arithmetic or Boolean expression.

FAULT 7 (NAME SET TWICE)

The offending statement declares a name which is already declared or used as a label in the current block.

FAULT 8 (INVALID NAME IN VALUE LIST)

The offending statement is a procedure heading. Within this heading a name has appeared in the value list which does not appear in the formal parameter list for the procedure.

FAULT 9 (INVALID PARAMETER SPECIFICATION)

The offending statement is a procedure heading. A name appears in the specification part which has not appeared in the formal parameter list. This error can sometimes occur if the first begin of the procedure body is omitted or incorrectly positioned, so that the declarations of the procedure body are contiguous with the parameter specification.

FAULT 10 (PARAMETER INCORRECTLY SPECIFIED)

The offending statement is a procedure heading. One of the parameters in the parameter list has not had its type specified, or has been specified as a non-existent type (e.g. string by value).

FAULT 11 (LABEL NOT SET)

This fault refers to a statement containing a

goto LABEL

where LABEL is not currently declared as of type label. This fault can also appear on a switch statement since labels appearing in a switch list must either be labels in the current block or currently declared as labels from outer blocks.

FAULT 12 (LABEL NOT ACCESSIBLE) LABEL

This fault may appear either on a goto statement or on the statement labelled by LABEL. The label is on a statement controlled by a for statement and the offending goto statement is such that it could lead from outside a for statement to within the statement, the effect of which is undefined under Section 4.6.6 of the ALGOL Report.

FAULT 14 (TOO MANY ENDS)

The statement referred to is an end statement which matches the hypothetical begin of the conceptually enclosing block which contains the declarations of the standard functions. The compilation will stop at this point.

FAULT 15 (MISSING ENDS)

The program text has terminated without the end corresponding to the first begin of the program.

FAULT 16 (NAME NOT SET)

A name has been encountered which has not been declared and which is not the name of a standard function or standard Input/Output function.

FAULT 17 (NOT PROCEDURE NAME)

A statement having the syntax of a procedure statement has been encountered but the name is not currently declared as a procedure name.

FAULT 18 (WRONG NO OF SUBSCRIPTS)

A name currently declared as an array has been used but the number of subscripts provided does not correspond to the number of bound pairs given in the array declaration.

FAULT 19 (WRONG NO OF PARAMETERS)

A procedure statement has been encountered where the number of parameters in the actual parameter list is not the same as the number of formal parameters in the procedure declaration.

FAULT 20 (PARAMETRIC ARRAY WRONG DIMENSION) ARR

The array presented as actual parameter corresponding to the formal array ARR has not the expected number of dimensions. It is not clear from the ALGOL Report whether or not it is valid ALGOL to present arrays of different dimensions as actual parameters to the same array formal parameter. EMAS ALGOL insists that all arrays presented as actual parameters to the same formal parameter should have the same number of dimensions.

FAULT 21 (PARAMETRIC PROCEDURE NOT VALID) PROC

The procedure presented as actual parameter in place of the formal procedure PROC does not have its own formal parameter list identical to that specified for PROC by means of the special comment statement.

FAULT 22 (ACTUAL PARAMETER NOT PERMITTED) PARM

The actual parameter presented in place of the formal parameter PARM is not permitted by the rules of formal-actual parameter correspondence as defined in Chapter 5 and more formally in Sections 4.7.4 to 4.7.6 of the ALGOL Report.

FAULT 23 (PROCEDURE NAME IN EXPRSSN)

A name currently declared as a typeless procedure has been discovered in an arithmetic or Boolean expression.

FAULT 24 (VARIABLE IN BOOLEAN EXPRSSN)

A name currently declared of type real or type integer has been found in a Boolean expression.

FAULT 25 (FOR VARIABLE INCORRECT)

The controlled variable is not of type integer or type real or is a procedure name or a subscripted variable. It is not clear from the ALGOL Report whether or not a subscripted variable is permissible as a controlled variable. EMAS ALGOL follows the recommendations of the IFIP working party and does not allow controlled variables to be subscripted, since this would be likely to result in the exact action of the loop being critically dependent on where in the loop the subscript is evaluated, thus producing programs which give quite different answers on different compilers.

FAULT 26 (DIV OPERANDS NOT INTEGER)

The integer division operator has been encountered with operands of type real.

FAULT 27 (LOCAL IN ARRAY BOUND)

This fault refers to an array declaration. One of the variables in the bound pair list for the array declaration is declared in the current block. This is clearly contrary to Section 5.2.4.2. of the ALGOL Report.

FAULT 29 (INVALID NAME IN LEFT PART LIST)

The left part list contains a procedure name but the assignment is not within the body of the procedure or the procedure is not a type procedure or the name is of a different type from others in the left part list. Assignments to a procedure name are only valid with typed procedures and within the procedure body.

FAULT 34 (TOO MANY LEVELS)

The current depth of nested blocks exceeds the compiler limit of 31.

FAULT 35 (TOO MANY PROCEDURE LEVELS)

The current depth of nested procedures exceeds the compiler limit of 6 for EMAS -ALGOL on System 4 or 12 for EMAS ALGOL on 2900.

FAULT 37 (ARRAY TOO MANY DIMENSIONS)

EMAS ALGOL limits arrays to 12 dimensions.

FAULT 40 (DECLARATION MISPLACED)

Declarations must be at the head of a block prior to any statements.

N.B. A dummy statement is a valid statement. Consequently, begin; integer I; would be faulted since the begin is followed by a dummy statement and the statement is followed by a declaration.

FAULT 42 (BOOLEAN VARIABLE IN EXPRSSN)

A name currently declared as of type Boolean has been found in an arithmetic expression.

FAULT 43 (ARRAY INSIDE OUT)

An array with constant bounds has been discovered where the upper bound is less than the lower bound. This is treated in EMAS ALGOL as an error, although some ALGOL experts maintain that it merely denotes an array of zero elements.

FAULT 47 (ILLEGAL ELSE)

An else follows an end but the begin corresponding to the end does not follow a then.

FAULT 48 (SUB CHAR IN STMNT)

The substitute character, denoting an invalid character, has been found in the program text. The statement is ignored.

FAULT 57 (BEGIN MISSING)

Declarations have been encountered within the hypothetical block which surrounds the user's program. The first begin has probably been omitted.

FAULT 99 (ADDRESSABILITY)

The compiler is unable to address a portion of the program. This occurs if the program exceeds 256K bytes of compiled code (K = 1024) or if the program has very large own arrays.

The following five faults cause compilation to terminate immediately.

FAULT 102 (WORK FILE TOO BIG)

The output from the second pass of the compiler has overflowed the work file (currently 2 megabytes). The program must be segmented.

FAULT 103 (NAMES TOO LONG)
FAULT 104 (TOO MANY NAMES)

The compiler's dictionary has overflowed. The program must be segmented. The dictionary holds 1023 names of average length six characters, including standard function names.

FAULT 106 (STRING CONSTANT TOO LONG)

String constants are limited to 255 characters. This is adequate for all normal purposes and this fault often means that the closing string quote has been omitted.

FAULT 107 (ASL EMPTY)

The compiler tables are full. The program is too large and must be segmented.

N.B. The compiler tables have been designed to enable programs of 10,000 average statements to be compiled.

RUN TIME ERRORS

Various faults can occur during the execution of an ALGOL program and cause execution to cease. If the program was compiled in diagnostic mode, the diagnostic package is entered after the fault has been reported. The function of the diagnostic package is as follows:

1. To identify the logical block or procedure in which the failure occurred; for example

   PROCEDURE FRED STARTING AT STMNT 31

   or

   BLOCK STARTING AT STMNT 6

2. To print out the values at the time of failure of all the local scalar variables of the logical block or procedure. If no value has been assigned to a variable, this is indicated. Arrays and parameters passed by name are not printed, but parameters passed by value are treated as local scalars. For example:

   ```
   LOCAL SCALAR VARIABLES
   I = 1
   J = NOT ASSIGNED
   X = 4.77777 &-5
   ```

   Note that the names of scalar variables are truncated to eight characters for diagnostic purposes only.

3. To indicate the statement from which the logical block or procedure was entered.

4. To repeat the above three steps for the logical block from which the current logical block was entered, and to repeat this process until the first begin of the program has been reached. Thus procedures which have been used recursively will have the variables declared in each activation printed out.

The user can invoke the diagnostic package without terminating execution of his program by means of the pseudo procedure statement MONITOR. For example:

```
if PERCENT >100 then MONITOR
```

The following list give the main error messages produced by the ALGOL run time control package. It does not include all the failures which might be produced by the operating system, and for these the programmer is referred to the EMAS User's Manual.

```
INTEGER OVERFLOW
REAL OVERFLOW
```

On the evaluation of an expression a number has been generated which is too large to be contained in a register of the appropriate type.

```
NOT ENOUGH STORE
```

On executing a declaration, insufficient space is available in the store to accommodate the variable requested.

```
SQRT NEGATIVE
LOG NEGATIVE
```

A negative argument has been passed to the appropriate routine.

```
INPUT FILE ENDED
```

The current file of input data has been exhausted.

```
SYMBOL IN DATA S
```

In executing a read instruction, the non-numeric symbol S has been found.

```
DIVIDE ERROR
```

A division has been attempted that would cause overflow, usually a division by zero. Some operating systems do not distinguish this fault from INTEGER OVERFLOW or REAL OVERFLOW.

```
SUBSTITUTE CHARACTER IN DATA
```

If a line containing the substitute character (indicating an invalid code or corrupted character) is read, this fault will occur on attempting to read the first character of the line, NOT necessarily the incorrect character.

```
ILLEGAL EXPONENTIATION
```

An incorrect exponentiation has been attempted; that is, the base and exponent are both zero, or the exponent is of type real and the base is less than zero.

TRIG FN INACCURATE

The argument of a SIN, COS or TAN function is so large ($>10^7$) that the results are no longer guaranteed.

TAN TOO LARGE

The argument passed to the TAN function is so near an odd multiple of $\pi/2$ that an overflow condition would exist.

EXP TOO LARGE

The argument passed to the EXP function is so large that overflow would be caused by evaluation. This can also be caused by exponentiation since if y is real, x^y is evaluated as EXP(Y*LN(X)).

LIBRARY FN FAULT n

This is a general fault which can arise in any of the routines in the Edinburgh IMP and FORTRAN Library. This error can only occur if the programmer elects to use the facilities described in Chapter 10 for calling IMP and FORTRAN routines. A full list of the values of n will be found in the Edinburgh IMP and FORTRAN Library Manual.

INT PT TOO LARGE

An attempt to convert a real to an integer (either implicitly, or explicitly by a call of ENTIER) has failed because the integer part of the real number is greater than the largest allowed integer. (In some operating systems, this fault may be reported as OVERFLOW).

UNASSIGNED VARIABLE

An attempt has been made to use a variable to which no value has been assigned or whose value has become undefined as per the ALGOL Report, Section 4.6.5.

ARRAY BOUND FAULT

An array suffix has been found outside the declared bounds.

UNDEFINED STREAM

A SELECT INPUT or SELECT OUTPUT call attempts to define a Stream which has not been defined.

PARAM NOT DESTINATION

This fault occurs when trying to assign to a parameter passed by name. The actual parameter supplied is not a variable of the correct type and thus the assignment cannot be completed.

ADDRESS ERROR

Various types of addressing fault can cause this message. The commonest are trying to write to a protected area or trying to access unallocated virtual store. Possible reasons for this happening are:

1. An access outside the bounds of an array when the bound checking option has been turned off.

2. An incorrect dummy procedure heading for a separately compiled procedure.

3. Mixed language working with incompatible parameters.

CHAPTER 12 QUALIFICATIONS TO THE ALGOL REPORT FOR EMAS ALGOL

Appendix 1 is a reproduction of the complete ALGOL Report. The reader may wish to amend the copy so that it may be used as a reference for EMAS ALGOL. The following sections require amendment.

Section 3.2.4

The last sentence should be changed to:

These functions are available without explicit declaration.

Section 3.3.4.3

Replace paragraph 2 by:

Writing c for the numerical value of an unsigned integer,
i for any other numerical value of type integer,
r for a numerical value of type real,
a for a numerical value of either type integer or type real,
the result of exponentiation is given by the following rules:

a↑c If c>0, axax...xa (c times), of the same type as a.
If c=0, if a≠0, 1, of the same type as a
if a=0, failure

a↑i If i>0, a*a*...*a (i times), of type real
If i=0, if a≠0, 1, of type real
if a=0, failure

If i<0, if a≠0,1/(a*a*...*a) (the denominator having i factors) of type real
if a=0, failure

a↑r If a>0, EXP(r*LN(a)), of type real
If a=0, if r>0, 0.0, of type real
if r<0 undefined
If a<0, failure

Section 3.5.1

Delete |<unsigned integer> from the first definition so that the first line reads:

<label>::=<identifier>

Section 3.5.2

Delete the first and last examples.

Section 3.5.5

Delete the entire section.

Section 4.1.1

Alter the last definition to read:

<program>::=<unlabelled block> |
<unlabelled compound statement>

Section 4.7.3.1

Add the following sentence at the end of the section:

Labels cannot be called by value.

Section 4.7.5.4

Add label identifier to the list of formal parameters so that the section reads:

..... procedure identifier or a string or a label identifier, because these latter do not possess values.

Section 4.7.5.5

Add the following to the existing text:

Actual parameters corresponding to a real, integer or Boolean formal parameter called by name, and to which assignments are made, must have the same type as that specified for the formal parameter.

Where assignments are not made, the actual parameters are regarded as expressions, and provision is made for appropriate transfer functions to be invoked as necessary. The latter does not apply where the formal parameter called by name is specified as an array; in such a case the actual parameters must always be of the specified type.

Where a formal parameter is specified as a procedure, all the corresponding actual parameters must be procedures with equivalent specification parts (of course the identifiers do not have to be identical). Further, for a formal parameter specified as of type procedure, the actual parameters must all be type procedures of the same type.

Section 5

Replace paragraph 4 by:

Declarations of simple variables and arrays may be marked with the additional declarator own. The difference between own and non-own variables is as follows:

Non-own variables are attached to the block in which they are local, both with regard to the meaning of their identifiers and with regard to the existence of their values. Each entry to a block, whether recursive or not, will bring into existence a new set of the non-own variables of this block, the values of these variables being initially undefined. On exit from a block, all the non-own variables created at the corresponding entry are lost, both with regard to their

identifiers and values.

Own variables, on the other hand, are local only with respect to the existence of their identifiers. Where the existence of their values is concerned, they behave as though they were declared in the outermost block of the program. Thus every entry into a block, whether recursive or not, will make the same set of values of the own variables of the block accessible. On exit from a block the values of the own variables of the block will be preserved. As with non-own variables, own variables are initially undefined.

Dynamic own arrays are not allowed. All arrays declared with declarator own must have the lower bound and upper bound parts of the bound pair list elements given as integers.

Section 5.2.2

Delete the second example.

Section 5.2.5

Delete from the third line of this section the words:

even if an array is declared as own

Section 5.4.1

Replace the definition of <specification part> by

```
<specification part>::=<empty>|
      <specifier><identifier list>;<comment specification>|
      <comment specification>
```

Insert the definitions

```
<comment formal parameter list>::=<formal parameter>|
      <comment formal parameter list>,<formal parameter>
<comment formal parameter part>::=<empty>|
      (<comment formal parameter list>):
<comment value part>::=value <identifier list>:|<empty>
<comment specification part>::=<specifier>
      <identifier list>|
      <comment specification part>:<specifier>
      <identifier list>|<empty>
<comment specification>::=comment
      <comment formal parameter part>
      <comment value part><comment specification part>;|
      <empty>
```

Section 5.4.5

Replace the words

may be omitted
by
must be supplied

Add the following paragraph to the existing text:

The comment specification part is present when the current specifier is procedure or type procedure and is used to indicate the parameter structure of the formal procedure. The comment specification is taken to apply to all formal procedures declared in the same list. If several procedures of differing parameter structures are required the declarator procedure or type procedure must be repeated as necessary. If the delimiter comment occurs in this position and the text of the comment does not exactly satisfy the amended syntax of section 5.4.1 the comment will be regarded as a text comment in the sense of Section 2.3.

Revised report on the algorithmic language ALGOL 60

Dedicated to the memory of William Turanski

by

J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy,
P. Naur, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois,
J. H. Wegstein, A. van Wijngaarden, M. Woodger

Edited by

Peter Naur

The report gives a complete defining description of the international algorithmic language ALGOL 60. This is a language suitable for expressing a large class of numerical processes in a form sufficiently concise for direct automatic translation into the language of programmed automatic computers.

The introduction contains an account of the preparatory work leading up to the final conference, where the language was defined. In addition the notions reference language, publication language, and hardware representations are explained.

In the first chapter a survey of the basic constituents and features of the language is given, and the formal notation, by which the syntactic structure is defined, is explained.

The second chapter lists all the basic symbols, and the syntactic units known as *identifiers*, *numbers*, and *strings* are defined. Further, some important notions such as quantity and value are defined.

The third chapter explains the rules for forming expressions, and the meaning of these expressions. Three different types of expressions exist: arithmetic, Boolean (logical), and designational.

The fourth chapter describes the operational units of the language, known as *statements*. The basic statements are: *assignment* statements (evaluation of a formula), *go to* statements (explicit break of the sequence of execution of statements), dummy statements, and *procedure* statements (call for execution of a closed process, defined by a procedure declaration). The formation of more complex structures, having statement character, is explained. These include: *conditional* statements, *for* statements, *compound* statements, and *blocks*.

In the fifth chapter the units known as *declarations*, serving for defining permanent properties of the units entering into a process described in the language, are defined.

The report ends with two detailed examples of the use of the language, and an alphabetic index of definitions.

Contents

PAGE

Introduction 204

1. Structure of the language 205
   1.1 Formalism for syntactic description 206

2. Basic symbols, identifiers, numbers, and strings. Basic concepts 206
   2.1 Letters 206
   2.2 Digits. Logical values 206
   2.3 Delimiters 206
   2.4 Identifiers 207
   2.5 Numbers 207
   2.6 Strings 207
   2.7 Quantities, kinds and scopes 207
   2.8 Values and types 208

3. Expressions 208
   3.1 Variables 208
   3.2 Function designators 208
   3.3 Arithmetic expressions 209
   3.4 Boolean expressions 210
   3.5 Designational expressions. 211

4. Statements 211
   4.1 Compound statements and blocks 211
   4.2 Assignment statements 212
   4.3 Go to statements 213
   4.4 Dummy statements 213
   4.5 Conditional statements 213
   4.6 For statements 214
   4.7 Procedure statements 214

5. Declarations 216
   5.1 Type declarations 216
   5.2 Array declarations 216
   5.3 Switch declarations 217
   5.4 Procedure declarations 217

Examples of procedure declarations 218

Alphabetic index of definitions of concepts and syntactic units 220

Introduction

Background

After the publication*† of a preliminary report on the algorithmic language ALGOL, as prepared at a conference in Zürich in 1958, much interest in the ALGOL language developed.

As a result of an informal meeting held at Mainz in November 1958, about forty interested persons from several European countries held an ALGOL implementation conference in Copenhagen in February 1959. A "hardware group" was formed for working cooperatively right down to the level of the paper-tape code. This conference also led to the publication by Regnecentralen, Copenhagen, of an ALGOL *Bulletin*, edited by Peter Naur, which served as a forum for further discussion. During the June 1959 ICIP Conference in Paris several meetings, both formal and informal ones, were held. These meetings revealed some misunderstandings as to the intent of the group which was primarily responsible for the formulation of the language, but at the same time made it clear that there exists a wide appreciation of the effort involved. As a result of the discussions it was decided to hold an international meeting in January 1960 for improving the ALGOL language and preparing a final report. At a European ALGOL Conference in Paris in November 1959, which was attended by about fifty people, seven European representatives were selected to attend the January 1960 Conference, and they represented the following organizations: Association Française de Calcul, British Computer Society, Gesellschaft für Angewandte Mathematik und Mechanik, and Nederlands Rekenmachine Genootschap. The seven representatives held a final preparatory meeting at Mainz in December 1959.

Meanwhile, in the United States, anyone who wished to suggest changes or corrections to ALGOL was requested to send his comments to the ACM *Communications* where they were published. These comments then became the basis of consideration for changes in the ALGOL language. Both the SHARE and USE organizations established ALGOL working groups, and both organizations were represented on the ACM Committee on Programming Languages. The ACM Committee met in Washington in November 1959 and considered all comments on ALGOL that had been sent to the ACM *Communications*. Also, seven representatives were selected to attend the January 1960 international conference. These seven representatives held a final preparatory meeting in Boston in December 1959.

* Preliminary report—International Algebraic Language, *Comm. Assoc. Comp. Mach.*, Vol. 1, No. 12 (1958), p. 8.

† Report on the Algorithmic Language ALGOL by the ACM Committee on Programming Languages and the GAMM Committee on Programming, edited by A. J. Perlis and K. Samelson, *Numerische Mathematik* Bd. 1, S. 41–60 (1959).

January 1960 Conference

The thirteen representatives,* from Denmark, England, France, Germany, Holland, Switzerland, and the United States, conferred in Paris from 11 to 16 January 1960.

Prior to this meeting a completely new draft report was worked out from the preliminary report and the recommendations of the preparatory meetings by Peter Naur, and the conference adopted this new form as the basis for its report. The Conference then proceeded to work for agreement on each item of the report. The present report represents the union of the Committee's concepts and the intersection of its agreements.

April 1962 Conference (Edited by M. Woodger)

A meeting of some of the authors of ALGOL 60 was held on 2–3 April 1962, in Rome, Italy, through the facilities and courtesy of the International Computation Centre. The following were present:

Authors	*Advisers*	*Observer*
F. L. Bauer	M. Paul	W. L. van der Poel
J. Green	R. Franciotti	(Chairman,
C. Katz	P. Z. Ingerman	IFIP TC 2.1
R. Kogon		Working Group
(representing		ALGOL)
J. W. Backus)		
P. Naur		
K. Samelson	G. Seegmüller	
J. H. Wegstein	R. E. Utman	
A. van Wijngaarden		
M. Woodger	P. Landin	

The purpose of the meeting was to correct known errors in, attempt to eliminate apparent ambiguities in, and otherwise clarify the ALGOL 60 Report. Extensions to the language were not considered at the meeting. Various proposals for correction and clarification, that were submitted by interested parties in response to the Questionaire in ALGOL Bulletin No. 14, were used as a guide.

This report constitutes a supplement to the ALGOL 60 Report (*Incorporated with it to form the present revision —Ed.*) which should resolve a number of difficulties therein. Not all of the questions raised concerning the original report could be resolved. Rather than risk hastily drawn conclusions on a number of subtle points, which might create new ambiguities, the committee decided to report only those points which they unanimously felt could be stated in clear and unambiguous fashion.

Questions concerned with the following areas are left for further consideration by Working Group 2.1 of IFIP, in the expectation that current work on advanced programming languages will lead to better resolution:

1. Side effects of functions.
2. The *call by name* concept.

* William Turanski of the American group was killed by an automobile just prior to the January 1960 Conference.

3. **Own:** static or dynamic.
4. **For** statement: static or dynamic.
5. Conflict between specification and declaration.

The authors of the ALGOL 60 Report present at the Rome Conference, being aware of the formation of a Working Group on ALGOL by IFIP, accepted that any collective responsibility which they might have with respect to the development, specification and refinement of the ALGOL language will from now on be transferred to that body.

This report has been reviewed by IFIP TC 2 on Programming Languages in August 1962, and has been approved by the Council of the International Federation for Information Processing.

As with the preliminary ALGOL report, three different levels of language are recognized, namely a Reference Language, a Publication Language and several Hardware Representations.

Reference Language

1. It is the working language of the committee.
2. It is the defining language.
3. The characters are determined by ease of mutual understanding and not by any computer limitations, coder's notation, or pure mathematical notation.
4. It is the basic reference and guide for compiler builders.
5. It is the guide for all hardware representations.
6. It is the guide for transliterating from publication language to any locally appropriate hardware representations.
7. The main publications of the ALGOL language itself will use the reference representation.

Publication Language

1. The publication language admits variations of the reference language according to usage of printing and handwriting (e.g. subscripts, spaces, exponents, Greek letters).
2. It is used for stating and communicating processes.
3. The characters to be used may be different in different countries, but univocal correspondence with reference representation must be secured.

Hardware Representations

1. Each one of these is a condensation of the reference language enforced by the limited number of characters on standard input equipment.
2. Each one of these uses the character set of a particular computer, and is the language accepted by a translator for that computer.
3. Each one of these must be accompanied by a special set of rules for transliterating from Publication or Reference language.

For transliteration between the reference language and a language suitable for publications, among others, the following rules are recommended.

Reference language	*Publication language*
Subscript brackets [ ]	Lowering of the line between the brackets and removal of the brackets.
Exponentiation ↑	Raising of the exponent.
Parentheses ( )	Any form of parentheses, brackets, braces.
Basis of ten $_{10}$	Raising of the ten and of the following integral number, inserting of the intended multiplication sign.

Description of the Reference Language

Was sich überhaupt sagen lässt, lässt
sich klar sagen; und wovon man nicht
reden kann, darüber muss man schweigen.
Ludwig Wittgenstein.

1. Structure of the language

As stated in the Introduction, the algorithmic language has three different kinds of representations—reference, hardware, and publication—and the development described in the sequel is in terms of the reference representation. This means that all objects defined within the language are represented by a given set of symbols—and it is only in the choice of symbols that the other two representations may differ. Structure and content must be the same for all representations.

The purpose of the algorithmic language is to describe computational processes. The basic concept used for the description of calculating rules is the well-known arithmetic expression containing as constituents numbers, variables, and functions. From such expressions are compounded, by applying rules of arithmetic composition, self-contained units of the language—explicit formulae—called *assignment statements.*

To show the flow of computational processes, certain non-arithmetic statements and statement clauses are added which may describe, e.g. alternatives, or iterative repetitions of computing statements. Since it is necessary for the function of these statements that one statement refers to another, statements may be provided with labels. A sequence of statements may be enclosed between the statement brackets **begin** and **end** to form a compound statement.

Statements are supported by declarations which are not themselves computing instructions, but inform the translator of the existence and certain properties of objects appearing in statements, such as the class of numbers taken on as values by a variable, the dimension of an array of numbers, or even the set of rules defining a function. A sequence of declarations followed by a sequence of statements and enclosed between **begin** and **end** constitutes a *block*. Every declaration appears in a block in this way and is valid only for that block.

A *program* is a block or compound statement which is not contained within another statement and which makes no use of other statements not contained within it.

In the sequel the syntax and semantics of the language will be given.*

1.1 Formalism for syntactic description

The syntax will be described with the aid of metalinguistic formulae.† Their interpretation is best explained by an example:

⟨ab⟩ :: = (| [| ⟨ab⟩ (| ⟨ab⟩ ⟨d⟩

Sequences of characters enclosed in the brackets ⟨ ⟩ represent metalinguistic variables whose values are sequences of symbols. The marks :: = and | (the latter with the meaning of *or*) are metalinguistic connectives. Any mark in a formula, which is not a variable or a connective, denotes itself (or the class of marks which are similar to it). Juxtaposition of marks and/or variables in a formula signifies juxtaposition of the sequences denoted. Thus the formula above gives a recursive rule for the formation of values of the variable ⟨ab⟩. It indicates that ⟨ab⟩ may have the value (or [or that given some legitimate value of ⟨ab⟩, another may be formed by following it with the character (or by following it with some value of the variable ⟨d⟩. If the values of ⟨d⟩ are the decimal digits, some values of ⟨ab⟩ are:

[(((1(37(
(12345(
(((
[86

In order to facilitate the study, the symbols used for distinguishing the metalinguistic variables (i.e. the sequences of characters appearing within the brackets ⟨ ⟩ as ab in the above example) have been chosen to be words describing approximately the nature of the corresponding variable. Where words which have appeared in this manner are used elsewhere in the text they will refer to the corresponding syntactic definition. In addition some formulae have been given in more than one place.

Definition:

⟨empty⟩ :: =
(i.e. the null string of symbols).

2. Basic symbols, identifiers, numbers, and strings

Basic concepts

The reference language is built up from the following basic symbols:

⟨basic symbol⟩ :: = ⟨letter⟩ | ⟨digit⟩ | ⟨logical value⟩ | ⟨delimiter⟩

2.1 Letters

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

This alphabet may arbitrarily be restricted, or extended with any other distinctive character (i.e. character not coinciding with any digit, logical value or delimiter).

Letters do not have individual meaning. They are used for forming identifiers and strings* (cf. sections 2.4 Identifiers, 2.6 Strings).

2.2.1 *Digits*

⟨digit⟩ :: = 0|1|2|3|4|5|6|7|8|9

Digits are used for forming numbers, identifiers, and strings.

2.2.2 *Logical values*

⟨logical value⟩ :: = **true** | **false**

The logical values have a fixed obvious meaning.

2.3 Delimiters

⟨delimiter⟩ :: = ⟨operator⟩ | ⟨separator⟩ | ⟨bracket⟩ | ⟨declarator⟩ | ⟨specificator⟩

⟨operator⟩ :: = ⟨arithmetic operator⟩ | ⟨relational operator⟩ | ⟨logical operator⟩ | ⟨sequential operator⟩

⟨arithmetic operator⟩ :: = + | − | × | / | ÷ | ↑

⟨relational operator⟩ :: = < | ≤ | = | ≥ | > | ≠

⟨logical operator⟩ :: = ≡ | ⊃ | ∨ | ∧ | ¬

⟨sequential operator⟩ :: = **go to** | **if** | **then** | **else** | **for** | **do**†

⟨separator⟩ :: = , | . | $_{10}$ | : | ; | := | ␣ | **step** | **until** | **while** | **comment**

⟨bracket⟩ :: = (|) | [|] | ‘ | ’ | **begin** | **end**

* Whenever the precision of arithmetic is stated as being in general not specified, or the outcome of a certain process is left undefined or said to be undefined, this is to be interpreted in the sense that a program only fully defines a computational process if the accompanying information specifies the precision assumed, the kind of arithmetic assumed, and the course of action to be taken in all such cases as may occur during the execution of the computation.

† Cf. J. W. Backus, "The syntax and semantics of the proposed international algebraic language of the Zurich ACM–GAMM conference," ICIP, Paris, June 1959.

* It should be particularly noted that throughout the reference language underlining (*for typographical reasons* **bold type** *is used synonymously—Ed.*) is used for defining independent basic symbols (see sections 2.2.2 and 2.3). These are understood to have no relation to the individual letters of which they are composed. Within the present report underlining will be used for no other purpose.

† **do** is used in **for** statements. It has no relation whatsoever to the **do** of the preliminary report, which is not included in ALGOL 60.

⟨declarator⟩ :: = **own** | **Boolean** | **integer** | **real** | **array** | **switch** | **procedure**
⟨specificator⟩ :: = **string** | **label** | **value**

Delimiters have a fixed meaning which for the most part is obvious, or else will be given at the appropriate place in the sequel.

Typographical features such as blank space or change to a new line have no significance in the reference language. They may, however, be used freely for facilitating reading.

For the purpose of including text among the symbols of a program the following "comment" conventions hold:

The sequence of basic symbols:	is equivalent to
; **comment** ⟨any sequence not containing ; ⟩ ;	;
begin comment ⟨any sequence not containing ; ⟩ ; **begin end** ⟨any sequence not containing **end** or ; or **else**⟩	**begin end.**

By equivalence is here meant that any of the three structures shown in the left-hand column may, in any occurrence outside of strings, be replaced by the symbol shown on the same line in the right-hand column without any effect on the action of the program. It is further understood that the comment structure encountered first in the text when reading from left to right has precedence in being replaced over later structures contained in the sequence.

2.4 Identifiers

2.4.1 *Syntax*

⟨identifier⟩ :: = ⟨letter⟩ | ⟨identifier⟩ ⟨letter⟩ | ⟨identifier⟩ ⟨digit⟩

2.4.2 *Examples*

q
Soup
V17a
a34kTMNs
MARILYN

2.4.3 *Semantics*

Identifiers have no inherent meaning, but serve for the identification of simple variables, arrays, labels, switches, and procedures. They may be chosen freely (cf., however, section 3.2.4 Standard functions).

The same identifier cannot be used to denote two different quantities except when these quantities have disjoint scopes as defined by the declarations of the program (cf. section 2.7 Quantities, kinds and scopes, and section 5 Declarations).

2.5 Numbers

2.5.1 *Syntax*

⟨unsigned integer⟩ :: = ⟨digit⟩ | ⟨unsigned integer⟩ ⟨digit⟩
⟨integer⟩ :: = ⟨unsigned integer⟩ | + ⟨unsigned integer⟩ | − ⟨unsigned integer⟩
⟨decimal fraction⟩ :: = . ⟨unsigned integer⟩
⟨exponent part⟩ :: = $_{10}$ ⟨integer⟩
⟨decimal number⟩ :: = ⟨unsigned integer⟩ | ⟨decimal fraction⟩ | ⟨unsigned integer⟩ ⟨decimal fraction⟩
⟨unsigned number⟩ :: = ⟨decimal number⟩ | ⟨exponent part⟩ | ⟨decimal number⟩ ⟨exponent part⟩
⟨number⟩ :: = ⟨unsigned number⟩ | + ⟨unsigned number⟩ | − ⟨unsigned number⟩

2.5.2 *Examples*

0	−200.084	$-.083_{10}-02$
! ?	$+07.43_{10}8$	$-_{10}7$
.5384	$9.34_{10}+10$	$_{10}-4$
+0.7300	$.2_{10}-4$	$+_{10}+5$

2.5.3 *Semantics*

Decimal numbers have their conventional meaning. The exponent part is a scale factor expressed as an integral power of 10.

2.5.4 *Types*

Integers are of type **integer.** All other numbers are of type **real** (cf. section 5.1 Type declarations).

2.6 Strings

2.6.1 *Syntax*

⟨proper string⟩ :: = ⟨any sequence of basic symbols not containing ´ or ` ⟩ | ⟨empty⟩
⟨open string⟩ :: = ⟨proper string⟩ | ´⟨open string⟩` | ⟨open string⟩ ⟨open string⟩
⟨string⟩ :: = ´⟨open string⟩`

2.6.2 *Examples*

´5*k*, , − ´ [[[´ ∧ = / : ` *Tt* ` `
´ . . *This* ␣ *is* ␣ *a* ␣ ´ *string* ` `

2.6.3 *Semantics*

In order to enable the language to handle arbitrary sequences of basic symbols the string quotes ´ and ` are introduced. The symbol ␣ denotes a space. It has no significance outside strings.

Strings are used as actual parameters of procedures (c.f. sections 3.2 Function designators and 4.7 Procedure statements).

2.7 Quantities, kinds and scopes

The following kinds of *quantities* are distinguished: simple variables, arrays, labels, switches, and procedures.

The scope of a quantity is the set of statements and expressions in which the declaration of the identifier associated with that quantity is valid. For labels see section 4.1.3.

2.8 Values and types

A *value* is an ordered set of numbers (special case: a single number), an ordered set of logical values (special case: a single logical value), or a label.

Certain of the syntactic units are said to possess values. These values will in general change during the execution of the program. The values of expressions and their constituents are defined in section 3. The value of an array identifier is the ordered set of values of the corresponding array of subscripted variables (cf. section 3.1.4.1).

The various *types* (**integer, real, Boolean**) basically denote properties of values. The types associated with syntactic units refer to the values of these units.

3. Expressions

In the language the primary constituents of the programs describing algorithmic processes are *arithmetic*, *Boolean*, and *designational*, expressions. Constituents of these expressions, except for certain delimiters, are logical values, numbers, variables, function designators, and elementary arithmetic, relational, logical, and sequential, operators. Since the syntactic definition of both variables and function designators contains expressions, the definition of expressions, and their constituents, is necessarily recursive.

⟨expression⟩ :: = ⟨arithmetic expression⟩ | ⟨Boolean expression⟩ | ⟨designational expression⟩

3.1 Variables

3.1.1 *Syntax*

⟨variable identifier⟩ :: = ⟨identifier⟩
⟨simple variable⟩ :: = ⟨variable identifier⟩
⟨subscript expression⟩ :: = ⟨arithmetic expression⟩
⟨subscript list⟩ :: = ⟨subscript expression⟩ |
⟨subscript list⟩, ⟨subscript expression⟩
⟨array identifier⟩ :: = ⟨identifier⟩
⟨subscripted variable⟩ :: =
⟨array identifier⟩ [⟨subscript list⟩]
⟨variable⟩ :: = ⟨simple variable⟩ |
⟨subscripted variable⟩

3.1.2 *Examples*

epsilon
detA
*a*17
Q[7, 2]
x[*sin*(*n*×*pi*/2), *Q*[3, *n*, 4]]

3.1.3 *Semantics*

A variable is a designation given to a single value. This value may be used in expressions for forming other values and may be changed at will by means of assignment statements (section 4.2). The type of the value of a particular variable is defined in the declaration for the variable itself (cf. section 5.1 Type declarations) or for the corresponding array identifier (cf. section 5.2 Array declarations).

3.1.4 *Subscripts*

3.1.4.1 Subscripted variables designate values which are components of multidimensional arrays (cf. section 5.2 Array declarations). Each arithmetic expression of the subscript list occupies one subscript position of the subscripted variable, and is called a *subscript*. The complete list of subscripts is enclosed in the subscript brackets [ ]. The array component referred to by a subscripted variable is specified by the actual numerical value of its subscripts (cf. section 3.3 Arithmetic expressions).

3.1.4.2 Each subscript position acts like a variable of type **integer** and the evaluation of the subscript is understood to be equivalent to an assignment to this fictitious variable (cf. section 4.2.4). The value of the subscripted variable is defined only if the value of the subscript expression is within the subscript bounds of the array (cf. section 5.2 Array declarations).

3.2 Function designators

3.2.1 *Syntax*

⟨procedure identifier⟩ :: = ⟨identifier⟩
⟨actual parameter⟩ :: = ⟨string⟩ | ⟨expression⟩ |
⟨array identifier⟩ | ⟨switch identifier⟩ |
⟨procedure identifier⟩
⟨letter string⟩ :: = ⟨letter⟩ | ⟨letter string⟩ ⟨letter⟩
⟨parameter delimiter⟩ :: = , |) ⟨letter string⟩ : (
⟨actual parameter list⟩ :: = ⟨actual parameter⟩ |
⟨actual parameter list⟩ ⟨parameter delimiter⟩
⟨actual parameter⟩
⟨actual parameter part⟩ :: = ⟨empty⟩ |
(⟨actual parameter list⟩)
⟨function designator⟩ :: = ⟨procedure identifier⟩
⟨actual parameter part⟩

3.2.2 *Examples*

sin (*a* − *b*)
J(*v* + *s*, *n*)
R
S(*s* − 5) *Temperature*: (*T*) *Pressure*: (*P*)
Compile (´ := `) *Stack*: (*Q*)

3.2.3 *Semantics*

Function designators define single numerical or logical values, which result through the application of given sets of rules defined by a procedure declaration (cf. section 5.4 Procedure declarations) to fixed sets of actual parameters. The rules governing specification of actual parameters are given in section 4.7 Procedure statements. Not every procedure declaration defines the value of a function designator.

3.2.4 *Standard functions*

Certain identifiers should be reserved for the standard functions of analysis, which will be expressed as procedures. It is recommended that this reserved list should contain:

abs(*E*)	for the modulus (absolute value) of the value of the expression *E*
sign(*E*)	for the sign of the value of *E*(+ 1 for *E* > 0, 0 for *E* = 0, − 1 for *E* < 0)
sqrt(*E*)	for the square root of the value of *E*
sin(*E*)	for the sine of the value of *E*
cos(*E*)	for the cosine of the value of *E*
arctan(*E*)	for the principal value of the arctangent of the value of *E*
ln(*E*)	for the natural logarithm of the value of *E*
exp(*E*)	for the exponential function of the value of *E* (e^E).

These functions are all understood to operate indifferently on arguments both of type **real** and **integer.** They will all yield values of type **real,** except for *sign*(*E*) which will have values of type **integer.** In a particular representation these functions may be available without explicit declarations (cf. section 5 Declarations).

3.2.5 *Transfer functions*

It is understood that transfer functions between any pair of quantities and expressions may be defined. Among the standard functions it is recommended that there be one, namely

entier(*E*) ,

which "transfers" an expression of real type to one of integer type, and assigns to it the value which is the largest integer not greater than the value of *E*.

3.3 Arithmetic expressions

3.3.1 *Syntax*

⟨adding operator⟩ :: = + | −
⟨multiplying operator⟩ :: = × | / | ÷
⟨primary⟩ :: = ⟨unsigned number⟩ | ⟨variable⟩ | ⟨function designator⟩ | (⟨arithmetic expression⟩)
⟨factor⟩ :: = ⟨primary⟩ | ⟨factor⟩ ↑ ⟨primary⟩
⟨term⟩ :: = ⟨factor⟩ | ⟨term⟩ ⟨multiplying operator⟩ ⟨factor⟩
⟨simple arithmetic expression⟩ :: = ⟨term⟩ | ⟨adding operator⟩ ⟨term⟩ | ⟨simple arithmetic expression⟩ ⟨adding operator⟩ ⟨term⟩
⟨if clause⟩ :: = **if** ⟨Boolean expression⟩ **then**
⟨arithmetic expression⟩ :: = ⟨simple arithmetic expression⟩ | ⟨if clause⟩ ⟨simple arithmetic expression⟩ **else** ⟨arithmetic expression⟩

3.3.2 *Examples*

Primaries:
$7.394_{10}-8$
sum
$w[i + 2, 8]$
$cos\,(y + z \times 3)$
$(a - 3/y + vu \uparrow 8)$

Factors:
omega
$sum \uparrow cos\,(y + z \times 3)$
$7.394_{10} - 8 \uparrow w[i + 2, 8] \uparrow (a - 3/y + vu \uparrow 8)$

Terms:
U
$omega \times sum \uparrow cos\,(y + z \times 3)/7.394_{10} - 8 \uparrow w[i + 2, 8] \uparrow (a - 3/y + vu \uparrow 8)$

Simple arithmetic expression:
$U - Yu + omega \times sum \uparrow cos\,(y + z \times 3)/7.394_{10} - 8 \uparrow w[i + 2, 8] \uparrow (a - 3/y + vu \uparrow 8)$

Arithmetic expressions:
$w \times u - Q(S + Cu) \uparrow 2$
if $q > 0$ **then** $S + 3 \times Q/A$ **else** $2 \times S + 3 \times q$
if $a < 0$ **then** $U + V$ **else if** $a \times b > 17$ **then** U/V **else if** $k \neq y$ **then** V/U **else** 0
$a \times sin\,(omega \times t)$
$0.57_{10}\ 12 \times a\ [N \times (N - 1)/2, 0]$
$(A \times arctan(y) + Z) \uparrow (7 + Q)$
if q **then** $n - 1$ **else** n
if $a < 0$ **then** A/B **else if** $b = 0$ **then** B/A **else** z

3.3.3 *Semantics*

An arithmetic expression is a rule for computing a numerical value. In case of simple arithmetic expressions this value is obtained by executing the indicated arithmetic operations on the actual numerical values of the primaries of the expression, as explained in detail in section 3.3.4 below. The actual numerical value of a primary is obvious in the case of numbers. For variables it is the current value (assigned last in the dynamic sense), and for function designators it is the value arising from the computing rules defining the procedure (cf. section 5.4.4 Values of function designators) when applied to the current values of the procedure parameters given in the expression. Finally, for arithmetic expressions enclosed in parentheses the value must through a recursive analysis be expressed in terms of the values of primaries of the other three kinds.

In the more general arithmetic expressions, which include **if** clauses, one out of several simple arithmetic expressions is selected on the basis of the actual values of the Boolean expressions (cf. section 3.4 Boolean expressions). This selection is made as follows: The Boolean expressions of the **if** clauses are evaluated one by one in sequence from left to right until one having the value **true** is found. The value of the arithmetic expression is then the value of the first arithmetic expression following this Boolean (the largest arithmetic expression found in this position is understood). The construction:

else ⟨simple arithmetic expression⟩

is equivalent to the construction:

else if true then ⟨simple arithmetic expression⟩

3.3.4 *Operators and types*

Apart from the Boolean expressions of **if** clauses, the constituents of simple arithmetic expressions must be of types **real** or **integer** (cf. section 5.1 Type declarations). The meaning of the basic operators and the types of the expressions to which they lead are given by the following rules:

3.3.4.1 The operators $+$, $-$, and $\times$ have the conventional meaning (addition, subtraction, and multiplication). The type of the expression will be **integer** if both of the operands are of **integer** type, otherwise **real**.

3.3.4.2 The operations ⟨term⟩ / ⟨factor⟩ and ⟨term⟩ ÷ ⟨factor⟩ both denote division, to be understood as a multiplication of the term by the reciprocal of the factor with due regard to the rules of precedence (cf. section 3.3.5). Thus, for example

$$a/b \times 7/(p - q) \times v/s$$

means

$$((((a \times (b^{-1})) \times 7) \times ((p - q)^{-1})) \times v) \times (s^{-1})$$

The operator / is defined for all four combinations of types **real** and **integer** and will yield results of **real** type in any case. The operator ÷ is defined only for two operands both of type **integer** and will yield a result of type **integer**, mathematically defined as follows:

$$a \div b = sign\,(a/b) \times entier\,(abs(a/b))$$

(cf. sections 3.2.4 and 3.2.5).

3.3.4 3 The operation ⟨factor⟩ ↑ ⟨primary⟩ denotes exponentiation, where the factor is the base and the primary is the exponent. Thus, for example

$2 \uparrow n \uparrow k$ means $(2^n)^k$

while

$2 \uparrow (n \uparrow m)$ means $2^{(n^m)}$

Writing *i* for a number of **integer** type, *r* for a number of **real** type, and *a* for a number of either **integer** or **real** type, the result is given by the following rules:

$a \uparrow i$	If $i > 0$, $a \times a \times \ldots \times a$ (i times), of the same type as a.
	If $i = 0$, if $a \neq 0$, 1, of the same type as a, if $a = 0$, undefined.
	If $i < 0$, if $a \neq 0$, $1/(a \times a \times \ldots \times a)$ (the denominator has $-i$ factors), of type **real**, if $a = 0$, undefined.
$a \uparrow r$	If $a > 0$, $exp\,(r \times ln(a))$, of type **real**.
	If $a = 0$, if $r > 0$, 0 . 0, of type **real**, if $r < 0$, undefined.
	If $a < 0$, always undefined.

3.3.5 *Precedence of operators*

The sequence of operations within one expression is generally from left to right, with the following additional rules:

3.3.5.1 According to the syntax given in section 3.3.1 the following rules of precedence hold:

first: ↑
second: × / ÷
third: + −

3.3.5.2 The expression between a left parenthesis and the matching right parenthesis is evaluated by itself and this value is used in subsequent calculations. Consequently the desired order of execution of operations within an expression can always be arranged by appropriate positioning of parentheses.

3.3.6 *Arithmetics of* **real** *quantities*

Numbers and variables of type **real** must be interpreted in the sense of numerical analysis, i.e. as entities defined inherently with only a finite accuracy. Similarly, the possibility of the occurrence of a finite deviation from the mathematically defined result in any arithmetic expression is explicitly understood. No exact arithmetic will be specified, however, and it is indeed understood that different hardware representations may evaluate arithmetic expressions differently. The control of the possible consequences of such differences must be carried out by the methods of numerical analysis. This control must be considered a part of the process to be described, and will therefore be expressed in terms of the language itself.

3.4 Boolean expressions

3.4.1 *Syntax*

⟨relational operator⟩ :: = < | ⩽ | = | ⩾ | > | ≠
⟨relation⟩ :: =
⟨simple arithmetic expression⟩ ⟨relational operator⟩ ⟨simple arithmetic expression⟩
⟨Boolean primary⟩ :: = ⟨logical value⟩ | ⟨variable⟩ | ⟨function designator⟩ | ⟨relation⟩ | (⟨Boolean expression⟩)
⟨Boolean secondary⟩ :: = ⟨Boolean primary⟩ | ¬ ⟨Boolean primary⟩
⟨Boolean factor⟩ :: = ⟨Boolean secondary⟩ | ⟨Boolean factor⟩ ∧ ⟨Boolean secondary⟩
⟨Boolean term⟩ :: = ⟨Boolean factor⟩ | ⟨Boolean term⟩ ∨ ⟨Boolean factor⟩
⟨implication⟩ :: = ⟨Boolean term⟩ | ⟨implication⟩ ⊃ ⟨Boolean term⟩
⟨simple Boolean⟩ :: = ⟨implication⟩ | ⟨simple Boolean⟩ ≡ ⟨implication⟩
⟨Boolean expression⟩ :: = ⟨simple Boolean⟩ | ⟨if clause⟩ ⟨simple Boolean⟩ **else** ⟨Boolean expression⟩

3.4.2 *Examples*

$x = -2$
$Y > V \vee z < q$
$a + b > -5 \wedge z - d > q \uparrow 2$
$p \wedge q \vee x \neq y$
$g \equiv \neg a \wedge b \wedge \neg c \vee d \vee e \supset \neg f$
if $k < 1$ **then** $s > w$ **else** $h < c$
if if if a **then** b **else** c **then** d **else** f **then** g **else** $h < k$

3.4.3 *Semantics*

A Boolean expression is a rule for computing a logical value. The principles of evaluation are entirely analogous to those given for arithmetic expressions in section 3.3.3.

3.4.4 *Types*

Variables and function designators entered as Boolean primaries must be declared **Boolean** (cf. section 5.1 Type declarations and section 5.4.4 Values of function designators).

3.4.5 *The operators*

Relations take on the value **true** whenever the corresponding relation is satisfied for the expressions involved; otherwise **false**.

The meaning of the logical operators ¬ (not), ∧ (and), ∨ (or), ⊃ (implies), and ≡ (equivalent), is given by the following function table.

*b*1	**false**	**false**	**true**	**true**
*b*2	**false**	**true**	**false**	**true**
¬ *b*1	**true**	**true**	**false**	**false**
*b*1 ∧ *b*2	**false**	**false**	**false**	**true**
*b*1 ∨ *b*2	**false**	**true**	**true**	**true**
*b*1 ⊃ *b*2	**true**	**true**	**false**	**true**
*b*1 ≡ *b*2	**true**	**false**	**false**	**true**

3.4.6 *Precedence of operators*

The sequence of operations within one expression is generally from left to right, with the following additional rules:

3.4.6.1 According to the syntax given in section 3.4.1 the following rules of precedence hold:

first: arithmetic expressions according to section 3.3.5.
second: $< \leqslant = \geqslant > \neq$
third: ¬
fourth: ∧
fifth: ∨
sixth: ⊃
seventh: ≡

3.4.6.2 The use of parentheses will be interpreted in the sense given in section 3.3.5.2.

3.5 Designational expressions

3.5.1 *Syntax*

⟨label⟩ :: = ⟨identifier⟩ | ⟨unsigned integer⟩
⟨switch identifier⟩ :: = ⟨identifier⟩
⟨switch designator⟩ :: =
⟨switch identifier⟩[⟨subscript expression⟩]
⟨simple designational expression⟩ :: = ⟨label⟩ |
⟨switch designator⟩ | (⟨designational expression⟩)
⟨designational expression⟩ :: = ⟨simple designational expression⟩ | ⟨if clause⟩ ⟨simple designational expression⟩ **else** ⟨designational expression⟩

3.5.2 *Examples*

17
*p*9
Choose [*n* − 1]
Town [**if** *y* < 0 **then** *N* **else** *N* + 1]
if *Ab* < *c* **then** 17 **else** *q* [**if** *w* ⩽ 0 **then** 2 **else** *n*]

3.5.3 *Semantics*

A designational expression is a rule for obtaining a label of a statement (cf. section 4 Statements). Again, the principle of the evaluation is entirely analogous to that of arithmetic expressions (section 3.3.3). In the general case the Boolean expressions of the **if** clauses will select a simple designational expression. If this is a label the desired result is already found. A switch designator refers to the corresponding switch declaration (cf. section 5.3 Switch declarations) and by the actual numerical value of its subscript expression selects one of the designational expressions listed in the switch declaration by counting these from left to right. Since the designational expression thus selected may again be a switch designator this evaluation is obviously a recursive process.

3.5.4 *The subscript expression*

The evaluation of the subscript expression is analogous to that of subscripted variables (cf. section 3.1.4.2). The value of a switch designator is defined only if the subscript expression assumes one of the positive values 1, 2, 3, . . . , *n*, where *n* is the number of entries in the switch list.

3.5.5 *Unsigned integers as labels*

Unsigned integers used as labels have the property that leading zeroes do not affect their meaning, e.g. 00217 denotes the same label as 217.

4. Statements

The units of operation within the language are called *statements*. They will normally be executed consecutively as written. However, this sequence of operations may be broken by **go to** statements, which define their successor explicitly, and shortened by conditional statements, which may cause certain statements to be skipped.

In order to make it possible to define a specific dynamic succession, statements may be provided with labels.

Since sequences of statements may be grouped together into compound statements and blocks, the definition of statement must necessarily be recursive. Also since declarations, described in section 5, enter fundamentally into the syntactic structure, the syntactic definition of statements must suppose declarations to be already defined.

4.1 Compound statements and blocks

4.1.1 *Syntax*

⟨unlabelled basic statement⟩ :: =
⟨assignment statement⟩ | ⟨go to statement⟩ |
⟨dummy statement⟩ | ⟨procedure statement⟩
⟨basic statement⟩ :: = ⟨unlabelled basic statement⟩ |
⟨label⟩ : ⟨basic statement⟩
⟨unconditional statement⟩ :: = ⟨basic statement⟩ |
⟨compound statement⟩ | ⟨block⟩
⟨statement⟩ :: = ⟨unconditional statement⟩ |
⟨conditional statement⟩ | ⟨for statement⟩
⟨compound tail⟩ :: = ⟨statement⟩ **end** |
⟨statement⟩ ; ⟨compound tail⟩

⟨block head⟩ :: = **begin** ⟨declaration⟩ |
⟨block head⟩ ; ⟨declaration⟩
⟨unlabelled compound⟩ :: = **begin** ⟨compound tail⟩
⟨unlabelled block⟩ :: =
⟨block head⟩ ; ⟨compound tail⟩
⟨compound statement⟩ :: = ⟨unlabelled compound⟩ |
⟨label⟩ : ⟨compound statement⟩
⟨block⟩ :: = ⟨unlabelled block⟩ | ⟨label⟩ : ⟨block⟩
⟨program⟩ :: = ⟨block⟩ | ⟨compound statement⟩

This syntax may be illustrated as follows: Denoting arbitrary statements, declarations, and labels, by letters S, D, and L, respectively, the basic syntactic units take the forms:

Compound statement:
L: L: . . . **begin** S ; S ; . . . S ; S **end**

Block:
L: L: . . . **begin** D ; D ; . . . D ; S ; S ; . . . S ; S **end**

It should be kept in mind that each of the statements S may again be a complete compound statement or block.

4.1.2 *Examples*

Basic statements:
$a := p + q$
go to *Naples*
START: *CONTINUE*: $W := 7.993$

Compound statement:
begin $x := 0$; **for** $y := 1$ **step** 1 **until** n **do** $x := x +$
$A[y]$; **if** $x > q$ **then go to** *STOP* **else if** $x > w - 2$
then go to S ;
Aw: St: $W := x +$ *bob* **end**

Block:
Q: **begin integer** i, k ; **real** w ;
for $i := 1$ **step** 1 **until** m **do**
for $k := i + 1$ **step** 1 **until** m **do**
begin $w := A[i, k]$; $A[i, k] := A[k, i]$;
$A[k, i] := w$
end *for i and k*
end *block Q*

4.1.3 *Semantics*

Every block automatically introduces a new level of nomenclature. This is realized as follows: Any identifier occurring within the block may through a suitable declaration (cf. section 5 Declarations) be specified to be local to the block in question. This means (*a*) that the entity represented by this identifier inside the block has no existence outside it, and (*b*) that any entity represented by this identifier outside the block is completely inaccessible inside the block.

Identifiers (except those representing labels) occurring within a block and not being declared to this block will be non-local to it, i.e. will represent the same entity inside the block and in the level immediately outside it. A label separated by a colon from a statement, i.e. labelling that statement, behaves as though declared in the head of the smallest embracing block, i.e. the smallest block whose brackets **begin** and **end** enclose that statement. In this context a procedure body must be considered as if it were enclosed by **begin** and **end** and treated as a block.

Since a statement of a block may again itself be a block the concepts local and non-local to a block must be understood recursively. Thus an identifier, which is non-local to a block A, may or may not be non-local to the block B in which A is one statement.

4.2 Assignment statements

4.2.1 *Syntax*

⟨left part⟩ :: = ⟨variable⟩ := |
⟨procedure identifier⟩ :=
⟨left part list⟩ :: = ⟨left part⟩ |
⟨left part list⟩ ⟨left part⟩
⟨assignment statement⟩ :: = ⟨left part list⟩ ⟨arithmetic expression⟩ | ⟨left part list⟩ ⟨Boolean expression⟩

4.2.2 *Examples*

$s := p[0] := n := n + 1 + s$
$n := n + 1$
$A := B/C - v - q \times S$
$S[v, k + 2] := 3 - arctan(s \times zeta)$
$V := Q > Y \wedge Z$

4.2.3 *Semantics*

Assignment statements serve for assigning the value of an expression to one or several variables or procedure identifiers. Assignment to a procedure identifier may only occur within the body of a procedure defining the value of a function designator (cf. section 5.4.4). The process will in the general case be understood to take place in three steps as follows:

4.2.3.1 Any subscript expressions occurring in the left part variables are evaluated in sequence from left to right.

4.2.3.2 The expression of the statement is evaluated.

4.2.3.3 The value of the expression is assigned to all the left part variables, with any subscript expressions having values as evaluated in step 4.2.3.1.

4.2.4 *Types*

The type associated with all variables and procedure identifiers of a left part list must be the same. If this type is **Boolean,** the expression must likewise be **Boolean.** If the type is **real** or **integer,** the expression must be arithmetic. If the type of the arithmetic expression differs from that associated with the variables and procedure identifiers, appropriate transfer functions are understood to be automatically invoked. For transfer from **real** to **integer** type the transfer function is understood to yield a result equivalent to

$$entier\ (E + 0.5)$$

where E is the value of the expression. The type asso-

ciated with a procedure identifier is given by the declarator which appears as the first symbol of the corresponding procedure declaration (cf. section 5.4.4).

4.3 Go to statements

4.3.1 *Syntax*

⟨go to statement⟩ :: = **go to** ⟨designational expression⟩

4.3.2 *Examples*

go to 8
go to *exit* [$n + 1$]
go to *Town* [**if** $y < 0$ **then** N **else** $N + 1$]
go to if $Ab < c$ **then** 17 **else** q [**if** $w < 0$ **then** 2 **else** n]

4.3.3 *Semantics*

A **go to** statement interrupts the normal sequence of operations, defined by the write-up of statements, by defining its successor explicitly by the value of a designational expression. Thus the next statement to be executed will be the one having this value as its label.

4.3.4 *Restriction*

Since labels are inherently local, no **go to** statement can lead from outside into a block. A **go to** statement may, however, lead from outside into a compound statement.

4.3.5 **Go to** *an undefined switch designator*

A **go to** statement is equivalent to a dummy statement if the designational expression is a switch designator whose value is undefined.

4.4 Dummy statements

4.4.1 *Syntax*

⟨dummy statement⟩ :: = ⟨empty⟩

4.4.2 *Examples*

L:
begin . . . ; *John*: **end**

4.4.3 *Semantics*

A dummy statement executes no operation. It may serve to place a label.

4.5 Conditional statements

4.5.1 *Syntax*

⟨if clause⟩ :: = **if** ⟨Boolean expression⟩ **then**
⟨unconditional statement⟩ :: = ⟨basic statement⟩ |
⟨compound statement⟩ | ⟨block⟩
⟨if statement⟩ :: =
⟨if clause⟩ ⟨unconditional statement⟩
⟨conditional statement⟩ :: = ⟨if statement⟩ |
⟨if statement⟩ **else** ⟨statement⟩ |
⟨if clause⟩ ⟨for statement⟩ |
⟨label⟩ : ⟨conditional statement⟩

4.5.2 *Examples*

if $x > 0$ **then** $n := n + 1$
if $v > u$ **then** V: $q := n + m$ **else go to** R
if $s < 0 \vee P < Q$ **then** AA: **begin if** $q < v$ **then** $a := v/s$
else $y := 2 \times a$ **end**
else if $v > s$ **then** $a := v - q$
else if $v > s - 1$ **then go to** S

4.5.3 *Semantics*

Conditional statements cause certain statements to be executed or skipped depending on the running values of specified Boolean expressions.

4.5.3.1 **If** *statement.* The unconditional statement of an **if** statement will be executed if the Boolean expression of the **if** clause is true. Otherwise it will be skipped and the operation will be continued with the next statement.

4.5.3.2 *Conditional statement.* According to the syntax two different forms of conditional statements are possible. These may be illustrated as follows:

if B1 **then** S1 **else if** B2 **then** S2 **else** S3; S4

and

if B1 **then** S1 **else if** B2 **then** S2 **else if** B3 **then** S3 ; S4

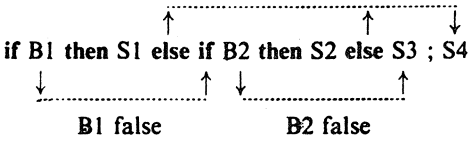
Here B1 to B3 are Boolean expressions, while S1 to S3 are unconditional statements. S4 is the statement following the complete conditional statement.

The execution of a conditional statement may be described as follows: The Boolean expressions of the **if** clauses are evaluated one after the other in sequence from left to right until one yielding the value **true** is found. Then the unconditional statement following this Boolean is executed. Unless this statement defines its successor explicitly the next statement to be executed will be S4, the statement following the complete conditional statement. Thus the effect of the delimiter **else** may be described by saying that it defines the successor of the statement it follows to be the statement following the complete conditional statement.

The construction
else ⟨unconditional statement⟩
is equivalent to
else if true then ⟨unconditional statement⟩

If none of the Boolean expressions of the **if** clauses is true, the effect of the whole conditional statement will be equivalent to that of a dummy statement.

For further explanation the following picture may be useful:



4.5.4 **Go to** *into a conditional statement*

The effect of a **go to** statement leading into a conditional statement follows directly from the above explanation of the effect of **else**.

4.6 For statements

4.6.1 *Syntax*

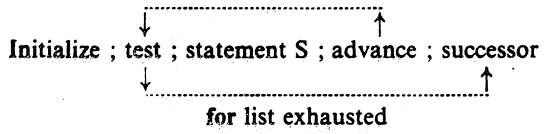
⟨for list element⟩ :: = ⟨arithmetic expression⟩ |
⟨arithmetic expression⟩ **step** ⟨arithmetic expression⟩
until ⟨arithmetic expression⟩ |
⟨arithmetic expression⟩ **while** ⟨Boolean expression⟩
⟨for list⟩ :: = ⟨for list element⟩ |
⟨for list⟩ , ⟨for list element⟩
⟨for clause⟩ :: = **for** ⟨variable⟩ := ⟨for list⟩ **do**
⟨for statement⟩ :: = ⟨for clause⟩ ⟨statement⟩ |
⟨label⟩ : ⟨for statement⟩

4.6.2 *Examples*

for q := 1 **step** s **until** n **do** $A[q] := B[q]$
for k := 1, $V1 \times 2$ **while** $V1 < N$ **do**
for $j := I + G, L,$ 1 **step** 1 **until** $N, C + D$ **do**
$A[k, j] := B[k, j]$

4.6.3 *Semantics*

A **for** clause causes the statement S which it precedes to be repeatedly executed zero or more times. In addition it performs a sequence of assignments to its controlled variable. The process may be visualized by means of the following picture:



In this picture the word initialize means: perform the first assignment of the **for** clause. Advance means: perform the next assignment of the **for** clause. Test determines if the last assignment has been done. If so the execution continues with the successor of the **for** statement. If not the statement following the **for** clause is executed.

4.6.4 *The* **for** *list elements*

The **for** list gives a rule for obtaining the values which are consecutively assigned to the controlled variable. This sequence of values is obtained from the **for** list elements by taking these one by one in the order in which they are written. The sequence of values generated by each of the three species of **for** list elements and the corresponding execution of the statement S are given by the following rules:

4.6.4.1 *Arithmetic expression.* This element gives rise to one value, namely the value of the given arithmetic expression as calculated immediately before the corresponding execution of the statement S.

4.6.4.2 **Step-until-***element.* An element of the form *A* **step** *B* **until** *C*, where *A*, *B*, and *C* are arithmetic expressions, gives rise to an execution which may be described most concisely in terms of additional ALGOL statements as follows:

$V := A$;
*L*1 : **if** $(V - C) \times sign(B) > 0$ **then go to** *Element exhausted*;
Statement S;
$V := V + B$;
go to *L*1;

where V is the controlled variable of the **for** clause and *Element exhausted* points to the evaluation according to the next element in the **for** list, or if the **step-until**-element is the last of the list, to the next statement in the program.

4.6.4.3 **While-***element.* The execution governed by a **for** list element of the form *E* **while** *F*, where *E* is an arithmetic and *F* a Boolean expression, is most concisely described in terms of additional ALGOL statements as follows:

*L*3 : $V := E$;
if ¬ *F* **then go to** *Element exhausted*;
Statement S;
go to *L*3;

where the notation is the same as in 4.6.4.2 above.

4.6.5 *The value of the controlled variable upon exit*

Upon exit out of the statement S (supposed to be compound) through a **go to** statement the value of the controlled variable will be the same as it was immediately preceding the execution of the **go to** statement.

If the exit is due to exhaustion of the **for** list, on the other hand, the value of the controlled variable is undefined after the exit.

4.6.6 **Go to** *leading into a* **for** *statement*

The effect of a **go to** statement, outside a **for** statement, which refers to a label within the **for** statement, is undefined.

4.7 Procedure statements

4.7.1 *Syntax*

⟨actual parameter⟩ :: = ⟨string⟩ | ⟨expression⟩ |
⟨array identifier⟩ | ⟨switch identifier⟩ |
⟨procedure identifier⟩
⟨letter string⟩ :: = ⟨letter⟩ | ⟨letter string⟩ ⟨letter⟩
⟨parameter delimiter⟩ :: = , |) ⟨letter string⟩ : (
⟨actual parameter list⟩ :: = ⟨actual parameter⟩ |
⟨actual parameter list⟩ ⟨parameter delimiter⟩
⟨actual parameter⟩
⟨actual parameter part⟩ :: = ⟨empty⟩ |
(⟨actual parameter list⟩)
⟨procedure statement⟩ :: =
⟨procedure identifier⟩ ⟨actual parameter part⟩

4.7.2 *Examples*

Spur (*A*) *Order*: (7) *Result to*: (*V*)
Transpose (*W*, *v* + 1)
Absmax (*A*, *N*, *M*, *Yy*, *I*, *K*)
Innerproduct (*A*[*t*, *P*, *u*], *B*[*P*], 10, *P*, *Y*)

These examples correspond to examples given in section 5.4.2.

4.7.3 *Semantics*

A procedure statement serves to invoke (call for) the execution of a procedure body (cf. section 5.4 Procedure declarations). Where the procedure body is a statement written in ALGOL the effect of this execution will be equivalent to the effect of performing the following operations on the program at the time of execution of the procedure statement:

4.7.3.1. *Value assignment* (*call by value*). All formal parameters quoted in the value part of the procedure declaration heading are assigned the values (cf. section 2.8 Values and types) of the corresponding actual parameters, these assignments being considered as being performed explicitly before entering the procedure body. The effect is as though an additional block embracing the procedure body were created in which these assignments were made to variables local to this fictitious block with types as given in the corresponding specifications (cf. section 5.4.5). As a consequence, variables called by value are to be considered as non-local to the body of the procedure, but local to the fictitious block (cf. section 5.4.3).

4.7.3.2 *Name replacement* (*call by name*). Any formal parameter not quoted in the value list is replaced, throughout the procedure body, by the corresponding actual parameter, after enclosing this latter in parentheses wherever syntactically possible. Possible conflicts between identifiers inserted through this process and other identifiers already present within the procedure body will be avoided by suitable systematic changes of the formal or local identifiers involved.

4.7.3.3 *Body replacement and execution.* Finally the procedure body, modified as above, is inserted in place of the procedure statement, and executed. If the procedure is called from a place outside the scope of any non-local quantity of the procedure body, the conflicts between the identifiers inserted through this process of body replacement and the identifiers whose declarations are valid at the place of the procedure statement or function designator will be avoided through suitable systematic changes of the latter identifiers.

4.7.4 *Actual-formal correspondence*

The correspondence between the actual parameters of the procedure statement and the formal parameters of the procedure heading is established as follows: The actual parameter list of the procedure statement must have the same number of entries as the formal parameter list of the procedure declaration heading. The correspondence is obtained by taking the entries of these two lists in the same order.

4.7.5 *Restrictions*

For a procedure statement to be defined it is evidently necessary that the operations on the procedure body defined in sections 4.7.3.1 and 4.7.3.2 lead to a correct ALGOL statement.

This poses the restriction on any procedure statement that the kind and type of each actual parameter be compatible with the kind and type of the corresponding formal parameter. Some important particular cases of this general rule are the following:

4.7.5.1 If a string is supplied as an actual parameter in a procedure statement or function designator, whose defining procedure body is an ALGOL 60 statement (as opposed to non-ALGOL code, cf. section 4.7.8), then this string can only be used within the procedure body as an actual parameter in further procedure calls. Ultimately it can only be used by a procedure body expressed in non-ALGOL code.

4.7.5.2 A formal parameter which occurs as a left part variable in an assignment statement within the procedure body and which is not called by value can only correspond to an actual parameter which is a variable (special case of expression).

4.7.5.3 A formal parameter which is used within the procedure body as an array identifier can only correspond to an actual parameter which is an array identifier of an array of the same dimensions. In addition, if the formal parameter is called by value, the local array created during the call will have the same subscript bounds as the actual array.

4.7.5.4 A formal parameter which is called by value cannot in general correspond to a switch identifier or a procedure identifier or a string, because these latter do not possess values (the exception is the procedure identifier of a procedure declaration which has an empty formal parameter part (cf. section 5.4.1) and which defines the value of a function designator (cf. section 5.4.4). This procedure identifier is in itself a complete expression).

4.7.5.5 Any formal parameter may have restrictions on the type of the corresponding actual parameter associated with it (these restrictions may, or may not, be given through specifications in the procedure heading). In the procedure statement such restrictions must evidently be observed.

4.7.6 Deleted.

4.7.7 *Parameter delimiters*

All parameter delimiters are understood to be equivalent. No correspondence between the parameter delimiters used in a procedure statement and those used in the procedure heading is expected beyond their number being the same. Thus the information conveyed by using the elaborate ones is entirely optional.

4.7.8 *Procedure body expressed in code*

The restrictions imposed on a procedure statement calling a procedure having its body expressed in non-ALGOL code evidently can only be derived from the characteristics of the code used and the intent of the user, and thus fall outside the scope of the reference language.

5. Declarations

Declarations serve to define certain properties of the quantities used in the program, and to associate them with identifiers. A declaration of an identifier is valid for one block. Outside this block the particular identifier may be used for other purposes (cf. section 4.1.3).

Dynamically this implies the following: at the time of an entry into a block (through the **begin**, since the labels inside are local and therefore inaccessible from outside) all identifiers declared for the block assume the significance implied by the nature of the declarations given. If these identifiers had already been defined by other declarations outside they are for the time being given a new significance. Identifiers which are not declared for the block, on the other hand, retain their old meaning.

At the time of an exit from a block (through **end**, or by a **go to** statement) all identifiers which are declared for the block lose their local significance.

A declaration may be marked with the additional declarator **own**. This has the following effect: upon a re-entry into the block, the values of **own** quantities will be unchanged from their values at the last exit, while the values of declared variables which are not marked as **own** are undefined. Apart from labels and formal parameters of procedure declarations and with the possible exception of those for standard functions (cf. sections 3.2.4 and 3.2.5) all identifiers of a program must be declared. No identifier may be declared more than once in any one block head.

Syntax

⟨declaration⟩ :: = ⟨type declaration⟩ |
⟨array declaration⟩ |
⟨switch declaration⟩ |
⟨procedure declaration⟩

5.1 Type declarations

5.1.1 *Syntax*

⟨type list⟩ :: = ⟨simple variable⟩ |
⟨simple variable⟩, ⟨type list⟩
⟨type⟩ :: = **real** | **integer** | **Boolean**
⟨local or own type⟩ :: = ⟨type⟩ | **own** ⟨type⟩
⟨type declaration⟩ :: = ⟨local or own type⟩ ⟨type list⟩

5.1.2 *Examples*

integer *p, q, s*
own Boolean *Acryl, n*

5.1.3 *Semantics*

Type declarations serve to declare certain identifiers to represent simple variables of a given type. **Real** declared variables may only assume positive or negative values including zero. **Integer** declared variables may only assume positive and negative integral values, including zero. **Boolean** declared variables may only assume the values **true** and **false.**

In arithmetic expressions any position which can be occupied by a real declared variable may be occupied by an integer declared variable.

For the semantics of **own,** see the fourth paragraph of section 5 above.

5.2 Array declarations

5.2.1 *Syntax*

⟨lower bound⟩ :: = ⟨arithmetic expression⟩
⟨upper bound⟩ :: = ⟨arithmetic expression⟩
⟨bound pair⟩ :: = ⟨lower bound⟩ : ⟨upper bound⟩
⟨bound pair list⟩ :: = ⟨bound pair⟩ |
⟨bound pair list⟩ , ⟨bound pair⟩
⟨ array segment ⟩:: =
⟨array identifier⟩ [⟨bound pair list⟩] |
⟨array identifier⟩, ⟨array segment⟩
⟨array list⟩ :: = ⟨array segment⟩ | ⟨array list⟩ , ⟨array segment⟩
⟨array declaration⟩ :: = **array** ⟨array list⟩ |
⟨local or own type⟩ **array** ⟨array list⟩

5.2.2 *Examples*

array *a, b, c*[7 : *n*, 2 : *m*], *s* [−2 : 10]
own integer array *A*[**if** *c* < 0 **then** 2 **else** 1 : 20]
real array *q*[−7 : −1]

5.2.3 *Semantics*

An array declaration declares one or several identifiers to represent multidimensional arrays of subscripted variables and gives the dimensions of the arrays, the bounds of the subscripts and the types of the variables.

5.2.3.1 *Subscript bounds.* The subscript bounds for any array are given in the first subscript bracket following the identifier of this array in the form of a bound pair list. Each item of this list gives the lower and upper bound of a subscript in the form of two arithmetic expressions separated by the delimiter : . The bound pair list gives the bounds of all subscripts taken in order from left to right.

5.2.3.2 *Dimensions.* The dimensions are given as the number of entries in the bound pair lists.

5.2.3.3 *Types.* All arrays declared in one declaration are of the same quoted type. If no type declarator is given the type **real** is understood.

5.2.4 *Lower upper bound expressions*

5.2.4.1 The expressions will be evaluated in the same way as subscript expressions (cf. section 3.1.4.2).

5.2.4.2 The expressions can only depend on variables and procedures which are non-local to the block for which the array declaration is valid. Consequently in the outermost block of a program only array declarations with constant bounds may be declared.

5.2.4.3 An array is defined only when the values of all upper subscript bounds are not smaller than those of the corresponding lower bounds.

5.2.4.4 The expressions will be evaluated once at each entrance into the block.

5.2.5 *The identity of subscripted variables*

The identity of a subscripted variable is not related to the subscript bounds given in the array declaration. However, even if an array is declared **own** the values of the corresponding subscripted variables will, at any time, be defined only for those of these variables which have subscripts within the most recently calculated subscript bounds.

5.3 Switch declarations

5.3.1 *Syntax*

⟨switch list⟩ :: = ⟨designational expression⟩ |
⟨switch list⟩, ⟨designational expression⟩
⟨switch declaration⟩ :: =
switch ⟨switch identifier⟩ := ⟨switch list⟩

5.3.2 *Examples*

switch *S* := *S*1, *S*2, *Q*[*m*], **if** *v* > − 5 **then** *S*3 **else** *S*4
switch *Q* := *p*1, *w*

5.3.3 *Semantics*

A switch declaration defines the set of values of the corresponding switch designators. These values are given one by one as the values of the designational expressions entered in the switch list. With each of these designational expressions there is associated a positive integer, 1, 2, . . . , obtained by counting the items in the list from left to right. The value of the switch designator corresponding to a given value of the subscript expression (cf. section 3.5 Designational expressions) is the value of the designational expression in the switch list having this given value as its associated integer.

5.3.4 *Evaluation of expressions in the switch list*

An expression in the switch list will be evaluated every time the item of the list in which the expression occurs is referred to, using the current values of all variables involved.

5.3.5 *Influence of scopes*

If a switch designator occurs outside the scope of a quantity entering into a designational expression in the switch list, and an evaluation of this switch designator selects this designational expression, then the conflicts between the identifiers for the quantities in this expression and the identifiers whose declarations are valid at the place of the switch designator will be avoided through suitable systematic changes of the latter identifiers.

5.4 Procedure declarations

5.4.1 *Syntax*

⟨formal parameter⟩ :: = ⟨identifier⟩
⟨formal parameter list⟩ :: = ⟨formal parameter⟩ |
⟨formal parameter list⟩ ⟨parameter delimiter⟩
⟨formal parameter⟩
⟨formal parameter part⟩ :: = ⟨empty⟩ |
(⟨formal parameter list⟩)
⟨identifier list⟩ :: = ⟨identifier⟩ |
⟨identifier list⟩ , ⟨identifier⟩
⟨value part⟩ :: = **value** ⟨identifier list⟩ ; | ⟨empty⟩
⟨specifier⟩ :: = **string** | ⟨type⟩ | **array** | ⟨type⟩ **array** |
label | **switch** | **procedure** | ⟨type⟩ **procedure**
⟨specification part⟩ :: = ⟨empty⟩ |
⟨specifier⟩ ⟨identifier list⟩ ; |
⟨specification part⟩ ⟨specifier⟩ ⟨identifier list⟩ ;
⟨procedure heading⟩ :: = ⟨procedure identifier⟩
⟨formal parameter part⟩;
⟨value part⟩ ⟨specification part⟩
⟨procedure body⟩ :: = ⟨statement⟩ | ⟨code⟩
⟨procedure declaration⟩ :: =
procedure ⟨procedure heading⟩ ⟨procedure body⟩ |
⟨type⟩ **procedure** ⟨procedure heading⟩ ⟨procedure body⟩

5.4.2 *Examples* (see also the examples at the end of the report)

```
procedure Spur (a) Order: (n) Result: (s) ; value n ;
array a ; integer n ; real s ;
begin integer k ;
s := 0 ;
for k := 1 step 1 until n do s := s + a[k, k]
end

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

integer procedure Step(u) ; real u ;
Step := if 0 ⩽ u ∧ u ⩽ 1 then 1 else 0
```

```
procedure Absmax (a) size: (n, m) Result: (y) Subscripts:
    (i, k) ;
comment The absolute greatest element of the matrix a, of
    size n by m is transferred to y, and the subscripts of this
    element to i and k ;
array a ; integer n, m, i, k ; real y;
begin integer p, q ;
y := 0 ;
for p := 1 step 1 until n do for q := 1 step 1 until m do
if abs(a[p, q]) > y then begin y := abs(a[p, q]) ;
    i := p; k := q end end Absmax
```

```
procedure Innerproduct (a, b) Order: (k, p) Result : (y) ;
    value k ;
integer k, p ; real y, a, b ;
begin real s ; s := 0 ;
for p := 1 step 1 until k do s := s + a × b ;
y := s
end Innerproduct
```

5.4.3 *Semantics*

A procedure declaration serves to define the procedure associated with a procedure identifier. The principal constituent of a procedure declaration is a statement or a piece of code, the procedure body, which through the use of procedure statements and/or function designators may be activated from other parts of the block in the head of which the procedure declaration appears. Associated with the body is a heading, which specifies certain identifiers occurring within the body to represent formal parameters. Formal parameters in the procedure body will, whenever the procedure is activated (cf. section 3.2 Function designators and section 4.7 Procedure statements) be assigned the values of or replaced by actual parameters. Identifiers in the procedure body which are not formal will be either local or non-local to the body depending on whether they are declared within the body or not. Those of them which are non-local to the body may well be local to the block in the head of which the procedure declaration appears. The procedure body always acts like a block, whether it has the form of one or not. Consequently the scope of any label labelling a statement within the body or the body itself can never extend beyond the procedure body. In addition, if the identifier of a formal parameter is declared anew within the procedure body (including the case of its use as a label as in section 4.1.3), it is thereby given a local significance and actual parameters which correspond to it are inaccessible throughout the scope of this inner local quantity.

5.4.4 *Values of function designators*

For a procedure declaration to define the value of a function designator there must, within the procedure body, occur one or more explicit assignment statements with the procedure identifier in a left part; at least one of these must be executed, and the type associated with the procedure identifier must be declared through the appearance of a type declarator as the very first symbol of the procedure declaration. The last value so assigned is used to continue the evaluation of the expression in which the function designator occurs. Any occurrence of the procedure identifier within the body of the procedure other than in a left part in an assignment statement denotes activation of the procedure.

5.4.5 *Specifications*

In the heading a specification part, giving information about the kinds and types of the formal parameters by means of an obvious notation, may be included. In this part no formal parameter may occur more than once. Specifications of formal parameters called by value (cf. section 4.7.3.1) must be supplied and specifications of formal parameters called by name (cf. section 4.7.3.2) may be omitted.

5.4.6 *Code as procedure body*

It is understood that the procedure body may be expressed in non-ALGOL language. Since it is intended that the use of this feature should be entirely a question of hardware representation, no further rules concerning this code language can be given within the reference language.

Examples of procedure declarations

Example 1

```
procedure euler (fct, sum, eps, tim) ; value eps, tim ;
    integer tim ;
real procedure fct ; real sum, eps ;
comment euler computes the sum of fct(i) for i from zero
    up to infinity by means of a suitably refined euler trans-
    formation. The summation is stopped as soon as tim
    times in succession the absolute value of the terms of the
    transformed series are found to be less than eps. Hence,
    one should provide a function fct with one integer argument,
    an upper bound eps, and an integer tim. The output is the
    sum sum. euler is particularly efficient in the case of a
    slowly convergent or divergent alternating series ;
begin integer i, k, n, t ; array m[0 : 15] ; real mn, mp, ds ;
    i := n := t := 0 ; m[0] := fct(0) ; sum := m[0]/2 ;
nextterm: i := i + 1 ; mn := fct(i) ;
          for k := 0 step 1 until n do
            begin mp := (mn + m[k])/2; m[k] := mn ;
              mn := mp end means ;
          if (abs(mn) < abs(m[n])) ∧ (n < 15) then
            begin ds := mn/2 ; n := n + 1 ;
              m[n] := mn end accept
          else ds := mn ;
          sum := sum + ds ;
          if abs(ds) < eps then t := t + 1 else t := 0 ;
          if t < tim then go to nextterm
end euler
```

Example 2*

```
procedure RK(x, y, n, FKT, eps, eta, xE, yE, fi) ;
  value x, y ; integer n ;
Boolean fi ; real x, eps, eta, xE ; array y, yE ;
  procedure FKT ;
comment : RK integrates the system
```

$$y'_k = f_k(x, y_1, y_2, \ldots, y_n) \quad (k = 1, 2, \ldots n)$$

of differential equations with the method of Runge-Kutta with automatic search for appropriate length of integration step. Parameters are: The initial values x and y[k] for x and the unknown functions $y_k(x)$. The order n of the system. The procedure FKT(x, y, n, z) which represents the system to be integrated, i.e. the set of functions f_k. The tolerance values eps and eta which govern the accuracy of the numerical integration. The end of the integration interval xE. The output parameter yE which represents the solution at x = xE. The Boolean variable fi, which must always be given the value **true** *for an isolated or first entry into RK. If, however, the functions y must be available at several meshpoints $x_0, x_1, \ldots, x_n$, then the procedure must be called repeatedly (with $x := x_k$, $xE = x_{k+1}$, for $k = 0, 1, \ldots, n-1$) and then the later calls may occur with fi =* **false** *which saves computing time. The input parameters of FKT must be x, y, n, the output parameter z represents the set of derivatives $z[k] = f_k(x, y[1], y[2], \ldots, y[n])$ for x and the actual y's. A procedure comp enters as a non-local identifier ;*

```
begin
  array z, y1, y2, y3[1 : n] ; real x1, x2, x3, H ;
    Boolean out ;
  integer k, j ; own real s, Hs ;
  procedure RK1ST(x, y, h, xe, ye) ; real x, h, xe ;
      array y, ye ;
    comment : RK1ST integrates one single RUNGE-
    KUTTA step with initial values x, y[k] which yields
    the output parameters xe = x + h and ye[k], the
    latter being the solution at xe. IMPORTANT: the
    parameters n, FKT, z enter RK1ST as non-local
    entities ;
    begin
      array w[1 : n], a[1 : 5] ; integer k, j ;
      a[1] := a[2] := a[5] := h/2 ; a[3]:= a[4] := h;
      xe := x ;
      for k := 1 step 1 until n do ye[k] := w[k] := y[k];
      for j := 1 step 1 until 4 do
      begin
        FKT(xe, w, n, z) ;
        xe := x + a[j] ;
        for k := 1 step 1 until n do
        begin
          w[k] := y[k] + a[j] × z[k] ;
          ye[k] := ye[k] + a[j + 1] × z[k]/3
        end k
      end j
    end RK1ST ;
BEGIN OF PROGRAM:
  if fi then begin H := xE − x; s := 0 end
  else H := Hs ; out := false ;
AA: if (x + 2.01 × H − xE > 0) ≡ (H > 0) then
      begin Hs := H ; out := true ; H := (xE − x)/2
      end if ;
      RK1ST(x, y, 2 × H, x1, y1) ;
BB: RK1ST(x, y, H, x2, y2) ; RK1ST(x2, y2, H, x3, y3);
      for k := 1 step 1 until n do
        if comp(y1[k], y3[k], eta) > eps then go to CC ;
      comment : comp(a, b, c) is a function designator, the
      value of which is the absolute value of the difference
      of the mantissae of a and b, after the exponents of
      these quantities have been made equal to the largest
      of the exponents of the originally given parameters
      a, b, c ;
      x := x3 ; if out then go to DD ;
      for k := 1 step 1 until n do y[k] := y3[k] ;
      if s = 5 then begin s := 0 ; H := 2 × H end if ;
      s := s + 1 ; go to AA ;
CC: H := 0.5 × H ; out := false ; x1 := x2 ;
      for k := 1 step 1 until n do y1[k] := y2[k] ;
      go to BB ;
DD: for k := 1 step 1 until n do yE[k] := y3[k]
end RK .
```

* This RK-program contains some new ideas which are related to ideas of S. Gill, "A process for the step by step integration of differential equations in an automatic computing machine," *Proc. Camb. Phil. Soc.*, Vol. 47 (1951), p. 96, and C. E Fröberg, "On the solution of ordinary differential equations with digital computing machines," *Fysiograf Sällsk. Lund*, Förhd 20, Nr. 11 (1950), pp. 136–52. It must be clear, however, that with respect to computing time and round-off errors it may not be optimal, nor has it actually been tested on a computer.

For alphabetic index, see next page

Alphabetic index of definitions of concepts and syntactic units

All references are given through section numbers. The references are given in three groups:

def Following the abbreviation "def" reference to the syntactic definition (if any) is given.

synt Following the abbreviation "synt" references to the occurrences in metalinguistic formulae are given. References already quoted in the def-group are not repeated.

text Following the word "text" the references to definitions given in the text are given.

The basic symbols represented by signs other than underlined* words have been collected at the beginning. The examples have been ignored in compiling the index.

* Bold faced.—*Ed.*

+, see: plus
−, see: minus
×, see: multiply
/, ÷, see: divide
↑, see: exponentiation
<, ≤, =, ≥, >, ≠, see: ⟨relational operator⟩
≡, ⊃, ∨, ∧, ¬, see: ⟨logical operator⟩
„, see: comma
., see: decimal point
$_{10}$, see: ten
:, see: colon
;, see: semicolon
:=, see: colon equal
␣, see: space
( ), see: parentheses
[ ], see: subscript bracket
‘ ’, see: string quote
⟨actual parameter⟩, def 3.2.1, 4.7.1
⟨actual parameter list⟩, def 3.2.1, 4.7.1
⟨actual parameter part⟩, def 3.2.1, 4.7.1
⟨adding operator⟩, def 3.3.1
alphabet, text 2.1
arithmetic, text 3.3.6
⟨arithmetic expression⟩, def 3.3.1 synt 3, 3.1.1, 3.3.1, 3.4.1, 4.2.1, 4.6.1, 5.2.1 text 3.3.3
⟨arithmetic operator⟩, def 2.3 text 3.3.4
array, synt 2.3, 5.2.1, 5.4.1
array, text 3.1.4.1
⟨array declaration⟩, def 5.2.1 synt 5 text 5.2.3
⟨array identifier⟩, def 3.1.1 synt 3.2.1, 4.7.1, 5.2.1 text 2.8
⟨array list⟩, def 5.2.1
⟨array segment⟩, def 5.2.1
⟨assignment statement⟩, def 4.2.1 synt 4.1.1 text 1, 4.2.3
⟨basic statement⟩, def 4.1.1 synt 4.5.1
⟨basic symbol⟩, def 2
begin, synt 2.3, 4.1.1
⟨block⟩, def 4.1.1 synt 4.5.1 text 1, 4.1.3, 5
⟨block head⟩, def 4.1.1
Boolean, synt 2.3, 5.1.1 text 5.1.3
⟨Boolean expression⟩, def 3.4.1 synt 3, 3.3.1, 4.2.1, 4.5.1, 4.6.1 text 3.4.3
⟨Boolean factor⟩, def 3.4.1
⟨Boolean primary⟩, def 3.4.1
⟨Boolean secondary⟩, def 3.4.1
⟨Boolean term⟩, def 3.4.1
⟨bound pair⟩, def 5.2.1
⟨bound pair list⟩, def 5.2.1
⟨bracket⟩, def 2.3
⟨code⟩, synt 5.4.1 text 4.7.8, 5.4.6
colon :, synt 2.3, 3.2.1, 4.1.1, 4.5.1, 4.6.1, 4.7.1, 5.2.1
colon equal :=, synt 2.3, 4.2.1, 4.6.1, 5.3.1
comma , , synt 2.3, 3.1.1, 3.2.1, 4.6.1, 4.7.1, 5.1.1, 5.2.1, 5.3.1, 5.4.1
comment, synt 2.3
comment convention, text 2.3
⟨compound statement⟩, def 4.1.1 synt 4.5.1 text 1
⟨compound tail⟩, def 4.1.1
⟨conditional statement⟩, def 4.5.1 synt 4.1.1 text 4.5.3
⟨decimal fraction⟩, def 2.5.1
⟨decimal number⟩, def 2.5.1 text 2.5.3
decimal point ., synt 2.3, 2.5.1
⟨declaration⟩, def 5 synt 4.1.1 text 1, 5 (complete section)
⟨declarator⟩, def 2.3
⟨delimiter⟩, def 2.3 synt 2
⟨designational expression⟩, def 3.5.1 synt 3, 4.3.1, 5.3.1 text 3.5.3
⟨digit⟩, def 2.2.1 synt 2, 2.4.1, 2.5.1
dimension, text 5.2.3.2
divide / ÷, synt 2.3, 3.3.1 text 3.3.4.2
do, synt 2.3, 4.6.1
⟨dummy statement⟩, def 4.4.1 synt 4.1.1 text 4.4.3
else, synt 2.3, 3.3.1, 3.4.1, 3.5.1, 4.5.1, text 4.5.3.2
⟨empty⟩, def 1.1 synt 2.6.1, 3.2.1, 4.4.1, 4.7.1, 5.4.1
end, synt 2.3, 4.1.1
entier, text 3.2.5
exponentiation ↑, synt 2.3, 3.3.1 text 3.3.4.3
⟨exponent part⟩, def 2.5.1 text 2.5.3
⟨expression⟩, def 3 synt 3.2.1, 4.7.1 text 3 (complete section)
⟨factor⟩, def 3.3.1
false, synt 2.2.2
for, synt 2.3, 4.6.1
⟨for clause⟩, def 4.6.1 text 4.6.3
⟨for list⟩, def 4.6.1 text 4.6.4
⟨for list element⟩, def 4.6.1 text 4.6.4.1, 4.6.4.2, 4.6.4.3
⟨formal parameter⟩, def 5.4.1 text 5.4.3
⟨formal parameter list⟩, def 5.4.1
⟨formal parameter part⟩, def 5.4.1
⟨for statement⟩, def 4.6.1 synt 4.1.1, 4.5.1 text 4.6 (complete section)
⟨function designator⟩, def 3.2.1 synt 3.3.1, 3.4.1 text 3.2.3, 5.4.4
go to, synt 2.3, 4.3.1
⟨go to statement⟩, def 4.3.1 synt 4.1.1 text 4.3.3
⟨identifier⟩, def 2.4.1 synt 3.1.1, 3.2.1, 3.5.1, 5.4.1 text 2.4.3
⟨identifier list⟩, def 5.4.1
if, synt 2.3, 3.3.1, 4.5.1
⟨if clause⟩, def 3.3.1, 4.5.1 synt 3.4.1, 3.5.1 text 3.3.3, 4.5.3.2
⟨if statement⟩, def 4.5.1 text 4.5.3.1
⟨implication⟩, def 3.4.1
integer, synt 2.3, 5.1.1 text 5.1.3
⟨integer⟩, def 2.5.1 text 2.5.4
label, synt 2.3, 5.4.1

⟨label⟩, def 3.5.1 synt 4.1.1, 4.5.1, 4.6.1 text 1, 4.1.3
⟨left part⟩, def 4.2.1
⟨left part list⟩, def 4.2.1
⟨letter⟩, def 2.1 synt 2, 2.4.1, 3.2.1, 4.7.1
⟨letter string⟩, def 3.2.1, 4.7.1
local, text 4.1.3
⟨local or own type⟩, def 5.1.1 synt 5.2.1
⟨logical operator⟩, def 2.3 synt 3.4.1 text 3.4.5
⟨logical value⟩, def 2.2.2 synt 2, 3.4.1
⟨lower bound⟩, def 5.2.1 text 5.2.4
non-local, text 4.1.3
minus −, synt 2.3, 2.5.1, 3.3.1 text 3.3.4.1
multiply ×, synt 2.3, 3.3.1 text 3.3.4.1
⟨multiplying operator⟩, def 3.3.1
⟨number⟩, def 2.5.1 text 2.5.3, 2.5.4
⟨open string⟩, def 2.6.1
⟨operator⟩, def 2.3
own, synt 2.3, 5.1.1 text 5, 5.2.5
⟨parameter delimiter⟩, def 3.2.1, 4.7.1 synt 5.4.1 text 4.7.7
parentheses ( ), synt 2.3, 3.2.1, 3.3.1, 3.4.1, 3.5.1, 4.7.1, 5.4.1 text 3.3.5.2
plus +, synt 2.3, 2.5.1, 3.3.1 text 3.3.4.1
⟨primary⟩, def 3.3.1
procedure, synt 2.3, 5.4.1
⟨procedure body⟩, def 5.4.1
⟨procedure declaration⟩, def 5.4.1 synt 5 text 5.4.3
⟨procedure heading⟩, def 5.4.1 text 5.4.3
⟨procedure identifier⟩, def 3.2.1 synt 3.2.1, 4.7.1, 5.4.1 text 4.7.5.4
⟨procedure statement⟩, def 4.7.1 synt 4.1.1 text 4.7.3
⟨program⟩, def 4.1.1 text 1
⟨proper string⟩, def 2.6.1
quantity, text 2.7
real, synt 2.3, 5.1.1 text 5.1.3
⟨relation⟩, def 3.4.1 text 3.4.5
⟨relational operator⟩, def 2.3, 3.4.1
scope, text 2.7
semicolon ;, synt 2.3, 4.1.1, 5.4.1
⟨separator⟩, def 2.3
⟨sequential operator⟩, def 2.3
⟨simple arithmetic expression⟩, def 3.3.1 text 3.3.3
⟨simple Boolean⟩, def 3.4.1
⟨simple designational expression⟩, def 3.5.1
⟨simple variable⟩, def 3.1.1 synt 5.1.1 text 2.4.3
space ␣, synt 2.3 text 2.3, 2.6.3
⟨specification part⟩, def 5.4.1 text 5.4.5
⟨specificator⟩, def 2.3
⟨specifier⟩, def 5.4.1
standard function, text 3.2.4, 3.2.5
⟨statement⟩, def 4.1.1, synt 4.5.1, 4.6.1, 5.4.1 text 4 (complete section)
statement bracket, see: **begin end**
step, synt 2.3, 4.6.1 text 4.6.4.2
string, synt 2.3, 5.4.1
⟨string⟩, def 2.6.1 synt 3.2.1, 4.7.1 text 2.6.3
string quotes ‘ ’, synt 2.3, 2.6.1 text 2.6.3
subscript, text 3.1.4.1
subscript bound, text 5.2.3.1
subscript brackets [ ], synt 2.3, 3.1.1, 3.5.1, 5.2.1
⟨subscript expression⟩, def 3.1.1 synt 3.5.1
⟨subscript list⟩, def 3.1.1
⟨subscripted variable⟩, def 3.1.1 text 3.1.4.1
successor, text 4
switch, synt 2.3, 5.3.1, 5.4.1
⟨switch declaration⟩, def 5.3.1 synt 5 text 5.3.3
⟨switch designator⟩, def 3.5.1 text 3.5.3
⟨switch identifier⟩, def 3.5.1 synt 3.2.1, 4.7.1, 5.3.1
⟨switch list⟩, def 5.3.1
⟨term⟩, def 3.3.1
ten $_{10}$, synt 2.3, 2.5.1
then, synt 2.3, 3.3.1, 4.5.1
transfer function, text 3.2.5
true, synt 2.2.2
⟨type⟩, def 5.1.1 synt 5.4.1 text 2.8
⟨type declaration⟩, def 5.1.1 synt 5 text 5.1.3
⟨type list⟩, def 5.1.1
⟨unconditional statement⟩, def 4.1.1, 4.5.1
⟨unlabelled basic statement⟩, def. 4.1.1
⟨unlabelled block⟩, def 4.1.1
⟨unlabelled compound⟩, def 4.1.1
⟨unsigned integer⟩, def 2.5.1, 3.5.1
⟨unsigned number⟩, def 2.5.1 synt 3.3.1
until, synt 2.3, 4.6.1 text 4.6.4.2
⟨upper bound⟩, def 5.2.1 text 5.2.4
value, synt 2.3, 5.4.1
value, text 2.8, 3.3.3
⟨value part⟩, def 5.4.1 text 4.7.3.1
⟨variable⟩, def 3.1.1 synt 3.3.1, 3.4.1, 4.2.1, 4.6.1 text 3.1.3
⟨variable identifier⟩, def 3.1.1
while, synt 2.3, 4.6.1 text 4.6.4.3

ABS Procedure 51
Actual Parameters 42
algol Statement 71
ARCTAN Procedure 51
Arithmetic Expression 9
Arithmetic Operators 9
Arrays 6
Array Declaration 27
Array Dimension 27
Assignment Statement 16

Basic Symbols 4
Binary I/O 62 ii
Blocks 25
Boolean Declarations 26
Boolean Expressions 11
Boolean Operators 12
Bounds 27

Call by Name 43
Call by Value 45
Character Codes 69
CLOSEDA Procedure 62 iv
CLOSESQ Procedure 62 ii
CLOSESTREAM Procedure 62 i
CODE Procedure 61
Comments 8
Compile Time Error Messages 75 ff
Compound Statements 23
Conditional Expression 18
Conditional Statement 17
Controlled Variable 19
COPYTEXT Procedure 65
COS Procedure 51
CPUTIME Procedure 64

Declarations 26 ff
Designational Expressions 18
Diagnostic Aids 75 ff
Direct Access Binary Files 62 iv
Dummy Statement 16

ENTIER Procedure 51
EPSILON Procedure 64
Evaluation of Expressions 10,13
Exact Actions of for Statements 22
EXP Procedure 51
Exponent part 5
Exponentiation (Rules of) 10,85
external Statement 72

for List 19
for List Element 19
for Statement 19 ff
Formal Parameters 37
Formal Procedures 40
Fortran Language Procedures 73

Functions 46

GETDA Procedure 62 v
GETSQ Procedure 62 iii
goto Statement 18

Hardware Representation 67

Identifiers 6
if...then Statement 17
if...then...else Statement 17
IMP language Procedure 72
INCHAR Procedure 63
ININTEGER Procedure 62 v
Input 55 ff
INREAL Procedure 63
Integer Declarations 26 ff
Integer Numbers 4

Jensen's Device 48

Labels 15
Labels as Parameters 40
LENGTH Procedure 63
LN Procedure 51
Logical Operators 12

MAXINT Procedure 64
MAXREAL Procedure 64
MINREAL Procedure 64
Mixed Language Programming 72
MONITOR Procedure 82
Multiple Assignments 16

Nesting Blocks 25
NEWLINE Procedure 58
NEWLINES Procedure 58
NEWPAGE Procedure 58
NEXTCH Procedure 65
NEXTSYMBOL Procedure 61
Numbers 4,5

OPENDA Procedure 62 iv
OPENSQ Procedure 62 ii
Operator Precedence 10
Optimisation 53
Order of Evaluation 10,13
OUTCHAR Procedure 63
OUTINTEGER Procedure 62 v
Output 55 ff
OUTPUT Procedure 64
OUTREAL Procedure 63
OUTSTRING Procedure 63
OUTTERMINATOR Procedure 63
own Arrays 32
own Variables 32

PAPERTHROW Procedure 65
Parameter Delimiters 38
PRINT Procedure 56
PRINTCH Procedure 65
PRINT STRING Procedure 59
PRINTSYMBOL Procedure 61
PRINT1900 Procedure 64
Procedure Body 35,36
Procedure Call 36 ff
Procedure Declaration 35
Procedure Heading 35
Procedures 35 ff
Procedures as Parameters 40
Procedures with Parameters 37
PUTDA Procedure 62 iv
PUTSQ Procedure 62 iii

READ Procedure 55
READBOOLEAN Procedure 64
READCH Procedure 65
READSYMBOL Procedure 61
READ1900 Procedure 64
Real Declarations 26 ff
Real Numbers 5
Recursion 47
Redeclaration 30
Relational Operators 11
Run Time Error Messages 81 ff
RWNDSQ Procedure 62 iii

Scope of Variables 30
Segmentation 71
SELECTINPUT Procedure 62 i
SELECTOUTPUT Procedure 62 i
Semi-colon 15
Separator 15
Sequential Access Binary Files 62 ii
SIGN Procedure 51
Simple Arithmetic Expressions 9
Simple Boolean Expressions 11
SIN Procedure 51
SKIPCH Procedure 65
SPACE Procedure 57
SPACES Procedure 57
Specifiers 39
Specification Part 39
SQRT Procedure 51
Standard Functions 51
Statements 15 ff
step-until Elements 20
STOP Procedure 64
Strings 8
Subscript 7
Subscripted Variables 7
Switch Declaration 29
Switch List 29
Switches 29
Switches as Parameters 40

Type Declaration 26

Unconditional Statement 15

Value Part 43
Variables 6,26

while Element 21
WRITEBOOLEAN Procedure 65
WRITETEXT Procedure 65

UNIVERSITY OF EDINBURGH

Edinburgh Regional Computing Centre

Edinburgh ALGOL Language Manual Update 1

Originator		Issue date
Neil Hamilton-Smith		August 1977

page 10 line 13 — replace A^3 by A^B

line 24 — should read

* / ÷

page 22 line 22 — should start

the cycle; V, A, B and C are evaluated

page 42 line 23 — should read

S + R:= ((A*A+B)+2)*(A*A+B)↑2+1.3

replace pages 59-62 — by the attached pages 59-62(v)

page 63 lines 13,14 — replace INSYMBOL by INCHAR

lines 25,26 — replace OUTSYMBOL by OUTCHAR

page 65 — procedure PAPERTHROW is now provided for programmers with working 1900 programs.

line 37 — add

For ALGOL on 2900 it is necessary to prefix READ1900 and PRINT1900 by ICL9CE

page 67 line 16 — after this line add

In quote mode quotes in strings and comments should occur in matching pairs or not at all.

page 68 line 16 — should read

or (/ ' ' or or (/

line 17 — should read

or /) ' ' or or /)

line 20 — should read

div(÷) %DIV '/' or 'DIV'

page 69 line 9 — remove the * from the triplet 96 ' *

page 76 foot — add

5. WARNING:- STRING CONSTANT TOO LONG
String constants are limited to 255 characters and will be truncated if necessary.

page 77 lines 10,14, 17,20,24,30 — add NAME after the)

page 77 line 34 — add LABEL after the)

page 78 lines 19, 22 — add NAME after the)

line 25 — add ARR after the)

line 29 — add NAME after the)

page 79 line 12,15 — add NAME after the)

line 26 — add LOCAL NAME after the)

page 80 line 11 — add NAME after the)

replace pages 109,110 — by the attached pages 109,110

N.B. procedures CLOSESTREAM, OPENSQ, CLOSESQ, PUTSQ, GETSQ, RWNDSQ, OPENDA, CLOSEDA, PUTDA, GETDA and PAPERTHROW currently have been provided for ALGOL(E) on the 2900 but have not been provided for EMAS ALGOL: a note will appear in the ERCC Technical Newsletter if and when any of these procedures is provided for EMAS ALGOL.