decsystem10

ALGOL PROGRAMMER'S REFERENCE MANUAL

digital

decsystem10

ALGOL
PROGRAMMER'S REFERENCE MANUAL

This manual reflects Version 6 of the ALGOL System.

Additional copies of this manual may be ordered from: Software Distribution Center,
Digital Equipment Corporation, Maynard, MA 01754 Order Code: DEC-10-LALMA-B-D

digital equipment corporation · maynard. massachusetts

First Printing, September 1971
Revised: September 1971
December 1971
May 1972
December 1972
July 1973
July 1974
May 1975
September 1975
Second Printing, March 1976

The postage prepaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist us in preparing future documentation.

CONTENTS

TABLES

CHAPTER 1
INTRODUCTION

1.1 GENERAL

DECsystem-10 ALGOL is an implementation of ALGOL-60; ALGOL is an abbreviation of ALGOrithmic Language, and 1960 is the year it was defined. The authoritative definition of ALGOL-60 is contained in the "Revised Report on the Algorithmic Language ALGOL-60",[1] hereafter referred to as the "Revised Report". This report leaves a number of ALGOL-60 features undefined, notably input/output, and permits the implementer of the language some latitude in interpreting other features. Many of these features have been discussed extensively since the publication of the Revised Report; some have been given rigorous interpretations in various versions of ALGOL, particularly the ALGOL-68 Language.[2]

Where there is need for interpretation in the Revised Report, such interpretations as seem reasonable have been made in light of current ALGOL opinion. Where no guidelines exist, ALGOL-68 is used as a basis. These points are discussed in Chapter 19.

1.2 DECsystem-10 ALGOL

The purpose of this manual is to teach the use of DECsystem-10 ALGOL. The manual is written both for the user who is familiar with ALGOL implementations and for the user who has no knowledge of ALGOL but is reasonably fluent in a high-level scientific programming language such as FORTRAN IV. This manual is <u>not</u> a primer in high-level languages.[3]

---

[1] "Revised Report on the Algorithmic Language ALGOL-60", Backus et al., <u>Communications of the ACM</u>, 1963, vol. 6, no. 1, pp. 1-17.

[2] "Report on the Algorithmic Language ALGOL-68", A. Van Wijngaarden (Editor), B. J. Mailloux, J. E. L. Peck, and C. H. A. Koster, Mathematisch Centrum, Amsterdam, MR101, October 1969.

[3] <u>A Primer of ALGOL-60 Programming</u>, E. W. Dijkstra, Academic Press, London, 1962.

Readers not thoroughly familiar with ALGOL should read the entire manual. Readers already familiar with ALGOL-60 should read all chapters except Chapters 5, 6, 7, 8, 9, 10, 11, 12, and 14, which need be referred to only briefly.

1.3 THE ALGOL COMPILER

The DECsystem-10 ALGOL Compiler is that part of the DECsystem-10 ALGOL System that reads programs written in DECsystem-10 ALGOL and converts them into a form (relocatable binary) that is acceptable to the DECsystem-10 Linking Loader. The compiler is also responsible for finding errors in the user's source program and reporting them to the user.

Slight constraints are imposed on the way the user writes his program. These constraints, made to gain the most desirable feature of a single-pass compiler, concern the order in which the user declares the identifiers in the program and the use of forward declarations under certain special circumstances.

Such a compiler can process ALGOL programs rapidly and does not require the use of any backing store. The minor restrictions imposed will not normally affect the user.

1.3.1 Compiler Extensions

The following ALGOL-60 extensions are allowed by the compiler:

a. A LONG REAL type, equivalent to FORTRAN's double precision, is added that gives the user power to handle double-precision real numbers.

b. An EXTERNAL procedure facility allows the user to compile procedures separately from the main program.

c. A WHILE statement, and an abbreviated form of the FOR statement, allow the user greater flexibility of iteration.

d. A new type STRING allows the user to manipulate strings of various size bytes. In addition, the user can individually manipulate the bytes within a string by means of a byte subscripting facility.

e. An integer remainder function REM, is provided.

f. Assignments are permitted within expressions.

g. Delimiter words may be represented in either reserved word format (upper case) or as non-reserved words enclosed in single quotes (primes).

h. Constants of type REAL may be expressed as an integer part and a decimal part only as in FORTRAN.

The compiler accepts reserved word delimiters in normal mode, but it can also accept programs using non-reserved delimiter words enclosed in primes. Refer to Chapter 18.

1.3.2 Compiler Restrictions

If the user is unfamiliar with any of the following terminology, he should refer to the Revised Report and to Paragraph 1.5.

The compiler imposes the following restrictions on ALGOL-60:

a. Numeric labels are not permitted.

b. All formal parameters must be specified.

c. Identifiers are restricted to 64 characters in length.

d. Arrays and scalars must be declared before switches and procedures.

e. Forward references for procedures and labels must be given under certain circumstances.

1.4 THE ALGOL OPERATING ENVIRONMENT

Programs compiled by the ALGOL compiler are run in a special operating environment that provides special services, including input/output facilities for the object program.

The ALGOL operating environment consists of:

a. The ALGOL Library, known as ALGLIB - a set of routines, some of which are incorporated into the user's program by the linking loader.

b. The ALGOL Object Time System, known as ALGOTS - responsible for organizing the smooth running of the program and providing services such as core management, peripheral device allocation, and fault monitoring in case the program encounters an error condition at run time.

Refer to Chapters 17 and 18 for a description of ALGLIB and ALGOTS.

1.5 TERMINOLOGY

Some of the following words, used in this manual, may be new to the reader. Many have a FORTRAN equivalent; where such an equivalent exists, it is enclosed in parentheses.

Delimiter Word - a single, English language word that is an inherent part of the structure of the ALGOL language. Such words cannot normally be used for other purposes. Example: BEGIN IF ARRAY.

Identifier - a name, established by user declaration, that represents some quantity within a program.

Label (Statement Number) - an identifier used to mark a certain statement in a program. Control of program execution can be transferred to the statement following the label. A numeric label (not available in DECsystem-10 ALGOL) is similar to a FORTRAN statement number.

(continued on next page)

Procedure (Subroutine, Function) - part of a program, which may be invoked by "calling". In general, parameters are supplied as arguments and a result may be returned.

Parameter (Formal Parameter - Dummy Variable, Actual Parameter - Argument) See Procedure. - A Formal Parameter is an identifier used within the procedure that represents the argument supplied when the procedure is called.

CHAPTER 2
PROGRAM STRUCTURE

2.1 BASIC SYMBOLS

DECsystem-10 ALGOL programs consist of a sequence of symbols from the DECsystem-10 ASCII character set. The meaning of individual characters, given in Table 2-1, is much the same as in other high-level languages.

Table 2-1
DECsystem-10 ALGOL Symbols

Symbol	Meaning or Use
A - Z	Used to construct identifiers and delimiter words.
a - z	Lower case letters; are treated as upper case letters except when they appear in string constants and ASCII constants.
0 - 9	Decimal digits; used to construct numeric constants and identifiers.
+	Arithmetic addition operator.
-	Arithmetic subtraction operator.
*	Arithmetic multiplication operator.
/	Arithmetic division operator.
↑	Arithmetic exponentiation operator.
( )	Parentheses; used in arithmetic expressions and to enclose parameters in procedure specifications and calls.
[ ]	Square brackets; used to enclose subscript bounds in array declarations, and array subscript lists.
,	Comma; general separator, placed between array subscripts, procedure parameters, items in switch lists, etc.
.	Decimal point; used in numeric constants and byte subscripting. Also, used as a readability symbol in identifiers.
;	Semicolon; used to terminate statements.

(continued on next page)

Table 2-1 (Cont)
DECsystem-10 ALGOL Symbols

Symbol	Meaning or Use
:	Colon; used to indicate labels, and separate lower and upper bounds in array declarations.
=	Equality; used in arithmetic and string comparisons.
#	Nonequality.
< >	Less than, greater than.
& @	Introduces exponent in floating-point numbers.
'	Prime, or single quote; used to enclose delimiter words when the non-reserved word implementation is used.
"	Opening and closing string quotes.
!	Comment.
%	Introduces an octal constant.
$	Introduces an ASCII constant.
←	Alternative to := (refer to Table 2-2).

2.2 COMPOUND SYMBOLS

Compound symbols consist of two adjacent basic symbols. Any intervening spaces or tabs do not affect their use. The compound symbols are shown in Table 2-2.

Table 2-2
Compound Symbols

Symbol	Usage
:=	Assignment
< =	Less than or equal to
>=	Greater than or equal to

2.3 DELIMITER WORDS

Certain upper-case letter combinations are reserved as part of the structure of the language and may not be used as identifiers unless the compiler used is a version accepting delimiter words in single quotes. Such an option is selected by using a special switch option (refer to Chapter 18). It is assumed

throughout this manual that the standard method of delimiter word representation is used, that is, reserved words.

For example, the delimiter word

```
BEGIN
```

will always appear in the text of this manual as shown above and cannot be used as an identifier in a program. If the alternative method of representation is used, it would appear as

```
'BEGIN'
```

and

```
BEGIN
```

could be used as an identifier. Table 2-3 contains a list of all the delimiter words used in the language.

Table 2-3
Delimiter Words Used in DECsystem-10 ALGOL

Reserved Word	Chapter Reference
AND	5.2.1
ARRAY	9
BEGIN	10
BOOLEAN	5.2
CHECKOFF	18
CHECKON	18
COMMENT	2.4
DIV	5.1
DO	8
ELSE	7.3
END	10
EQV	5.2.1
EXTERNAL	11.9
FALSE	4.2
FOR	8
FORWARD	11.8
GO	7.2
GOTO	7.2
IF	7.3
IMP	5.2.1
INTEGER	3.2
LABEL	11
LINE	18
LISTOFF	18
LISTON	18
LONG	3.2

(continued on next page)

DECsystem-10 ALGOL also permits the use of a decimal point as a "readability symbol" in the alphabetic portion of identifiers. These readability symbols can appear between two alphabetic characters of an identifier and are ignored by the compiler. Thus:

```
ONCE.AGAIN
```

and

```
PI.BY.TWO
```

have exactly the same effect as

```
ONCEAGAIN
```

and

```
PIBYTWO
```

respectively.

Note that

```
ALPHA3.5
```

and

```
BETA.22
```

are not identifiers, since the decimal point does not appear between two alphabetic characters.

3.2 SCALAR DECLARATIONS

A declaration reserves an identifier to represent a particular quantity used in a program. Such declarations are mandatory in ALGOL. At any particular point during program execution, the form of the variable or quantity associated with the identifier depends on the type of variable. The type of variable is controlled by the type of identifier which represents it.

There are five scalar variables, that is, variables which contain a single value:

a. Integer
b. Real
c. Long Real
d. Boolean
e. String

Integer, real, and long real variables are capable of holding numerical values of the appropriate type (and only of that type). The range of values is as follows: integer: -34,359,738,368 through 34,359,738,367; real and long real: approximately -1.7&38 through 1.7&38; values less than approximately 1.4&-39 in magnitude are represented by zero.

Boolean variables (similar to FORTRAN's Logical variables) can hold a Boolean quantity, which is usually one of the states TRUE or FALSE but, in general, can be any pattern of 36 bits.

String variables are somewhat more complicated. A full discussion of their properties is presented in Chapter 13. At this point, it is sufficient to say that string variables are really pointers to byte strings.

All of the above variables can be declared for use by preceding a list of the identifiers to be used by the appropriate delimiter word for their type. Throughout this manual, a "list of items" consists of those items arranged sequentially and separated by commas.

Examples:

```
INTEGER I,J,K;
LONG REAL DOUBLE,P,Q,ELEPHANT;
BOOLEAN ISITREALLYTRUE;
STRING S,T;
```

CHAPTER 4
CONSTANTS

4.1 NUMERIC CONSTANTS

There are three forms of numeric constants:

a. Integer constants
b. Real constants
c. Long Real constants

4.1.1 Integer Constants

Integer constants consist of a number of adjacent decimal digits, subject to the constraint that the number represented must be in the range 0 through 34,359,738,367.

NOTE

Any preceding sign that appears in the program is not considered part of the constant.

Examples:

```
3

24

9276541
```

See also Section 4.3

4.1.2 Real Constants

Real constants consist of a decimal number (containing either an integral part or a fractional part, or both) followed by an optional exponent. If the decimal number is unity, it may be omitted. The exponent consists of either the & or @ symbol followed by an optionally signed integer. This has the effect of multiplying the decimal number by the power of ten specified in the exponent. If no decimal number appears, a value of unity is assumed.

The range of real constants is approximately 1.4&-39 to 1.7&38; numbers less than 1.4&-39 are represented by zero. Real numbers are stored to a significance of approximately eight and one-half decimal digits.

Examples:

Representation	Value
3.141592653589793	3.14159265
.0001	0.0001
4.37&5	437000.0
5&-3	0.005
&-6	0.000001

4.1.3 Long Real Constants

Long real constants are used to represent numeric quantities to approximately twice the precision available with real numbers: about seventeen decimal digits. Long real constants are formed by writing a real constant in floating-point form, but replacing the & or @ by && or @@.

The range of long real constants is the same as that of real constants, except numbers below approximately 3.0&&-30 can only be represented to single precision due to hardware considerations.

Examples:

Representation	Value
3.14159265358979323846&&0	3.1415926535897932
12&&-3	0.012

4.2 OCTAL AND BOOLEAN CONSTANTS

Octal constants consist of the symbol % followed by a number of octal digits. Up to twelve significant digits may appear (leading zeros are ignored); these digits are right justified.

Examples:

```
%777777777774

%0470
```

Octal constants may only be used in Boolean expressions.

Boolean constants consist of the words TRUE and FALSE. They are equivalent to the octal constants %777777777777 and %000000000000, respectively.

4.3 ASCII CONSTANTS

Up to five ASCII symbols can be packed right justified to give an integer-type constant. The format is a dollar sign ($), followed by up to five ASCII symbols enclosed within a delimiting symbol pair. The leading delimiter symbol immediately follows the $, and may be a readable character or an invisible one such as a space. Thus, the user can generate a single ASCII character constant by placing one space on each side of it, and preceding the triplet by a dollar sign.

Examples:

Text	Octal Value
$ A $/01234/	000000 000101 160713 516674

4.4 STRING CONSTANTS

String constants allow the user to store any reasonable length string of ASCII characters within a program. The length of such a constant is restricted only by the amount of core storage available to the user for the execution of the program. String constants may be used, typically, to output a message during the execution of the program or as values assigned to string variables.

The string of symbols is enclosed within quotes ("). There are restrictions on the symbols that may appear within the string.

a. [ ] ; and " may not appear alone.
b. [and] may appear if they are properly paired.
c. Single occurrences of [ ] ; and " are represented by [[ ]] ;; and "" respectively.
d. Where a string has to be broken across two or more lines of source, the carriage return and line feed characters can be ignored by preceeding them with a control-back arrow character.

Note that [[and]] are stored as such in the byte string generated by the compiler. ;; and "" are stored as a single ; or ", respectively. The restriction on the representation of ; is made to protect the user against quoting his whole program by missing out a ".

Square brackets are used to enclose symbols that have a specific effect when the string is output. These are discussed in Paragraph 16.6.2.

Examples:

```
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"

"REMEMBER THAT SPACES ETC. ARE SIGNIFICANT"

"[P5C]INPUT DATA:[5C]"

""""A[[I]] := 0.1;;""""
```

CHAPTER 5
EXPRESSIONS

5.1 ARITHMETIC EXPRESSIONS

DECsystem-10 ALGOL arithmetic expressions are written in a form similar to that used in FORTRAN and many other high-level scientific computer languages. The usual algebraic rules concerning precedence of operators and brackets are followed (see Table 5-1).

Table 5-1
Operator Precedence

Operator	Priority (decreasing)
parentheses	1
exponentiation	2
multiplication and division	3
addition and subtraction	4

There are two additional operators, DIV and REM, that indicate integer division and remainder, respectively. They have the same precedence as ordinary division. Within the precedence scheme, the order of evaluation is always from left to right. For example:

```
X↑Y↑Z means (X↑Y)↑Z
```

and

```
I DIV J REM K means (I DIV J) REM K
```

Unlike FORTRAN, when ordinary division of one integer by another is performed, the real result is not rounded to an integer value.

The difference between the various types of division is clarified by the following examples:

7/4 yields a result of 1.75, whereas
7 DIV 4 yields a result of 1, and
7 REM 4 yields a result of 3

The interpretation of integer division for negative integers follows:

Let M, N > 0, then

```
-M DIV N = M DIV (-N) = -(M DIV N)
-M DIV (-N) = M DIV N
```

The integer remainder operator, REM, is defined so that for all integral M,N:

```
M REM N = M - N*(M DIV N)
```

5.1.1 Identifiers and Constants

Arithmetic expressions consist of operands, that is, identifiers and constants, of the three types, integer, real and long real, together with the arithmetic operands + - * / DIV REM and ↑ and parentheses where necessary.

Identifiers are used to represent variables whose values are used when they appear in some calculation.

Since automatic conversion takes place as necessary when an expression is evaluated, the user may freely mix the three different types of identifiers and constants.

Integer quantities may have more precision than can be represented in a real variable. The user must beware of possible loss of significance in integral quantities used in mixed type expressions.

5.1.2 Special Functions

Three special functions are provided for use in arithmetic expressions. The first is the transfer function, ENTIER, which converts a real or long real quantity into an integer quantity defined as the largest integer value not exceeding the argument.

Thus

```
ENTIER(3.5) = 3
```

and

```
ENTIER(-3.5) = -4
```

The special function ABS yields the absolute value (also known as the modulus) of its argument. The argument may be any integer, real, or long real quantity; the result is always of the same type as the argument.

Thus

```
ABS(-3.5) = 3.5
```

and

```
ABS(-3) = 3
```

The special function SIGN is the signum function whose argument can be integer, real, or long real. The result is always integral, being minus one or zero or plus one, depending on whether the argument is negative, zero, or greater than zero, respectively.

Thus

```
SIGN(-3.5) = -1
SIGN(0) = 0
SIGN(3.5) = 1
```

NOTE

ENTIER, ABS, and SIGN are not delimiter words. They may be used for other purposes in a program.

Examples of simple arithmetic expressions follow:

```
X
I + 3
X*Y/Z
P+Q/R
X2 + Y
XJ-4
J + ENTIER(K-2)
SIGN(ENTIER(J/K) + 1)
(X + Y) ↑ (-I)
```

5.2 BOOLEAN EXPRESSIONS

Boolean expressions involve Boolean identifiers, Boolean and octal constants, arithmetic conditions, and Boolean operators interspersed in an order similar to that of arithmetic expressions.

5.2.1 Boolean Operators

There are five Boolean operators listed here in decreasing order of precedence.

a. NOT (unary operator)
b. AND
c. OR
d. IMP (implication)
e. EQV (equivalence)

NOT is a unary operator that complements a Boolean quantity in the same way that a unary minus sign negates an arithmetic quantity in an arithmetic expression. In this case, it changes FALSE to TRUE, and vice versa.

Table 5-2 gives the result of A OP B where OP stands for one of the Boolean operators AND, OR, IMP, or EQV, for all values of A and B.

Table 5-2
Function of Boolean Operators

A	FALSE		TRUE	
B	FALSE	TRUE	FALSE	TRUE
A AND B	FALSE	FALSE	FALSE	TRUE
A OR B	FALSE	TRUE	TRUE	TRUE
A IMP B	TRUE	TRUE	FALSE	TRUE
A EQV B	TRUE	FALSE	FALSE	TRUE

In addition, the following theorems hold true:

A IMP B is equivalent to NOT A OR B,
A EQV B is equivalent to A AND B OR NOT A AND NOT B.

5.2.2 Evaluation of Boolean Variables

Actually, Boolean variables may have a value consisting of any pattern of bits, rather than be confined to the values TRUE and FALSE. The logical operations operate on a bit-by-bit basis according to the preceding rules.

The actual test employed to determine the truth of a Boolean expression such as

B AND C

is to evaluate it and regard it as true if its value is nonzero, i.e., at least one bit is set, otherwise it is false.

This is particularly important when octal constants are used in Boolean expressions. For example, if the user wishes to test a particular bit in a Boolean variable, an appropriate octal constant can be used, for example:

B AND %1

is a Boolean expression that is true if and only if the bottom (least significant) bit of B is a one.

5.2.3 Arithmetic Conditions

Arithmetic conditions are used as operands in Boolean expressions. They consist of two arithmetic expressions coupled with a comparator. The comparator, which decides the particular type of test to be performed on the two expressions, is one of the following:

<	less than
<=	less than or equal to
=	equals
>	greater than
>=	greater than or equal to
#	not equal to

Such an arithmetic condition can be regarded as true or false according to whether the condition specified by the comparator is met when the arithmetic expressions on each side of it are evaluated. The resulting condition may form part of a Boolean expression.

The following examples of Boolean expressions, shown in Table 5-3, also involve arithmetic conditions.

5.3 INTEGER AND BOOLEAN CONVERSIONS

An integer quantity can be converted to a Boolean quantity by means of the dummy function BOOL. Similarly, the dummy function INT converts a Boolean quantity to an integer quantity.

Table 5-3
Boolean Expressions

Expression	Meaning
NOT B	NOT B
B AND NOT C	B AND (NOT C)
A OR B AND C	A OR (B AND C)
B EQV X<Y	B EQV (X<Y)
X+Y<Z AND B OR P=Q	(((X+Y)<Z) AND B) OR (P=Q)

The value passed by these functions is unchanged: the functions are included for semantic correctness. Thus:

```
BOOL(I)
```

may be regarded as a Boolean operand, and

```
INT(B)
INT(%400000000000)
```

as integer operands.

BOOL and INT are not reserved words. They can be used for other purposes by declaring them as required. However, this practice should be avoided since it could lead to confusion.

CHAPTER 6
STATEMENTS AND ASSIGNMENTS

6.1 STATEMENTS

The statement is the basic operational unit in ALGOL-60. It describes an operation to be performed at run time, such as an assignment.

6.2 ASSIGNMENTS

Assignments convey the value produced by the execution of an expression to a destination variable of the appropriate type. This is done by writing the destination identifier, followed first by the symbols : and = and then by the expression to be evaluated. Thus

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

causes the result of the addition of the values contained in the variables Y and Z to be placed in the variable X.

When an assignment is made to a variable type differing from that of the result of the expression, a type conversion is performed. Integer, real and long real expressions may be assigned to variables of any of these three types, but not to any other types. Boolean and string expressions can only be assigned to a variable of their own type.

If a real or long real value is assigned to an integer type variable, a rounding process occurs.

```
I := X
```

results in an integral value equal to

```
ENTIER(X + Ø.5)
```

being assigned to I.

When an integer expression is assigned to a real or long real variable, a conversion to that type is performed. Real to long real conversion simply consists of zeroing the low-order precision word of

the long real result after assignment of the real result to the high-order part of the long real variable. Long real to real assignments truncate the low-order part of the long real expression, after appropriat rounding.

6.3 MULTIPLE ASSIGNMENTS

A value may be assigned simultaneously to several variables of the same type by a multiple assignment. This takes a form such as

```
P := R := S := X + Y - Z
```

where the result of adding Y to X and subtracting Z is assigned to P, R, and S simultaneously.

All identifiers on the left-hand side of a multiple assignment must be of the same type. If the user wishes to assign a value to two or more different types of variables, the "assignment within expression" (embedded assignment) feature must be used, as below.

A parenthesized assignment may be substituted for any operand in an expression. For example,

```
X := (Y := P+Q)/Z
```

This causes the embedded assignment to be made after the inner expression P+Q is evaluated. Where a type conversion is performed as part of an embedded assignment, the operand type is the same as that assigned to the variable in the embedded assignment. Thus

```
X := (I := 3.4)
```

sets I equal to 3 and X equal to 3.0.

6.4 EVALUATION OF EXPRESSIONS

All expressions in DECsystem-10 ALGOL are evaluated observing the normal algebraic rules of precedence, including bracketing.

Within the precedence structure, expressions are always evaluated from left to right. For example, if X is a scalar, and F a function procedure (see Chapter 11) that alters X,

```
X := X+F
```

may have a different effect than

```
X := F+X
```

This is known as a "side effect".

Consider also:

```
A[I] := (I := I+1)
```

The subscript I is always evaluated before I is incremented, as it is to the left of the embedded assignment, within the statement. Thus the above expression is equivalent to

```
J := I; I := I+1; A[J] := I
```

The user can always predict the order of evaluation of an expression and can count on such things as

```
X := (P := P+Q)/(P+R)
```

being evaluated correctly, thus giving the same result as

```
P := P+Q
X := P/(P+R);
```

6.5 COMPOUND STATEMENTS

A compound statement consists of a number of statements, preceded by BEGIN, separated by semicolons, and terminated by END. ALGOL statements, unlike those in FORTRAN, are terminated by a semicolon not by the end of a line of text.

For example:

```
BEGIN
        I := 3; J := 4;
        K := I + J;
        X := K
END
```

is a compound statement. Semicolons do not have to appear after the BEGIN or before the END; BEGIN and END act as a type of bracket.

The usefulness of compound statements will become apparent in later chapters.

CHAPTER 7

CONTROL TRANSFERS, LABELS, AND CONDITIONAL STATEMENTS

7.1 LABELS

A label is a method of marking a place in a program so that control can be transferred to that point from elsewhere in the program.

DECsystem-10 ALGOL uses identifiers as labels. These identifiers are placed before statements and are followed by a colon. Numeric labels are permitted in the Revised Report, but are not implemented in DECsystem-10 ALGOL. Most implementations of ALGOL-60 do not allow integer labels.

For example:

```
COMP: X := X + Y
```

is a statement labeled by COMP.

More than one label can be attached to a statement if required; thus,

```
LAB1: LAB2: Y := 0
```

7.2 UNCONDITIONAL CONTROL TRANSFERS

A transfer of control, or "jump", to a statement in a program is effected by a GOTO statement. This statement consists of the word GOTO followed by the name of the label attached to the relevant statement. The two words GO TO can be used instead of the word GOTO in any statement where GOTO can be used. Thus:

```
BEGIN    INTEGER I,J,K;
LAB:     I := J := 3;
         K := I + J;
         GOTO LAB
END
```

is an example of a somewhat tedious program. Clearly, to write any reasonable program, it is necessary to be able to jump conditionally.

8.1.1 STEP-UNTIL Element

This particular form deserves closer inspection, because it is not quite as simple as it appears. For example, consider

```
FOR I := 1 STEP I UNTIL N DO S
```

The statement S is obeyed with I taking an initial value of 1, and being incremented by I until the final value N is achieved. The question is, "Is the I after the STEP recalculated during each turn around the loop, or does it have a constant value equal to the initial value of I?"

The answer is slightly more complicated. Consider the general case

```
FOR V := E1 STEP E2 UNTIL E3 DO S
```

This is defined to have exactly the same effect as

```
        V := E1;
L1:     IF (V - E3)*SIGN(E2) > 0 THEN GOTO L2;
        S;
        V := V + E2;
        GOTO L1;
L2:
```

Clearly, the value of I following the STEP in the previous example is evaluated, if necessary, twice during each turn around the loop, once in the sign test at L1, and again to update V. ALGOL allows the user to modify V, E1, E2, and E3 freely throughout the loop, and takes account of all these changes in the evaluation of the loop.

NOTE

DECsystem-10 ALGOL allows the user the abbreviated form

FOR V := E1 UNTIL E3 DO S

instead of

FOR V := E1 STEP 1 UNTIL E3 DO S

8.1.2 WHILE Element

A FOR statement with a single WHILE element takes the form

```
FOR V := E WHILE B DO S
```

This is interpreted as follows:

```
L1:     V := E.
        IF NOT B THEN GOTO L2;
        S;
        GOTO L1;
L2:
```

Once again, the complexity of the loop may be affected by changing V and E within the loop.

8.2 WHILE STATEMENT

The WHILE statement is an enhancement of ALGOL-60 provided in DECsystem-10 ALGOL. It takes the general form

```
WHILE B DO S
```

and is interpreted as follows:

```
L1:     IF NOT B THEN GOTO L2;
        S;
        GOTO L1;
L2:
```

8.3 GENERAL NOTES

1. Within a FOR statement of any kind, the user can change the controlling variable or any other variable appearing within the action of the loop. Such changes predictably affect the execution of the loop by the rules given above.
2. On exit from a FOR statement either by jumping out of the loop or by exhausting the FOR elements, the controlling variable has a well-defined value equal to the last assigned value of the controlling variable. This may not be true of other ALGOL-60 implementations. Section 4.6.4 of the Revised Report should be studied carefully in this connection.

CHAPTER 9
ARRAYS

9.1 GENERAL

Arrays are essentially collections of variables of the same type, allowing the user to address them individually by means of a common name and a unique subscript or subscripts. In the simplest case, an array is a vector and is known as a one-dimensional array. A matrix is a two-dimensional array, etc.

There is no limit to the number of subscripts allowed, other than those imposed by the ability of the computer to store the array.

9.2 ARRAY DECLARATIONS

Arrays may be of type integer, real, long real, Boolean, or string. They are declared in a similar fashion to scalar variables, except the size of the array must be stated. For each subscript that the array possesses, a lower and an upper bound, called the "bound pair" for that subscript, must be given.

For example, to declare two one-dimensional integer arrays A and B with lower bound 1 and upper bound 5:

```
INTEGER ARRAY A,B[1:5]
```

Note that the lower and upper bounds are enclosed in square brackets and separated by a colon.

When there are two or more subscripts, the declaration is similar, and the bound pairs are separated by commas. Thus

```
LONG REAL ARRAY P,Q,R[-5:2,0:10]
```

declares three real arrays, P, Q and R, with the first subscript bounded by -5 and 2 and the second subscript bounded by 0 and 10.

It is possible to declare arrays of different sizes in the same statement provided they are of the same type:

```
REAL ARRAY A[1:10], B,C[1:10,1:12]
```

Note also that in the case of real arrays, the REAL may be omitted in the declaration, and is assumed by default, thus:

```
ARRAY A[1:10], B,C[1:10,1:12]
```

The bounds in an array need not be static, as in the examples above. In general, they may be any arithmetic expressions, which are evaluated to give an integral value for the individual bound pairs. The use of such dynamic array declarations will become apparent later.

9.3 ARRAY ELEMENTS

An individual element of an array can be referred to by following the name of the array by a list of subscripts in square brackets. The number of subscripts must be identical to the number in the array declaration. Thus, a typical element of A used in the last declaration might be

A[5] or A[9] or generally, A[I]

where I is some integer expression or, in general, any expression whatsoever, with the limitation that its value when used as a subscript and evaluated as an integer is in the range 1 through 10, the bounds of the array A.

As an example of the use of arrays, consider the declaration

```
REAL ARRAY D,E,F [1:10,1:10]
```

and suppose that it was required to set F equal to the matrix product of D and E:

```
FOR I := 1 UNTIL 10 DO
FOR J := 1 UNTIL 10 DO
BEGIN   X := 0;
        FOR K := 1 UNTIL 10 DO X := X + D[I,K]*E[K,J];
        F[I,J] := X
END
```

Note that X is used to accumulate the inner product of the multiplication for all values of I and J. It would be very inefficient not to use such a variable, because F would otherwise be needlessly involved in the inner loop of the computation.

Also, note that an element of an array of a particular type may be used anywhere that a scalar variable of the same type may be used, even in such places as the controlling variable in a FOR statement.

CHAPTER 10
BLOCK STRUCTURE

10.1 GENERAL

ALGOL program structure is somewhat more complicated than other high-level languages, such as FORTRAN. An ALGOL program consists of a number of "blocks" arranged hierarchically; a block consists of the words BEGIN and END enclosing declarations and (optionally) statements.

Thus:

```
BEGIN
      BEGIN
      END
      BEGIN
                  BEGIN
                  END
      END
END
```

is an ALGOL program, assuming appropriate declarations and statements in the blocks.

The block structure offers the user many interesting features not available in non-block structured languages. For instance, the user may declare an identifier that appears to conflict with another identifier in an enclosing block. Thus:

```
BEGIN       INTEGER I;
            BEGIN INTEGER I;
            END
END
```

In fact, there is no conflict as there are two different Is. The only I that statements in the outer block can "see" is the one in the outer block. Similarly, any statements in the inner block will always use the I in that block. Such a declaration in an inner block is known as a "local" variable; it takes precedence over declarations occurring at an outer or more "global" level. In general, all variables can be "seen" from any point in a program that is either in the same block as the declaration or in a block that is enclosed by the block in which the declaration of the variable occurred. Note that a more local variable is always taken in preference to a relatively global variable. Consider the following example:

```
BEGIN       INTEGER I,J;
            [1]
            BEGIN       INTEGER J,K
                        [2]
            END;
            BEGIN       INTEGER I,K
                        [3]
            END
END
```

Any statements occurring at point [1] can see the declarations of I and J, which are local, but cannot see the declarations of J and K in the first inner block, or the declarations of I and K in the second inner block. At [2], the local variables J and K can be seen, as can the global variable I in the outer block. The global variable J is not seen because the local variable J takes precedence over it; the variables I and K in the second inner block are not seen at all. A similar situation occurs at [3]; here both local variables I and K, as well as the global variable J, are seen.

Note that the "scope" of a variable is the set of all places in a program where it can be seen and therefore used. This term will be used frequently throughout this text.

In general, it is more efficient to use local variables in preference to global ones. This statement is also true of most ALGOL-60 implementations. Where a non-local variable is used frequently, it is advisable to assign its value to a local variable and use that in preference. For example:

```
BEGIN       INTEGER I;
            .....
            I := .....
            BEGIN       INTEGER II;
                        II := I;
                        .....
                        ..... II .....
                        END
                        .....
END
```

Here, in the inner block, a local variable II is used, and assigned the value of the global variable I for use throughout the local block.

10.2 ARRAYS WITH DYNAMIC BOUNDS

The concept of the scope of a variable can be applied most usefully to arrays. In DECsystem-10 ALGOL, all arrays are constructed at execution time, that is, no fixed space is reserved for them by the compiler, irrespective of whether their bounds are static or dynamic. When a declaration of an array is encountered within a block, the space required to construct it is obtained and the array is laid out. When the end of the block enclosing the array is reached, that is, the array variable is no longer within scope, the space utilized by the array is recovered and can be used later for other arrays.

Consider the case of a problem in which the size of an array to be used in a calculation is dependent on the data to be processed. The programmer has the choice of making the array large enough to cope with the worst case (in many languages he does not have any choice at all) or constructing the array with dynamic bounds to suit the size required by the particular data. The first method has the disadvantage of wasting space on many occasions; the latter method only has the minor disadvantage of the overhead needed to construct the array. Such overhead is very small compared to the running time of most programs; therefore, the second method is more desirable.

Consider the following example:

```
BEGIN       INTEGER N;
L:          N := .....
            .....
            BEGIN ARRAY A[1:N,1:N];
            .....
            END;
            GOTO L
END
```

A value for N is calculated in this example, possibly dependent on some data read into the program, and used to declare the array A, which is used to process the data in the inner block. When the end of the inner block is reached, the space used by A is recovered and control passes to L, where another value for N is calculated, and the process repeated.

CHAPTER 11
PROCEDURES

Procedures are similar in concept to the FORTRAN subroutine, although more sophisticated and general in their possible applications.

A "procedure" is a portion of an ALGOL program that is given a name to identify it and can be "called" from any part of a program which is in the scope of the body of the procedure. A procedure can execute a number of statements, or it can return a value if it is a function procedure. In addition, it may or may not have parameters.

In DECsystem-10 ALGOL, a procedure can be one of the following types: integer, real, long real, Boolean or string, or it may be typeless. The formal parameters of a procedure, known as "dummy variables" in FORTRAN, can be one of the following types: integer, real, long real, Boolean or string, as scalars, arrays or procedures, or label. There are seventeen different types of parameters. In addition, all of these parameters may appear in two different modes, neither of which is the same as FORTRAN's method of handling parameters.

11.1 PARAMETERS CALLED BY "VALUE"

Calling parameters by "value" is the most common and, with the exception of arrays, the most efficient way to pass a parameter to a procedure. The value of the expression presented in a procedure call, known as the actual parameter, is evaluated on entry to the procedure and assigned to a formal parameter within the procedure. This formal parameter acts exactly as if it were a local variable of the procedure which is initialized with the value of the actual parameter supplied in the call to the procedure.

Since, in the case of arrays or strings, a new copy of the array or string is made, this type of parameter-passing for arrays and strings (if they are very long) should be avoided unless it is specifically required.

11.2 PARAMETERS CALLED BY "NAME"

Calling parameters by "name" is a very sophisticated method of passing a parameter to an ALGOL procedure. Whenever the formal parameter associated with the actual parameter in a procedure body

appears in the body of the procedure, the actual parameter is re-evaluated as if it appeared in the procedure body at that point. For example, if the actual parameter were an array element such as

```
A[I]
```

it would be re-evaluated using the value of I available each time the formal parameter is used, not the value of I at the time the procedure body is entered.

Table 11-1 shows the different types of formal parameters, together with valid actual parameters that can be substituted in a procedure call.

Table 11-1
Parameters in a Procedure Call

Formal Parameter Type	Permissible Actual Parameter
Integer Real Long Real	Any arithmetic expression
Boolean	Any Boolean expression
String	Any string expression (refer to Chapter 13)
Label	A label or switch element (refer to Chapter 12 and Paragraph 14.4)
Switch	A switch
Integer Array	An array of type integer*
Real Array (or Array)	An array of type real*
Long Real Array	An array of type long real*
Boolean Array	An array of type Boolean
String Array	An array of type string
Procedure	A non-type procedure
Integer Procedure Real Procedure Long Real Procedure	A procedure of type integer, real, or long real
Boolean Procedure	A procedure of type Boolean
String Procedure	A procedure of type string

*In the case where the array parameter is called by value, any arithmetic type (integer, real, or long real) array is allowed as an actual parameter. A type conversion takes place during the copying process.

11.3 PROCEDURE HEADINGS

Procedure headings identify the type of procedure and the number and type of its parameters. They precede the body of the procedure.

A procedure heading consists of:

a. The type of procedure (omitted in the case of typeless procedures).

b. The word PROCEDURE followed by the name of the procedure.

c. A semicolon if the procedure has no parameters; otherwise

d. A list of the formal parameters, enclosed in parentheses, and followed by a semicolon.

e. Specifications of the formal parameters. Omitting formal parameter specifications, this looks like

```
LONG REAL PROCEDURE LR;

BOOLEAN PROCEDURE BOOLCON (I,J,K);

PROCEDURE CALC(THETA,X);
```

The formal parameter specification that follows consists of a list of descriptions of the formal parameters, appearing in any order, and a value specification if any of the parameters are to be called by value. (If this is omitted, the parameters, by default, will be called by name.) For example, the specification of the formal parameters for the second example above might be:

```
VALUE I,J; INTEGER I,J,K;
```

meaning that all three formal parameters are of type integer (scalars), and I and J are to be called by value, while K is to be called by name. A typical formal parameter specification for the third example might be:

```
REAL PROCEDURE THETA; ARRAY X;
```

11.4 PROCEDURE BODIES

The body of a procedure is that part that follows the procedure heading. It consists of a single statement, a compound statement, or a block. In the last-mentioned case, there may be declarations of local variables within the block, and also other blocks or procedures. Consider the following examples of realistic procedures:

a. A real procedure, squareroot, to calculate the square root of a real quantity. The first parameter is the argument; the second is a label that is used as an escape if the argument is found to be negative. The result of the procedure is the square root of the argument. Note how the result of the calculation is assigned to the procedure by placing the name of the procedure on the left-hand side of an assignment.

```
REAL PROCEDURE SQUAREROOT(X,L);
        VALUE X; REAL X; LABEL L;
BEGIN   REAL Y,Z;
        IF X < 0 THEN GOTO L;
        Y := (1 + X)/2;
IT:     Z := (X/Y + Y)/2;
        IF ABS(Z - Y) < 1&-6 THEN GOTO OK;
        Y := Z; GOTO IT;
OK:     SQUAREROOT := Z
END
```

The previous example uses the Newton-Rapheson method of finding the square root of a number: taking an initial approximation (1 + X)/2 and iterating until the difference between successive approximations is less than 1&-6. Although this is a very simple procedure, it is more enlightening with the aid of some commentary. The DECsystem-10 ALGOL alternative method of commentary (refer to Chapter 2) is used for brevity:

```
REAL PROCEDURE SQUAREROOT(X,L);
  VALUE X; REAL X; LABEL L;
BEGIN ! CALCULATES THE VALUE OF SQRT(X)
      USING THE NEWTON-RAPHESON METHOD.
      L IS USED FOR AN ESCAPE IF X < 0;
  REAL Y,Z;
  IF X < 0 THEN GOTO L;  ! EXIT IF X < 0;
  Y := (1+X)/2;          ! FIRST APPROXIMATION;
IT:
  Z := (X/Y + Y)/2;      ! ITERATE;
  IF ABS(Z-Y) < 1&-6
     THEN GOTO OK;       ! TEST FOR CONVERGENCE;
  Y := Z; GOTO IT;       ! OTHERWISE CONTINUE;
```

(continued on next page)

```
OK:
  SQUAREROOT := Z;          ! FINAL RESULT;
END
```

b. This function evaluates the sum of the values of any real procedure G over the integers 1 ..... N, where N is also a parameter of the procedure.

```
REAL PROCEDURE SUM(G,N);
        VALUE N; REAL PROCEDURE G; INTEGER N;
BEGIN   INTEGER I; REAL X;
        X := 0;
        FOR I := 1 UNTIL N DO X := X + G(N);
        SUM := X
END
```

Notice in this example how the formal parameter G is invoked so that the actual procedure that is substituted for G is called.

11.5 PROCEDURE CALLS

In the preceding example, the procedure G was "called". Since G is a function procedure, it is only necessary for its name to appear in an expression for the procedure to be entered with the actual parameters specified substituted for the formal parameters.

The procedure squareroot can be called in a similar way, for example:

```
P := SQUAREROOT(Z + 0.5)
```

causes the square root of Z + 0.5 to be calculated.

An example of the use of the procedure sum can be used to calculate the sums of the square roots of the first J integers, with the result squared, as follows:

```
X := SUM(SQUAREROOT,J)↑2;
```

Here is a further example of a procedure and its calls:

```
PROCEDURE MATRIXMULT(A,B,C,N);
 VALUE N; ARRAY A,B,C ; INTEGER N;
BEGIN    INTEGER I,J,K; REAL X;
         COMMENT THIS PROCEDURE PERFORMS THE MATRIX
         MULTIPLICATION OF B AND C AND PUTS THE RESULT
         IN A.  THE ARRAYS ARE ASSUMED TO BE SQUARE
         AND OF BOUNDS 1:N,1:N;

         FOR I := 1 UNTIL N DO
         FOR J := 1 UNTIL N DO
         BEGIN X := 0;
         FOR K := 1 UNTIL N DO X := X +
                B[I,K]*C[K,J];
                A[I,J] := X
         END
END
```

A typical call for this procedure might be

```
MATRIXMULT(E,F,G,N);
```

or

```
MATRIXMULT(E,F,F,N);
```

Since the arrays are called by name, a call such as MATRIXMULT(E,E,F,N); would give rather interesting results.

This call could be made to work by calling B and C by value. However, this would increase the overhead of the procedure considerably.

11.6 ADVANCED USE OF PROCEDURES

11.6.1 Jensen's Device

This method of using a procedure exploits the power and flexibility of the call-by-name concept. Consider the following example:

```
REAL PROCEDURE SUM(I,N,X); VALUE N; INTEGER I,N; REAL X;
BEGIN   REAL Y;
        Y := 0;
        FOR I := 1 UNTIL N DO Y := Y + X;
        SUM := Y
END
```

On the surface, the procedure appears to calculate the value of N*X. However, consider the call

```
Z := SUM(J,10,A[J]);
```

and remember that J and A[J] are parameters called by name. Since I and consequently J take new values, each X in the loop is evaluated as a particular value of A[J], using the value of J just assigned. Hence the above call calculates

```
A[1] + A[2] + ..... + A[10].
```

Similarly, the call

```
Z := SUM(K,M,A[I,K]*B[K,J]);
```

calculates the (I,J)th inner product of A and B.

11.6.2 Recursion

ALGOL procedures have the inherent ability of recursion, that is, they may call themselves, directly or indirectly, to any reasonable depth. (The only restriction is the amount of core storage available to the object program.)

An often-quoted and very inefficient method of calculating the factorial function of a small positive integer N is:

```
INTEGER PROCEDURE FACTORIAL(N); VALUE N; INTEGER N;
IF N = 1 THEN FACTORIAL := 1
ELSE FACTORIAL := N*FACTORIAL(N-1);
```

Note that this procedure has only a single statement, but no local variables. Therefore, it can be written in a very compact form. A call such as

```
J := FACTORIAL(6);
```

causes the procedure to be entered with N equal to 6. The call to FACTORIAL inside FACTORIAL enters the procedure a second time with N equal to 5, but this N is different from the one to the previous N, which retains its value of 6, as it is stored in a different space. In this particular case, FACTORIAL is entered six times, the last time with N equal to 1.

11.7 LAYOUT OF DECLARATIONS WITHIN BLOCKS

Declarations must always be made at the head of a block, before any assignments, procedure calls, etc., in the following order: 1) scalars and arrays and 2) procedures and switches (see Chapter 12).

Any procedure bodies that occur in a block should follow the declarations at the head of the block, although this is only enforced when necessary. Consider the following example:

```
BEGIN
PROCEDURE P(X); VALUE X; REAL X;
BEGIN    INTEGER J;
         .....
         J := I;
         .....
END;
INTEGER I;
```

The assignment of I to J within the body of P utilizes the I that is declared following the body of P, rather than some global I. However, the compiler has not yet seen this I and, therefore, cannot take any rational action. In a case such as this, the user must declare I before the body of P:

```
BEGIN    INTEGER I;
PROCEDURE P(X); VALUE X; REAL X;
BEGIN    INTEGER J;
         .....
         J := I;
         .....
END;
```

If the user neglects to declare I before P, the compiler can easily detect the condition, because either I is unknown at the time of the assignment to J, or else there is a more global I available, whereupon an error message will occur when the declaration of I is found following the body of P.

11.8 FORWARD REFERENCES

Although most ALGOL-60 compilers operate in two or more passes, the DECsystem-10 ALGOL compiler operates in one pass. Consequently, it has to make some minor restrictions to ALGOL-60 in order not to restrict the user in other ways.

A forward reference for a procedure has to be given when a procedure is called (either directly, or indirectly, by passing its name as an actual parameter in a procedure call) before its body is encountered by the compiler. In most cases the user can avoid this situation by a minor re-ordering of the program. However, in rare cases like the following, where procedure P calls procedure Q, and vice versa, a forward reference, as shown, must be given.

```
BEGIN
FORWARD REAL PROCEDURE Q;
PROCEDURE P(X); VALUE X; REAL X;
BEGIN REAL Y;
          .....
          Y := Q(X);
          .....
END;
REAL PROCEDURE Q(Z); VALUE Z; REAL Z;
BEGIN REAL F;
          .....
          F := P(Z);
          .....
END;
.....
```

In general, a forward reference consists of the word FORWARD, followed by the type of the procedure (omitted if the procedure is typeless), the word PROCEDURE, and the name of the procedure. For example:

```
FORWARD LONG REAL PROCEDURE INTEGRATE
```

or

```
FORWARD PROCEDURE PROBLEM
```

Note that the forward reference must occur in the same block as the procedure body to which it refers.

A forward reference has to be given for a label in one of the following rare cases:

a. The label is used as an actual parameter in a procedure call, and has not yet appeared in the program.

b. A variable of identical name has appeared in the program and is in the scope of the procedure call.

For example:

```
BEGIN REAL L;
        .....
BEGIN FORWARD L;
        .....
        P(L);
        .....
L: .....
END;
        .....
```

In this case, a forward reference for L must be given.

11.9 EXTERNAL PROCEDURES

If it is required to compile a procedure independently of a program (see Paragraph 18.1.1), an EXTERNAL declaration must be made in the program instead of the procedure. The form of this is the same as that of a FORWARD declaration, but with the word FORWARD replaced by EXTERNAL. For example:

```
EXTERNAL INTEGER PROCEDURE CALC
```

Such an EXTERNAL declaration can be made in any block within the program, and has the same scope as if the procedure appeared at that point.

At present all EXTERNAL procedure names referenced in a program or scanned in a library must differ in their first six characters, as only the first six characters are available to LINK.

11.10 ADDITIONAL METHODS OF COMMENTARY

Two further ways of writing commentary are available to the user in addition to COMMENT and ! described in Section 2.4.

11.10.1 Comment After END

Following the delimiter word END, the user may add commentary, terminated by a semicolon, with the following restrictions:

1. The commentary may only contain letters and digits.
2. If the reserved delimiter word mode of compilation is employed, any words appearing in the comment may not be delimiter words.

For example:

```
END OF PROC INVERT;
```

11.10.2 Comments Within Procedure Headings

This method of commentary allows the user to comment formal parameters in a procedure heading. This is done by enclosing the commentary, which may consist of letters only, between the symbols) and :(and omitting the comma on the left of the formal parameter. This cannot apply to the first formal parameter.

The example in Section 11.6.1 can thus be rewritten:

```
REAL PROCEDURE SUM(I) COUNT:(N) INCREMENT:(X);
```

In a similar fashion, a call to such a procedure can be commented. The following example uses the call to SUM in Section 11.6.1:

```
Z:=SUM(K) COUNTER:(M) CROSS PRODUCT: (A[I,K]*B[K,J]);
```

CHAPTER 12
SWITCHES

12.1 GENERAL

Switches enable the user to jump to one of a number of labels, depending on the value of an arithmetic expression. In addition, they provide an automatic detection when such an expression is out of range for the switch.

12.2 SWITCH DECLARATIONS

A switch declaration takes the form of the word SWITCH followed by (a) the name of the switch, (b) an assignment (:=), and (c) a list of labels, called switch elements, all of which must be in the scope of the switch declaration. For example:

```
SWITCH SW := LAB,L1,L2,OK,STOP
```

A switch name must follow the usual rules of scope with regard to its use and, therefore, must not conflict with any local variable of the same name.

In addition to the example above, a switch element itself may be one of the labels in the switch declaration.

12.3 USE OF SWITCHES

A jump to a particular label in a switch declaration is made by following the word GOTO with the name of the switch and an arithmetic expression in square brackets. Thus:

```
GOTO SW[I]
```

This causes control to pass to the I'th label in the switch declaration, unless I is negative or zero, or is larger than the number of switches in the switch declaration. In either case, there is no transfer of control. If the expression in square brackets is not integral, it is evaluated and rounded as usual.

Consider the following more complicated example:

```
SWITCH SW := LAB,L1,L2,OK,STOP;
SWITCH TW := L3,SW[J],L4;
.....
GOTO TW[I];
```

If I has the value 3, a jump to L4 occurs. If I has the value 2 and J has the value 1, a jump to LAB occurs, via SW.

More sophisticated switch elements are described in Chapter 14.

CHAPTER 13
STRINGS

13.1 GENERAL

DECsystem-10 ALGOL-60 includes a major extension to the string features defined in the Revised Report. Users wishing to run their programs on machines other than the DECsystem-10 should check whether the compiler they will use offers similar facilities. Scalar, array or procedure variables may be of type STRING, and are declared by the delimiter word STRING. The byte size, length and contents of string variables are defined via the various assignment statements described below.

Typical string declarations might be:-

```
STRING S,T; STRING ARRAY SA[1:10];
STRING PROCEDURE B(X); VALUE X; REAL X;
```

13.2 STRING EXPRESSIONS AND ASSIGNMENTS

String expressions are limited to a single string variable, a string procedure call or a string constant. (For a full description of string constants see Section 4.4.) The only string operators are the comparison operators and the assignment operator. All other operations are achieved via the string library procedures described in Section 13.7.

String expressions can be assigned only to string variables. For example:-

```
S:=T;
SA[I]:=SA[3];
SA[2]:=B(Z);
T:="ANY ""OLD"" IRON";
```

13.3 BYTE STRINGS

The value associated with a string variable is a byte string. A byte string is a sequence of bytes of a uniform size between one and thirty-six, which can be efficiently handled by the DECsystem-10 hardware. In some ways they can be thought of as arrays, but the most important difference between an array and a byte string is that the size of a byte string can vary continuously. Thus when a byte string is created by means of a READ statement, the programmer need not know how long the string will be. The routine starts accepting characters after it encounters the opening quotation marks and continues until it finds the closing ones.

When one string is assigned to another, e.g.,

```
S:=T;
```

then a copy of T is made to which S will refer. Any previous value S had is destroyed (and the heap space it occupied is released). Subsequent changes to the value of T will not affect S, or vice versa, unless a further assignment is made from one to the other.

Please note that this is an important change from the implementation of strings prior to Version 5.

13.4 BYTE SUBSCRIPTING

Byte strings can be modified by means of the byte subscripting mechanism. Individual bytes in a string are referenced by following the string variable name by a decimal point and then the subscript number enclosed in square brackets. For example

```
S.[I]
```

refers to the I'th byte of string S. The subscript may be any arithmetic expression and is evaluated in exactly the same way as an array subscript.

Byte-subscripted string variables are regarded as being of type integer, having an integer value equivalent to the byte to which they refer. It follows that to change the value of a particular

byte in a string, a byte-subscript must appear on the left-hand side of an arithmetic statement with the appropriate new value on the right-hand side. Should the new value be too large to be held in the byte, it is simply truncated. No warning is given.

13.5 NULL STRINGS

Until a value is assigned to a string by the program, it is assumed to have the value null. That is, it is assumed to contain no bytes. Any attempt to reference it by a byte-subscript will result in a fatal run-time error, though it can be used on the right-hand side of a string assignment, in which case the variable to which it is assigned similarly becomes null.

13.6 STRING COMPARISONS

Two byte strings can be compared with each other using the usual comparison operators. For example

```
IF S < T THEN GO TO L;
```

where S and T are string variables, string constants or string procedures. The effect of the comparison is to compare the strings byte-by-byte, the "lesser" string being that with the first lower value byte, working from left to right. Thus "ABCD" is less than "ABCE" or "ABCDE". Where the strings to be compared are of different byte sizes, then the smaller bytes are regarded as being extended on the left by null bits.

In the special case of ASCII strings (strings of byte size 7, like string constants), trailing nulls and trailing blanks, or any mixture thereof, are treated as equal. Similarly ASCII strings of different lengths will compare equally if the extra length comprises only spaces and nulls. In all other cases strings of unequal length can only be regarded as equal if the extra length is comprised entirely of null bytes.

13.7 LIBRARY PROCEDURES

Section 16.6 deals with the input and output procedures that are applicable to strings.

The procedures LINK, LINKR and TAIL that were included in the library until Version 3B have been dispensed with.

13.7.1 Concatenation

A string can be assigned the concatenated value of two strings with the string procedure CONCAT. For example

```
S:=CONCAT(T,U);
S:=CONCAT(S,T);
```

If, in the first example, T had a different byte size from U, then the size of the first string encountered (T in this case), would be adopted by S. The bytes copied from U would be truncated or filled with null bits as appropriate.

13.7.2 Length and Size

The primary attributes of a string, its length in bytes and byte size in bits, are returned by the integer LENGTH and SIZE, respectively.

Thus

```
I:=LENGTH(S); J:=SIZE(S);
```

would return the number of bytes in string S in I, and the number of bits in each byte in J.

In the case of a null string both procedures return zero.

13.7.3 Copying

The new byte string can be generated from an existing one by means of the string procedure COPY. This procedure can have one, two or three parameters.

a. The effect of copy with one parameter is precisely the same as a simple string assignment, but this feature has been retained for the sake of continuity.

b. Where there are two parameters, e.g.,

```
S:=COPY(T,M);
```

where M is an arithmetic expression, then S is assigned the value of the first through M'th bytes of T.

c. If there are three parameters, e.g.,

```
S:=COPY(T,M,N);
```

where both M and N are arithmetic expressions, then S is assigned the value of the M'th through N'th bytes of T.

13.7.4 Newstring

Although it is not possible to refer to bytes in a null string (see Section 13.5 above), it is possible to create a string of any appropriate byte size but containing nulls, by using this string procedure. NEWSTRING takes two parameters, the first being the number of null bytes to be assigned to the string, and the second their size.

For example

```
S:=NEWSTRING(100,7);
```

causes a null string of 100 ASCII nulls to be assigned to S. Notice that although S is 100 bytes long at this point, and thus byte subscripts up to and including S.[100] are valid, if a subsequent assignment of another string were made to S, both its length and its byte-size might change (depending, of course, on the length and byte-size of the other string!).

13.7.5 Delete

In Section 13.5 it was explained that if a null string is assigned to another string, then that string also becomes null, and its previous value is lost. Any space that the previous value occupied is returned to the heap, as with any ordinary string assignment. The typeless procedure DELETE has the same effect on the string passed to it as a parameter as the assignment of a null string would have. Deleting a null string has no effect, beyond using computer time.

CHAPTER 14
CONDITIONAL EXPRESSIONS AND STATEMENTS

14.1 GENERAL

ALGOL-60 allows great flexibility in the construction of expressions and conditions.

Consider, for example, if a user wanted to set a variable I equal to 0 or 1 according to the value of a Boolean variable B, he could write:

```
I := 0;
IF B THEN I := 1;
```

Also, consider the case where a user wants to perform some action, depending on the value of B:

```
IF B THEN X1 := Y; IF NOT B THEN X2 := Y;
```

14.2 CONDITIONAL OPERANDS

ALGOL-60 allows the user to substitute a conditional operand for any operand in an expression by the use of a construction involving IF ..... THEN ..... ELSE.

For instance, the first example above can be rewritten

```
I := IF B THEN 0 ELSE 1;
```

Clearly, this is more compact and of great use in cases such as:

```
J := J + (IF K < 1 THEN 1-K ELSE K-1);
```

Note that the conditional operand must be bracketed. It may be unbracketed only when it forms the complete expression itself.

In general, a conditional operand may replace an operand in any arithmetic or Boolean expression. It may also be used in place of a label as the element in a switch list, for example:

```
SWITCH SW := L1, IF B THEN L2 ELSE L3, L4;
```

It is also permitted, of course, in an array subscript (and also in a byte subscript), for example:

```
X := A[I, IF L = 0 THEN J ELSE J+1];
```

Since a conditional operand may replace any operand in an expression, it may also replace operands in conditional expressions. Consider the following example:

```
IF IF B THEN B1 ELSE B2 THEN I := I + 1;
```

This looks complicated but is really quite simple if brackets are inserted for clarity. Thus:

```
IF (IF B THEN B1 ELSE B2) THEN I := I + 1;
```

14.3 CONDITIONAL STATEMENTS

The reader was introduced to conditional statements of the form

```
IF B THEN S1 ELSE S2
```

in Chapter 7. The full power of this type of statement can now be demonstrated.

First, S1 and S2 can be compound statements or blocks. For example:

```
IF I < 0 THEN
BEGIN          I := -I; B := FALSE
END ELSE
BEGIN          I := I + 1; GOTO L2
END
```

Second, the whole structure of the IF ..... THEN ..... ELSE statement can be made more powerful by using conditional statements within themselves. For example:

```
IF X < 0 THEN X := 0 ELSE IF B THEN GOTO L
```

This is equivalent to the simple sequence of statements:

```
        IF NOT X < 0 THEN GOTO L1;
        X := 0; GOTO L2;
L1:     IF NOT B THEN GOTO L2;
        GOTO L;
L2:
```

Clearly the former method of expression is both briefer and more elegant.

Conditional statements take the general form

```
IF B THEN S1 ELSE S2
```

where S1 and S2 may themselves be conditional statements with the provision that if there is ambiguity, bracketing using BEGIN and END must be used to remove it. Consider the following illegal example:

```
IF B THEN IF X = 0 THEN Y := Z ELSE P := Q;
```

This could be interpreted as

```
IF B THEN BEGIN IF X = 0 THEN Y := Z END ELSE P := Q;
```

or

```
IF B THEN BEGIN IF X = 0 THEN Y := Z ELSE P := Q END;
```

The first case is interpreted as:

```
IF NOT B THEN GOTO L1;
IF NOT X = 0 THEN GOTO L2;
Y := Z; GOTO L2;
L1: P := Q;
L2:
```

The second case is interpreted as:

```
        IF NOT B THEN GOTO L2;
        IF NOT X = 0 THEN GOTO L1;
        Y := Z; GOTO L2;
L1:     P := Q;
L2:
```

ALGOL-60 forbids such ambiguities by forbidding the sequence THEN IF ..... THEN ..... ELSE.

14.4 DESIGNATIONAL EXPRESSIONS

A designational expression is something that acts as an argument in a GOTO statement, either directly or indirectly, via a formal procedure parameter of type label. It may simply be a label or a switch element. Thus the following are designational expressions:

```
L
IF B THEN L1 ELSE L2
IF X < 0 THEN SW[I] ELSE IF X+Y >= Z THEN TW[J] ELSE L
```

These designational expressions would be used in the following manner:

```
GOTO L;
GOTO IF B THEN L1 ELSE L2;
GOTO IF X < Ø THEN SW[I] ELSE IF X+Y >= Z THEN TW[J] ELSE L;
```

CHAPTER 15
OWN VARIABLES

15.1 GENERAL

Own variables are a special kind of ALGOL variable, and may be of type integer, real, long real, Boolean or string, either scalar or array. They have the following properties:

a. Although they follow the normal scope rules, they are not recursive, the same copy of each variable being used in all occurrences of a procedure or block.

b. The values they contain when control passes out of a block are retained and are still available when the block is re-entered.

c. They are initialized to zero before execution of the program. (FALSE in the case of Boolean own variables.) OWN STRINGS are initialized to possess no byte string.

Own variables are declared by writing the usual declaration with the word OWN preceding it. For example:

```
OWN INTEGER I,J,K;

OWN REAL ARRAY THETA[1:M]
```

15.2 OWN ARRAYS

Own arrays are implemented in a completely dynamic fashion in DECsystem-10 ALGOL. The declaration proceeds according to the following rules.

a. If this is the first time the array is declared, space is obtained and then the array laid out. If the array has been laid out before, proceed to Step b.

b. The bounds are examined to see if they are identical to those of the previous construction of this array. If they are the same, the array is left unaltered; otherwise, proceed to Step c.

c. A new array is constructed and those elements that it has in common, if any, with the old array are copied into it; the remaining elements are zeroed. The old array is then deleted and the space used by it is recovered for future use.

For example, if an own array A is declared as follows:

```
OWN REAL ARRAY AL[1:M,M:N];
```

where M = 2 and N = 5 the first time, and M = 1 and N = 4 the second time, the elements [1,2], [1,3] and [1,4] are copied over, and the remaining elements of the new array are zeroed.

CHAPTER 16
DATA TRANSMISSION

16.1 GENERAL

Data transmission encompasses the input and output of data between the user's program and peripheral devices, such as disk, DECtape, magnetic tape, card reader, card punch, and line printer. The DECsystem-10 ALGOL object-time system, in conjunction with the ALGOL library, provides the user with a set of basic procedures for handling data from most DECsystem-10 devices in a uniform fashion. The user may also perform input/output operations with virtual peripherals that manifest themselves as byte strings in the user's program.

All peripheral devices that the user requires are under his control completely and can be allocated or released at any time throughout the execution of the program. The user can handle up to sixteen devices simultaneously (seventeen, if one of them is the terminal attached to his job), any number of which may be file devices (disk, DECtape) and have an independent file open.

16.2 ALLOCATION OF PERIPHERAL DEVICES

Peripheral devices are allocated to the user's program by calls to the library procedures INPUT or OUTPUT. A call to one of these procedures usually has two parameters. The first is the channel number, an integer in the range 0 to 15, on which the device is to operate. Only one device at a time may be operated on a channel; a channel provides either input or output facilities, except in the case of a terminal, where the input and output functions are performed simultaneously on the same channel. The second parameter is either a string or a string constant. The text contained in the string is the logical name of the device to be allocated to this channel.

The DECsystem-10 Users Handbook should be consulted for an explanation of what constitutes a logical device name. In the simplest case, it may be the actual name of the peripheral device. The device names shown in Table 16-1 are recognized as standard.

Table 16-1
Standard Device Names

Device Name	Peripheral
DSK	Disk
DTA	DECtape
MTA	Magnetic tape
CDR	Card reader
CDP	Card punch
LPT	Line printer
PTR	Paper-tape reader
PTP	Paper-tape punch
PLT	Plotter
TTY	Terminal

For example, to allocate the card reader for use as an input device on channel 5, the user would use the statement

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

or, if S were a string possessing a byte string that had the characters CDR in it,

```
INPUT(5,S);
```

Similarly, if the disk were to be used as an output device on channel 9:

```
OUTPUT(9,"DSK");
```

Note that with the exception of terminals, all devices are allocated to operate in one direction only; thus, if the user wants input and output from the disk, he must use two separate channels.

Terminals are always allocated bidirectionally, irrespective of whether the user uses INPUT or OUTPUT. For example,

```
INPUT(0,"TTY");
```

allocates the user's terminal for input and output on channel 0.

16.2.1 Device Modes

Normally, a device is allocated in ASCII mode, that is, when the user reads a character from the device it is a 7-bit byte representing readable text, such as a stored source program or data. To allocate the device in a different mode, a third parameter is specified in the call to the INPUT or OUTPUT procedure. Thus, to allocate a disk to channel 9 in image binary mode (the mode used for the storage of binary data on a disk), the user can use

```
OUTPUT(9,"DSK",11);
```

The DECsystem-10 Assembly Language Handbook should be consulted for a full explanation of the different modes used with peripheral devices. The INPUT and OUTPUT procedures allow the user to allocate any standard peripheral device in any buffered mode.

16.2.2 Buffering

The INPUT and OUTPUT procedures normally allocate two buffers for each allocated device; terminals are allocated two buffers for input and two for output. The user may desire to use either one or more than one buffer for a device. For example, in a non-compute bound job that uses a lot of disk transfers at odd intervals, four or even eight buffers may be desirable to increase the speed of execution of the program.

The number of buffers to be used can be controlled by adding a fourth parameter to the procedure call. Thus, to allocate a disk on channel 14 in mode 0 with eight buffers, the call is

```
OUTPUT(14,"DSK",0,8);
```

Note that the mode must always be specified in this case, otherwise there would be an ambiguity in the third parameter.

16.2.3 Error Returns

Normally, if the device allocation fails (for example if the device is in use by another job), a suitable message is typed and the program terminated. The user can prevent this by providing, as the fifth parameter to INPUT or OUTPUT, a label to which control is to be passed in the event of an error. Note that the third and fourth parameters must be present in this case to avoid ambiguity: they may be zero when the default values will be used. For example:

```
OUTPUT(14,"MTA",0,0,ERROR.LABEL);
```

If the actual label parameter is a switch whose subscript is out of range, the procedures behave as though the label parameter were absent.

16.3 SELECTING INPUT/OUTPUT CHANNELS

Before a user uses a device to transfer data, assuming that the device has already been allocated to some channel, the appropriate input or output channel must be "selected" for use as the input or output channel. All data input and output always occurs on the currently selected input channel and output channel, respectively. The user may change the selection of channels at any time, switching from one channel to another without loss of data, irrespective of whether

complete lines (or records) of data have been read or not. In fact, the DECsystem-10 input/output system does not assume any structure in the data: all input and output channels are regarded as pipelines through which the user pulls or pushes data.

To select an input channel, a call to the procedure SELECTINPUT must be made. This has one parameter, which is the channel number. Thus

```
SELECTINPUT(5);
```

causes input channel 5 to be selected.

Similarly, the procedure SELECTOUTPUT is used to select an output channel.

16.4 FILE DEVICES

Some peripheral devices, such as disk and DECtape, require the opening of a specifically named file before any input or output operations can be performed. This optionally may be performed on spooled devices (refer to Chapter 3 of DECsystem Operating System Commands for a description of spooling). The opening of this file is performed by means of the procedure OPENFILE, which is called after the device has been allocated to a channel. The procedure call has two parameters: the channel number on which the device has been allocated and a string variable possessing a byte string or a string constant, the text of which is the name of the file.

The user can also specify a protection and/or project-programmer number of a file by means of optional third and fourth Boolean or integer parameters. For example, to open a file with protection 177 on disk area [11,50] the user could write

```
OPENFILE (9,"TEST.DAT",%177,%000011000050);
```

When a user has finished with a file it should be closed. A file is closed by using the procedure CLOSEFILE, with a parameter that is the channel number on which the file is open. Thus,

```
CLOSEFILE(9);
```

closes the file that is open on channel 9.

The user may also rename or delete existing files: if a file is already open, use of OPENFILE causes the file to be renamed with the new name supplied. Thus the sequence

```
OPENFILE (5,"TEST1.DAT");
OPENFILE (5,"TEST2.DAT");
```

causes the file with name TEST1.DAT to be renamed TEST2.DAT.

If the string containing the new name is null, the original file is deleted. Thus,

```
OPENFILE (5,"TEST3.DAT");
OPENFILE (5,"");
```

causes the file TEST3.DAT to be deleted.

16.4.1 Error Returns

Normally, if the operation requested fails (for example an input file does not exist), a suitable message is typed and the program terminated. The user can prevent this by providing a label and an optional integer variable as the fifth and sixth parameters to OPENFILE. In the event of an error, control will be passed to the label, with an error-code set into the integer variable if present. The error-codes are those returned by the ENTER and LOOKUP UUO'S: refer to Appendix E of the DECsystem-10 Monitor Calls Manual.

Note that the third and fourth parameters must be present if the error return parameter is specified: they may be zero in which case the default values will be used.

If the actual label parameter is a switch whose subscript is out of range, the procedure behaves as though the label parameter were absent. The integer error-code parameter is called by name.

16.5 RELEASING DEVICES

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

```
RELEASE(5);
```

releases the device allocated to channel 5. If the device is a file device, and a file is still open on the device, it is automatically closed. Releasing a device on a channel causes a channel to become free; if this channel is currently selected for input or output operations, it is deselected.

If an attempt is made to allocate a device to a channel that already has a device allocated, the allocated device is first released and, if a file is open on it, it is closed before releasing the device.

If a user terminates his program without releasing devices on channels, they are automatically released.

16.6 BASIC INPUT/OUTPUT PROCEDURES

16.6.1 Byte Processing Procedures

The following procedures may be used with any device to handle bytes of any standard size (1 to 36 bits). However, because they are normally used with devices supplying or receiving ASCII bytes, they are "symbol" oriented.

a. INSYMBOL(S); - (where S is usually some integer variable) causes the next byte to be read from the currently selected input channel and stored in S.

b. OUTSYMBOL(J); - (where J is usually some integer expression) causes the value of J to be output as a byte to the currently selected output channel. If J is too large for the byte size of the device in use, it is truncated to size.

c. NEXTSYMBOL(S); - acts in exactly the same way as INSYMBOL except that the byte pointer for the input channel is not advanced to the next available byte. This gives the user a look-ahead facility of one byte.

d. SKIPSYMBOL; - causes the next byte from the selected input channel to be read and ignored.

e. BREAKOUTPUT; - causes all bytes in the buffer of a device to be sent immediately to it. This procedure is normally used to conduct a question-and-answer dialogue on a terminal, with the question and answer on the same line. Normally, a block of data is sent to a device only when the buffer is full (the exception being the terminal, where a break is sent at the end of each line).

16.6.2 String Output

A byte string may have its contents transferred to the currently selected output channel by means of the procedure WRITE, whose single parameter is either a string constant or a string variable that possesses the string to be output. For example:

```
WRITE(S);
```

or

```
WRITE("THE MOON IS MADE OF GREEN CHEESE");
```

With exceptions explained in the following paragraphs, all of the bytes in the string are output literally, with the exception, of course, of the quotes in a string constant, which are not in fact stored in the byte string at all. The important thing to remember is that, unlike some other ALGOL implementations, spaces and other non-printing symbols in byte strings are meaningful.

Special editing characters are permitted within square brackets within the text of a byte string. These have a special function:

P	Page throw
C or N	New line (C stands for carriage return, line feed)
T	Tab
S	Space
B	Break output

Any combination of these characters, with optional repetition counts preceding them, can appear within square brackets in a byte string and are output as their special interpretation demands. For example:

```
WRITE("ABCD[P2C5S]EFGH");
```

causes the following to be output:

a. the symbols ABCD followed by a page throw
b. two new lines and five spaces
c. the symbols EFGH.

In order to output the symbols

```
[ ] " or ;
```

they must appear in the form

```
[[ ]] "" or ;;
```

respectively. Thus

```
WRITE("""A[[I]] := 3;;""");
```

causes the text

```
"A[I] := 3;"
```

to be output.

16.6.3 Miscellaneous Symbol Procedures

The procedures SPACE, TAB, PAGE, and NEWLINE cause the appropriate number of spaces, tabs, page throws, or new lines to be output, depending on their single parameter, which is an integer expression. If the parameter is omitted a value of one is assumed. Thus

```
SPACE(5);
```

causes five spaces to be output, whereas

```
SPACE;
```

or

```
SPACE(1);
```

causes one space to be output.

16.6.4 Numeric and String Procedures

Numeric procedures are used to read and print numeric quantities. They will normally be used with a device that is operating in ASCII mode. They are capable of processing integer, real, or long real quantities in fixed-point and floating-point representations.

16.6.4.1 Numeric Input Data - Numeric data for input can be represented in any format that would be acceptable as a numeric constant in a program, irrespective of the type of variable involved. When a number is read, an automatic type conversion is performed, giving a result of the same type as if an assignment of the data represented as a constant in the program had been executed.

There is a minor restriction in that no spaces, tabs, or other non-printing symbols may appear in such numeric data except between the exponent sign (& or @ for real, && or @@ for long real) and the exponent. Otherwise, any symbol that is not itself a part of a numeric quantity may act as a terminator for such a quantity. It is strongly recommended that spaces, tabs, or new lines be used as separators. For example:

```
3.4 -9.6 1.36 -52
0 14.9
```

Note that in reading a numeric quantity, the terminating symbol, that is, the first symbol that is not part of the number, is lost.

DECsystem-10 ALGOL also allows the user to input floating-point data written in FORTRAN format, that is, using E for & or @, and D for && or @@. Note, however, that no other special effects inherent in FORTRAN formatting are introduced.

The procedure READ is used to input numeric data and also strings. This procedure may have any number of parameters, of type integer, real, long real, Boolean, or string.

The effect is as follows:

a. For integer, real and long real variables, a number is read and converted to the type appropriate to the parameter and then assigned to the variable.

b. For Boolean, a number is read as if for an integer variable, and assigned to the variable.

c. For a string variable, the data text is scanned until a quote (") is found, and the text following this up to but not including the next free quote is read in and a byte string generated, which is then possessed by the string variable.

If the sequence "" is found, a single " is stored, and reading of the string continues.

16.6.4.2 Numeric Output Data - Numeric data is output by means of the procedure PRINT. This procedure may have one, two, or three parameters, the first of which is the variable to be printed. This variable may be an integer, real, or long real. The second and third parameters determine the format to be used and are integer expressions. If they are omitted, they are assumed to be zero. The effect of the various combinations of the format integers, M and N, is as follows:

M > 0, N > 0:	Fixed-point printing, M places before the decimal point, N places after. A sign, space if positive, - if negative appears before the number. Zeros before the decimal point are replaced by spaces and the sign moved up to the number. This format always outputs M+N+2 symbols.
M > 0, N = 0:	The same as the preceding except that (a) no fractional part appears, and (b) the decimal point is suppressed. This format always outputs M+1 symbols.
M = 0, N > 0:	Floating-point format, consisting of a sign, a decimal digit, a decimal point, N more decimal digits, and an exponent consisting of & for real, && for long real followed by the exponent sign and a two-digit exponent, zero suppressed from the left. This format outputs N+7 symbols for real and N+8 symbols for long real quantities.

If only two parameters appear, format M,0 is assumed for integer variables, and format 0,N for real and long real quantities, where M and N are, respectively, the values of the second parameter.

If only one parameter appears, the format is interpreted as 0,0 which assumes standard printing modes of 11,0 for integer quantities, 0,9 for real quantities, and 0,17 for long real quantities.

If the user requests more digits to be printed than are significant in real or long real numbers, the appropriate number of zeros follow a properly rounded printing of the number to the maximum precision available.

16.6.4.3 Octal Input/Output - The procedures READOCTAL and PRINTOCTAL, respectively, allow the user to input and output quantities in octal format.

On input, for single precision variables, up to 12 octal digits are read, preceded by the symbol %, the terminator being any non-numeric symbol. For long real variables, two such octal numbers must be presented for input, each preceded by the symbol %.

On output, 12 octal digits, preceded by the symbol %, are printed for single precision variables. For long real variables, two quantities each with 12 octal digits are printed, with a space separating them.

The foregoing procedures have one scalar parameter which may be of type integer, real, long real or Boolean.

16.7 DEFAULT INPUT/OUTPUT

If the user does not select any input or output channels, input and output occur via an "invisible" channel from and to the user's terminal. Thus, for simple programs where the user wishes to input a few numbers and print a few results, he simply uses READ, types in the data on line through his terminal, and gets back the results from PRINT.

16.8 LOGICAL INPUT/OUTPUT

In addition to the 16 channels used to communicate with peripheral devices, an additional 16 channels, numbered from 16 to 31, are provided. These are input or output channels that use byte strings as a means of storage.

By means of the procedures INPUT or OUTPUT, the user can attach a channel to a byte string possessed by a string variable, and can read and write bytes from and to this byte string, either to or from a peripheral device, or to and from another byte string.

```
INPUT(20,S);
```

or

```
OUTPUT(20,S);
```

cause the byte string possessed by the string variable S to be used as logical channel 20; this channel may subsequently be selected for input or output, as appropriate.

The user is still free, of course, to manipulate the individual bytes within the byte string by means of the byte-subscripting facilities available. Such facilities enable the user to read a file from a peripheral device into a string, process it in any way whatsoever, and output it again.

16.9 SPECIAL OPERATIONS

These procedures are used on channels assigned to magnetic tapes. They consist of the procedures BACKSPACE, ENDFILE and REWIND, each having one parameter, i.e., the channel number on which the operation is to be performed.

Since there is no implicit structure on a magnetic tape, these procedures enable the user to build up formats in any way he chooses.

16.10 I/O CHANNEL STATUS

The status of any input or output channel can be determined at any time by means of the Boolean procedure IOCHAN, which takes as its single parameter an integer quantity which is the channel number. The status returned is bit coded as follows:

Bit	Value	Meaning if Set
18	%400000	Device is physical (i.e., not logical)
19	%200000	Directory device
20	%100000	Terminal device
21	%040000	ASCII mode
22	%020000	Magnetic tape
23	%010000	Plotter
24	%004000	Set for default TTY on channel -1
25	%002000	Device is spooled
26	%001000	Device can do input
27	%000400	Device is initialized for input
28	%000200	File is open for input
29	%000100	End of file encountered
30	%000040	Input ok status
31	%000020	Device can do output
32	%000010	Device is initialized for output
33	%000004	File is open for output
34	%000002	Device quota (exceeded)
35	%000001	Output status ok

Some of these bits are of little use to the user, but, for example, if a device is allocated, and the user does not know whether or not it is a file device, he can use IOCHAN to determine this. The bits of particular use to the user are the input and output end-of-file (note that end-of-file on output is a logical status indicating that, for example, a disk quota is exceeded or a DECtape is full, or in the case of a logical device, the byte string is full).

When IOCHAN is used, the end-of-file flags are always cleared, if set, so that the user may proceed to read a magnetic tape after an end-of-file marker is found.

The following example shows how the user would handle an unknown device whose name is given to the program via the user's terminal:

```
BEGIN
    STRING DEVICE, FILE; INTEGER CHANNEL;
    WRITE ("CHANNEL NO: "); BREAK.OUTPUT;
    READ (CHANNEL);
    WRITE ("[C]DEVICE NAME: "); BREAK.OUTPUT;
    READ (DEVICE);
    OUTPUT (CHANNEL, DEVICE);
    IF IOCHAN (CHANNEL) AND %200000 THEN
      BEGIN
        WRITE ("[C]FILE NAME: "); BREAK.OUTPUT;
        READ (FILE);
        OPENFILE (CHANNEL, FILE)
      END;
          .
          .
          .
END
```

NOTE

When using Boolean expressions involving IOCHAN, the rules for evaluation followed in this implementation should be borne in mind. See Section 5.2.1.

16.11 TRANSFERRING FILES

Once a device has been allocated to an input or an output channel, a complete file of information may be transferred between them automatically by calling the parameter-less procedure TRANSFILE. This procedure copies bytes from one device to another from the currently selected input channel to the currently selected output channel, until an end-of-file status is raised on either the input or output channel.

16.12 CURRENTLY SELECTED CHANNEL NUMBERS

The number of the channel currently selected for input or output may be obtained by use of the integer procedures INCHAN or OUTCHAN.

CHAPTER 17
THE DECsystem-10 OPERATING ENVIRONMENT

The operating environment of DECsystem-10 ALGOL programs consists of those procedures in the DECsystem-10 ALGOL Library required by the user's program, and the DECsystem-10 ALGOL Object Time System.

The former are those procedures detailed in Chapters 13 and 16, together with those described below. These procedures can be thought of as existing in a block surrounding the user's program, and, therefore, are available when called. Their names, however, are in no sense reserved as are words such as BEGIN.

Note also that these procedures are only present in the user's program when required. They are loaded by the DECsystem-10 Linking Loader when so directed by the DECsystem-10 ALGOL Compiler. The user is not required to take any action to include these procedures, other than make a call to them. A complete list of library procedures is given below.

17.1 MATHEMATICAL PROCEDURES

The following procedures expect one argument, of real type, and yield a real type result.

Procedure Name	Function
SIN	Sine
COS	Cosine
ARCTAN	Arctangent
SQRT	Square root
EXP	Exponential
LN	Logarithm (to base e)
TAN	Tangent
ARCSIN	Arcsine
ARCCOS	Arccosine
SINH	Sinh
COSH	Cosh
TANH	Tanh

Note that if arguments of type integer or long real are given in an ALGOL call to these procedures, the compiler plants the appropriate conversion code.

The following procedures expect one argument, of long real type, and yield a long real type result. Note that theyare formed by adding an L before the equivalent single precision procedure.

Procedure Name	Function
LSIN	Sine
LCOS	Cosine
LARCTAN	Arctangent
LSQRT	Square root
LEXP	Exponential
LLN	Logarithm (to base e)

The functions ENTIER, ABS and SIGN are also available, as described in Section 5.1.2

17.2 STRING PROCEDURES

For details of the procedures CONCAT, LENGTH, SIZE, COPY, NEWSTRING and DELETE, see Paragraph 13.7.

17.3 UTILITY PROCEDURES

17.3.1 Array Dimension Procedures

The integer procedure DIM, which takes as its parameter the name of an array of any type, yields a result that is the number of dimensions of the array. This is most useful when the user passes an array as a parameter and wishes to check if it is, for example, a matrix.

The integer procedures LB and UB also take as first parameters the name of an array; their second parameter is the subscript number. The result is the lower or upper bound, respectively, of the subscript specified by the second parameter. The following procedure uses these to clear real matrices.

```
PROCEDURE ZERO(A); ARRAY A;
BEGIN
  INTEGER I,J;
  IF DIM(A) = 2 THEN
  BEGIN
    INTEGER L1,L2,U1,U2;
    L1 := LB(A,1); U1 := UB(A,1);
    L2 := LB(A,2); U2 := UB(A,2);
    FOR I := L1 UNTIL U1 DO
      FOR J := L2 UNTIL U2 DO A[I,J] := 0
  END
END
```

17.3.2 Minima and Maxima Procedures

The integer procedures IMIN and IMAX, the real procedures RMIN and RMAX, and the long real procedures LMIN and LMAX are used, respectively, to determine the minimum or maximum of a number of arguments of the appropriate type. These procedures normally accept up to ten parameters (this figure may be changed by re-assembling the ALGOL library with a different parameter.

For example:

```
I := IMIN(J,K);
X := RMAX(Y+Z,RMIN(Y-Z,0));
```

17.3.3 Field Manipulations

The procedures GFIELD and SFIELD enable the user to manipulate a field within any integer, real, long real, Boolean or string variable. The integer parameters I and J specify a byte of length J bits whose leftmost bit is the I'th bit (counting from zero at the left-hand side). The byte specified may be from 1 to 36 bits in length and may be at any position in the variable.

For single word variables (integer, real, Boolean), I may range from 0 to 35, with the constraint I + J <= 36. For double word variables (long real and string), I may range from 0 to 71, with the constraint I + J <= 72.

The integer procedure GFIELD uses I and J as the second and third parameters; the first parameter is the variable. The result is the value of the byte (right justified) specified by I, J.

Thus

```
K := GFIELD(A,3,5);
```

gives the value of the byte consisting of bits 3 through 7 of A.

The procedure SFIELD sets a byte specified by the second and third parameters I, J to the value specified by the fourth parameter, of type integer. Thus

```
SFIELD(A,3,5,0);
```

zeros the byte specified in the first example.

17.4 DATA TRANSMISSION PROCEDURES

For details of these procedures refer to Chapter 16.

17.5 FORTRAN INTERFACE PROCEDURES

F-10 or F-40 FORTRAN subroutines may be incorporated in ALGOL object programs by loading these subroutines with the ALGOL main program (and any other separate ALGOL procedures).

Such FORTRAN subroutines should be specified by an EXTERNAL declaration in the ALGOL program and, depending on which compiler was used to compile them, are called by the procedures shown in the table.

Table 17-1

TYPE / FORTRAN	NONTYPE	INTEGER	REAL	LONG REAL (DOUBLE PR.)	BOOLEAN (LOGICAL)
F-10	F1ØCALL	F1ØICALL	F1ØRCALL	F1ØDCALL	F1ØLCALL
F-40	CALL	ICALL	RCALL	DCALL	LCALL

The first parameter in these procedure calls must be the name of the FORTRAN subroutine. Subsequent parameters are taken as the arguments to the procedures.

CALL and F1ØCALL are used as single statements, for example:

```
CALL (FORT,X,Y)
```

is equivalent to

```
CALL FORT (X,Y)
```

in a FORTRAN program.

ICALL etc. must appear in the appropriate context in an expression, thus

```
P := Q + ICALL(Z)
```

NOTE

The parameters of CALL, ICALL, etc., are restricted to integer, real, long real, or Boolean expressions with the restriction that if the expression is a single variable, it must be a local scalar or a format parameter called by value.

17.6 GENERAL INFORMATION ROUTINE

The integer procedure INFO provides information about various aspects of the environment, depending on the value of the parameter with which it is called.

parameter	returns integer value of
0	core size in words
1	date (15-bit format)
2	time (ticks since midnight)
3	time (micro-seconds since midnight)
4	runtime (milliseconds)
5	processor type (1=KA, 2=KI, 3=KL)
6	number of stack shifts so far
7	compiler version word

For example

```
PRINT(INFO(4));
```

might produce

```
1134600
```

- the job's runtime up to now in milliseconds.

17.7 DATE AND TIME IN ASCII FORMAT

Three routines are provided for returning the current time and date in string format suitable for printing without modification. The two date routines, FDATE and VDATE, give the option of a standard three-character abbreviation for the month (FDATE), or a variable-length string with the name of the month in full (VDATE). In both cases the year is given in full. String procedure TIME gives an eight-character string with the current time as HH:MM:SS.

For example

```
WRITE(VDATE); NEWLINE; WRITE(TIME);
```

might produce

```
27-JANUARY-1975
12:16:55
```

17.8 RANDOM NUMBER ROUTINES

Three routines have been included to provide a random number capability. The number generator is similar to that used in the FORTRAN library, which is documented in the Science Library manual. The ALGOL version is, however, initialized randomly. If it is required to generate a

sequence of pseudo-random numbers in a repeatable way, then procedure SETRAN should be called before the first reference to RAND, with the required initial value. For example, to generate the same sequence as a FORTRAN program using the default starting value currently used by FORTRAN, SETRAN(-1); should be included in the ALGOL program. The third procedure is SAVRAN which returns the value of the last random number without invoking the number generator.

RAND and SAVRAN are INTEGER procedures; SETRAN is non-type.

CHAPTER 18
RUNNING AND DEBUGGING PROGRAMS

18.1 COMPILATION OF ALGOL PROGRAMS

DECsystem-10 ALGOL programs are compiled by the ALGOL compiler under the standard DECsystem-10 timesharing monitor. The compiler is called by typing

```
R ALGOL
```

at monitor command level.

The DECsystem-10 ALGOL Compiler responds by typing an asterisk on the user's terminal. The user then types a command string to the compiler, specifying the source file(s) from which the program is to be compiled, and the output files for listing and output of relocatable binary. The command string takes the form:

```
OUTPUT-FILE,LISTING-FILE=SOURCE-FILES
```

followed by a carriage-return (ALTMODE cannot be used to terminate a command string).
A file takes one of the forms

```
DEVICE:FILE-NAME.FILE-EXTENSION
```

or

```
DEVICE:FILE.NAME
```

for directory devices (disk and DECtape)

or

```
FILE-NAME.FILE-EXTENSION
```

or

```
FILE-NAME
```

where DSK is assumed to be a default device.

In the case of non-directory devices, the format is simply

DEVICE:

In cases where no FILE-EXTENSIONS are specified, the standard defaults REL for the relocatable binary output file, LST for the listing file, and ALG for the source file are assumed.

SOURCE-FILES

consists of one file or a list of files separated by commas. If a DEVICE is specified for the first file, and not for succeeding files, the second and following files are taken from the same device as the first.

Example:

```
EULER,TTY:=EULER
```

[read source from DSK:EULER.ALG, write relocatable binary on DSK:EULER.REL and listing on the user's terminal].

```
MTA0:,DSK:SIM26=SIM26,PARAM.TST
```

[read source from DSK:SIM26.ALG, DSK:PARAM.TST, write relocatable binary on device MTA0, and listing on file DSK:SIM26.LST].

Certain switches may be set by the user in the command string. These are:

BUFFERS:n	Set number of buffers in compiler's I/O buffer-ring to n.
CHECKON	Compile run-time array-bound checking (see 18.5.1.1).
CHECKOFF	Do not compile run-time array-bound checking, regardless of any CHECKON statements in the source.
HEAP:n	Set the initial size of the dynamic core area (which is used for I/O buffers, strings and OWN arrays) to n words. This area is dynamically expanded at run-time if necessary; its final size is typed out at the end of execution if the object program is loaded with DDT.
HELP	*Type helpful text.
KA10	Produce code to run on the KA10 processor.
KI10	Produce code to run on the KI10 processor.
KL10	Produce code to run on the KL10 processor.
LIST	*List the source program (default if a listing-device is specified in the command-string).
NOERRORS	Do not type error-messages on the terminal.
NOLIST	Do not list the source program.
NONUMBERS	The source program does not have line sequence numbers in columns 73 to 80 (default).
NOQUOTES	Delimiter words are not in quotes (default).

NUMBERS	The source program has line sequence numbers in columns 73 to 80.
PRODUCTION	Do not compile trace information or output expanded symbol-table to the .REL file (not yet implemented).
QUOTED	*Delimiter words are in quotes.
TEMPCODE:n	Set length of TEMPCODE area in compiler to n words: this is only necessary if the compiler produces a message to that effect, which may happen for some very complicated statement constructs.
TRACE	Control tracing. (Not yet implemented.)

Switches may be shortened to a unique abbreviation: those marked with an asterisk (*) in the above list may also be given as a single character. Values (n) are in decimal.

These switches are set by preceding them with a / after a file-specification, for example:

```
PROD,PROD=PROD1/L,PROD2/NOL/HEAP:2000
```

causes file PROD1 to be compiled with a listing, PROD2 to be compiled without listing, and the initial size of the run-time dynamic core area to be set to 2000 words (the default size is 521 words).

The ALGOL compiler reports all source program errors both on the user's terminal and in the listing device (if it is other than the terminal). After compiling a program, the compiler returns with another asterisk, whereupon the user may compile another program, or type ↑C to return to monitor level.

18.1.1 Compilation of Free-Standing Procedures

DECsystem-10 ALGOL allows the user to compile procedures independent of programs that call them. Such procedures may either follow the main program in the source file (but may not appear before it), or may be in an independent source file either singly or together. The user uses exactly the same process to compile such files.

If the user requires to call those procedures from the main ALGOL program, the appropriate EXTERNAL declarations must be made (refer to Paragraph 11.9).

18.2 LOADING ALGOL PROGRAMS

ALGOL programs are loaded by means of the DECsystem-10 Linking Loader in exactly the same way as programs generated by MACRO-10 and FORTRAN (for details, refer to the DECsystem-10 Assembly Language Handbook).

LINK-10 automatically causes all procedures required from the ALGOL Library (ALGLIB) to be incorporated into the user's program.

For example, consider the source file MAIN.ALG which contains the ALGOL main program and the files SUB1.ALG and SUB2.ALG which contain free-standing procedures.

The user may compile these files to give one relocatable binary file by typing the following command string to the ALGOL compiler,

```
MAIN,MAIN=MAIN,SUB1,SUB2
```

and loading the resulting program by giving the command string

```
MAIN/GO
```

to LINK-10. Alternatively, the three source files can be compiled independently by typing three command strings to the ALGOL compiler, for example:

```
MAIN,MAIN=MAIN

SUB1,SUB1=SUB1

SUB2,SUB2=SUB2
```

and giving LINK-10 the command string.

```
MAIN,SUB1,SUB2/G
```

After a program has been loaded, it may be executed.

18.3 RUNNING ALGOL PROGRAMS

ALGOL programs are executed by typing the console command

```
START
```

or any of its valid abbreviations. If the program executes successfully, it finishes by printing the execution time statistics (core store used and execution and elapsed times) on the user's terminal, and returns to monitor command level.

18.4 CONCISE COMMAND LANGUAGE

The concise command language (CCL) features in the DECsystem-10 monitor may be used to facilitate the compilation and execution of ALGOL programs. They are used in exactly the same way as for programs written in DECsystem-10 FORTRAN. For details, refer to the DECsystem-10 Users Handbook.

Switches to the ALGOL compiler are enclosed in parentheses and separated by slashes; for example:

```
EXECUTE          FOO(QUOTED/HEAP:2000/KI)
```

18.5 RUN-TIME DIAGNOSTICS AND DEBUGGING

If a run-time error occurs during the execution of an ALGOL object program, an error message is produced, detailing the type of error, and its address within the user's program. Such errors fall into two categories - fatal and non-fatal.

A mechanism has been provided by which the user can trap non-fatal errors, and, when they occur, transfer control to a label within the user's program. Each such error has a unique number, and a table of these appears below. The Library procedure TRAP, used to trap non-fatal error has the following specification:

```
PROCEDURE TRAP (N,L); VALUE N,L;
  INTEGER N; LABEL L;
```

Where N is the number of the error to be trapped, and L is a label to which control is required to be passed when the error occurs.

Once such a trap is set up by a call to TRAP, it remains in force until another call to TRAP sets a trap to a different label, or until the trap is turned off by

TRAP (N)

i.e., omitting the label parameter in a trap call.

Note that the trap label is a formal parameter by value.

Table 18-1
Error Trap Numbers

TRAP NO.	ERROR
18	FLOATING POINT OVERFLOW
19	FIXED POINT OVERFLOW
32	INPUT OR OUTPUT DEVICE UNAVAILABLE
33	ILLEGAL MODE FOR INPUT OR OUTPUT DEVICE
34	INPUT OR OUTPUT ON UNDEFINED CHANNEL
35	ATTEMPT TO READ OR WRITE ON DIRECTORY DEVICE WITHOUT FILE OPEN
37	FILE NOT AVAILABLE OR RENAME FAILURE
38	ATTEMPT TO READ OR WRITE OVER END-OF-FILE
39	ERROR CONDITION ON INPUT OR OUTPUT
40	ILLEGAL CHARACTER IN NUMERIC DATA
41	OVERFLOW IN NUMERIC DATA
42	ERROR CONDITION ON CLOSING FILE
43	ILLEGAL INPUT-OUTPUT OPERATION
44	I-O CHANNEL NUMBER OUT OF RANGE
48	SQRT ARGUMENT NEGATIVE
49	LN ARGUMENT ZERO OR NEGATIVE
50	EXP ARGUMENT TOO LARGE
51	INVERSE MATHS FUNCTION ARGUMENT OUT OF RANGE
52	TAN ARGUMENT TOO LARGE

18.5.1 Facilities to Aid in Program Debugging

18.5.1.1 Checking - The directive

CHECKON

when placed anywhere in a user's program causes all array subscripts from this point onward in the program to be checked at run-time for being in range. The directive

CHECKOFF

nullifies this action. Note that use of this facility causes the generated program to be slightly larger, and to run slower.

The compiler switches /CHECKON and /CHECKOFF may also be used: they override any CHECKON or CHECKOFF statements in the source program.

NOTE

Most inexplicable errors arising during the execution of an ALGOL program are caused by an array subscript being out of range. Whenever such errors occur, the program should be recompiled with the array bound check feature on, and rerun.

18.5.1.2 Controlling Listing of the Source Program - Normally, a listing of the source program is output with the object program during compilation. The user can suppress this listing entirely by means of the /NOLIST compiler switch. However, if the user wishes to suppress only part of the listing and then continue listing, he can control the listing from within his program by means of the statements

```
LISTOFF
LISTON
```

The LISTOFF statement causes listing to be suppressed from the point in the program where LISTOFF was encountered to either the end of the program or until a LISTON statement is encountered. The LISTON statement causes listing to continue after it had been suppressed by a LISTOFF statement. The LISTON and LISTOFF statements have no effect if the /NOLIST switch is included in the compiler command string.

18.5.1.3 Setting Line Numbers in Listings - Ordinarily, the lines in the listing file are numbered sequentially starting at 1 and incrementing by 1. The user can, however, change the line numbers by placing sequence numbers in columns 73 through 80 of the source program and compiling with the /NUMBERS switch. Another way in which the user can change the line numbers is by means of the LINE statement. The statement

```
LINE n
```

causes the next line number to be set to n, which is a decimal integer. The line numbers that follow are incremented by 1 until either another LINE statement is encountered or the program terminates.

18.6 CROSS REFERENCE LISTING

The /CREF switch has been implemented, causing ALGOL to generate output for CREF. Output takes the following formats:

Variables and Labels: each occurrence of a variable or label name is recorded, with a # whenever a defining reference is made. In the case of labels the line reference is to the line which causes code for the label to be generated, which may follow the line where the label itself appears. No distinction is made amongst different incarnations of a variable at various block levels.

Blocks: the messages "START OF BLOCK n" and "END OF BLOCK n" are suppressed in the CREF listing. However, there is a separate CREF of blocks on the first page following the program. As with labels, the line-numbers refer to those causing code to be generated, not necessarily those on which BEGIN or END appear in the source.

18.7 STACK ANALYSIS

The stack analysis takes two forms - if a program stops with an error, the stack is scanned to give the names of the active procedures. Thus a typical error message now reads:

```
?RUN-TIME ERROR AT ADDRESS 000162


IN PROCEDURE OPENFILE
CALLED FROM PROCEDURE PQRSTUVWXYZ
CALLED FROM PROCEDURE THISPROCEDUREISNEXTTOINNERMOST
CALLED FROM PROCEDURE THISONEISNEARLYTHEOUTERMOST
CALLED FROM PROCEDURE THISISTHEOUTERMOSTPROCEDURE
CALLED FROM MAIN PROGRAM
FILE DSK:NOSUCH.FLE NOT AVAILABLE OR RENAME FAILURE ON CHANNEL #  1
```

The above type of analysis is automatic whenever an error is detected; the second type is invoked by a REEnter command after program execution, followed by typing P (for Profile). This causes a list of all the procedures, labels and library procedures referenced in the program to be printed, together with a count of the actual number of times each was referenced (or passed through in the case of labels). As with the trace print, labels can be distinguished by a trailing colon (:) and library procedures by a trailing asterisk (*), but different procedures with the same name must be identified from the order in which they appear, which is the same as the order in which they were loaded.

Note that in the special case of overlaid programs only the root segment is scanned.

```
ACTION (H FOR HELP)? P
                     -


PROFILE PRINT.

COUNT    NAME
-----    ----

  1      PQRSTUVWXYZ
  1      THISPRROCEDUREISNEXTTOINNERMOST
  1      THISONEISNEARLYTHEOUTERMOST
  1      THISISTHEOUTERMOSTPROCEDURE
  0      READ*
  1      INPUT*
  0      OUTPUT*
  1      OPENFILE*
```

18.8 TRACE

18.8.1 Dynamic Trace

Two types of Trace are available: dynamic and post-mortem.

The user who is at a terminal and wishes to see a full dynamic trace of his program should type the following command sequence:

```
.LOAD TRTST
 ---- -----
ALGOL:  TRTST
LINK:   LOADING

EXIT

.REE
 ---


ALGOL DIAGNOSTIC SYSTEM

FACILITIES (H FOR HELP)? T
                         -

FACILITIES (H FOR HELP)? G
                         -
```

This produces (at least) one line for each trace reference, as follows:

```
******SECTION
****S6:
********PHASE
************WRITE* PHASE
************PRINT* 6
************WRITE* :     BEGIN
********PHASE
************WRITE* END

******SECTION
****S7:
********PHASE
************WRITE* PHASE
************PRINT*  7
************WRITE* :     BEGIN
********PARFOR
********PHASE
************WRITE* END
```

Dynamic trace output always clears the terminal output buffer and begins a new line. The current dynamic block level is represented by two asterisks for each level, printed along the line. Note that a notional block surrounds the main program and each procedure. A maximum of sixty asterisks are printed before the line is folded: folding is shown by a number in the first two columns representing multiples of thirty dynamic block levels.

The name of the procedure or label appears at the end of the line of asterisks. As with the profile, library procedures are distinguished by a trailing asterisk, and labels by a trailing colon.

Procedures are traced at entry but not exit; however, a procedure exit can be inferred from the block levels.

Note that in Dynamic Trace mode, ordinary TTY output may be interspersed amongst Trace output (as in the example above).

18.8.2 Post-Mortem Trace

A post-mortem trace will be printed automatically if a batch job gives a run-time error. Under timesharing, after typing out the location and type of error, and the stack analysis, the diagnostic system offers the user various options. Of these T (for Trace print), produces a trace

similar to the one described under Dynamic Trace above, but with spaces instead of asterisks to represent block levels, and without, of course, any other TTY output. The post-mortem trace entries are maintained in a circular buffer in the Heap. The default length of the buffer is 100 (decimal) but it can be altered by using the /TRACE switch to the compiler, with an appropriate value. Tracing is the default option: users can compile their programs without trace information by using the /PRODUCTION switch to the compiler. However, the Trace entry mechanism is quite efficient, and users are urged not to exclude it, as this also has the effect of making the stack analysis less useful. Library procedures are always traced.

18.9 PERFORMANCE ANALYSIS

Various features are provided which are designed to help users wishing to evaluate the performance of their programs.

18.9.1 Heap Space

DECsystem-10 ALGOL is very flexible in its use of memory: space for arrays, strings, I/O buffers and so on is allocated in an area called the Heap, which lies between the program code and the stack. The default initial Heap size is 521 (decimal) words: if a program needs more Heap than is currently available the object-time system moves the stack up in memory to make more room, obtaining more core from the Monitor as necessary (subject to over-riding constraints such as the user's COREMAX).

For most programs this provides a reasonable balance between core use and computation. However, the stack shifting mechanism can be very expensive if it is used often and for small expansions. The user can find out how the Heap is being used by typing REENTER, and then $ (for statistics) after the program has executed (or simply typing CONTINUE immediately after the End of execution message). The produces output of the following format:

```
EXECUTION TIME:
ELAPSED TIME:
MAXIMUM HEAP SIZE:     , # OF STACK SHIFTS:
```

Maximum # of used words in the heap table:

(The last figure is a measure of the fragmentation of the free space in the heap: it is only produced if the optional Heap Integrity Checker is present in the object-time system; i.e., when assembly switch FTGETCHK is turned on.)

If the number of stack shifts is larger than, say, ten, an improvement in performance could be expected from re-compiling the program with the /HEAP switch value set to the maximum heap size given by the Statistics type-out.

18.9.2 Code Utilization

By judicious placement of labels and subsequent printing of the program's profile, the frequency with which a program executes particular sections of code can be monitored. The new library procedure INFO can be used to obtain more detailed timing information.

CHAPTER 19
TECHNICAL NOTES

These notes concern the authors' particular interpretation of the "Revised Report on the Algorithmic Language ALGOL-60" and its implementation.

a. At all times, strict left-to-right evaluation of statements is employed. Section 3.4.6 of the Revised Report has been construed by some experts to mean that left-to-right evaluation of expressions is not required. However, there are undoubtedly many ALGOL-60 programs in existence that rely on this feature.

b. Section 4.3.5 of the Revised Report requires that a GOTO Statement with a designational expression which is a switch with a subscript out of range be regarded as a dummy statement. Neither DECsystem-10 ALGOL nor any other ALGOL-60 implementations, to the knowledge of the authors, follow this rule if there is a side-effect involved in the evaluation of the subscript.

ALGOL INDEX

DECsystem-10 ALGOL
Programmer's Reference Manual
DEC-10-LALMA-B-D

READER'S COMMENTS

NOTE: This form is for document comments only. Problems with software should be reported on a Software Problem Report (SPR) form.

Did you find errors in this manual? If so, specify by page.

_____

_____

_____

_____

_____

_____

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

_____

_____

_____

_____

_____

_____

Is there sufficient documentation on associated system programs required for use of the software described in this manual? If not, what material is missing and where should it be placed?

_____

_____

_____

_____

_____

_____

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Non-programmer interested in computer concepts and capabilities

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code or Country _____

If you require a written reply, please check here. ☐

Please cut along this line.

---------- Fold Here ----------

---------- Do Not Tear - Fold Here and Staple ----------

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

digital

Software Communications
P. O. Box F
Maynard, Massachusetts 01754