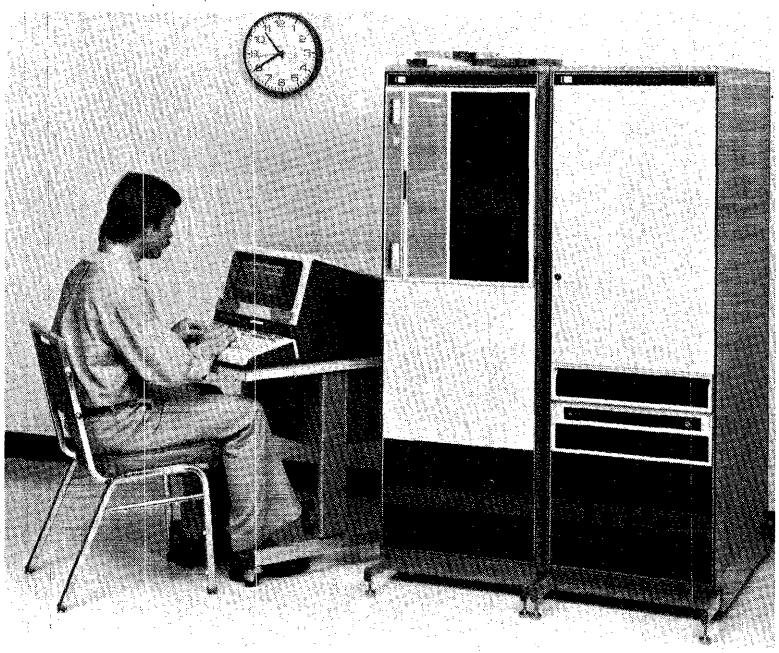
HEWLETT hp PACKARD

HP ALGOL

Reference Manual



HP ALGOL

Reference Manual

HEWLETT-PACKARD COMPANY
11000 WOLFE ROAD, CUPERTINO, CALIFORNIA, 95014

PART NO. 02116-9072

Printed in U.S.A. 11/76

LIST OF EFFECTIVE PAGES

Changed pages are identified by a change number adjacent to the page number. Changed information is indicated by a vertical line in the margin of the page. Original pages (Change 0) do not include a change number. Insert latest changed pages and destroy superseded pages.

Change 0 (Original) ..... NOV 1976

NOTICE

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

PREFACE

This manual describes the HP ALGOL language for Hewlett-Packard 2000 computer systems. HP ALGOL is similar to the source language described in "Revised Report on the Algorithmic Language ALGOL 60," *Communications of the ACM,* January, 1963.

Different versions of HP ALGOL have been designed to run on these operating systems:

SIO Subsystem
DOS III Operating Systems
Real-Time Executive II (RTE-II)
Real-Time Executive III (RTE-III)

In addition, programs compiled by ALGOL can run under the Basic Control System (BCS) and RTE memory based systems. You should check the reference manual of your system for information on how HP ALGOL is implemented.

This is a reference manual, not a training manual. You should already be familiar with your operating system and have had some programming experience.

For information concerning subroutines provided with your system, see *DOS/RTE Relocatable Library Reference Manual* (Part No. 24998-90001).

The first section of this manual describes ALGOL programs in general, and includes a comparison of HP ALGOL and ALGOL 60. The second section defines the basic concepts of ALGOL and explains how to declare identifiers. Section III gives a detailed description of assignment, GO TO, IF, CASE, WHILE, DO, FOR, and PAUSE statements. Input/Output statements and declarations are described in section IV. An explanation of procedures, both internal and external, is presented in section V. Section VI gives some general information about the HP ALGOL compiler. Section VII presents sample HP ALGOL programs. HP ALGOL and HP FORTRAN IV are compared in section VIII. Appendix A contains compiler and system error messages. Appendix B contains the HP ALGOL syntax in BNF productions. Appendix C lists the Hewlett-Packard character set for computer systems.

CONTENTS

CONTENTS (continued)

TABLES

INTRODUCTION

SECTION I

The word ALGOL is an acronym for ALGOrighmic Language. This section has been written to give you a brief introduction to algorithms, to describe the structure of ALGOL programs, and to point out the differences between ALGOL 60 and HP ALGOL.

1-1. ALGORITHMS AND PROGRAMS

An algorithm is a step by step method used to solve a problem or accomplish some task. A common algorithm is the definition of a factorial:

Factorial of n (n!):
If n = 0 then n! = 1.
If n > 0 then n! = n × (n-1) × ... × 3 × 2 × 1.
Example: 5! = 5 × 4 × 3 × 2 × 1 = 120

In this form you can understand the algorithm, but it cannot be executed on a computer. A computer program is an algorithm that has been written in a language which the computer understands. ALGOL is one such computer language.

A compiler is a program that translates computer programs from a symbolic language (such as ALGOL or FORTRAN) into binary code that can be loaded on a computer. The program read into the compiler is called the source code or source program. The machine instructions generated by the compiler are called the object code or object program. The object code can be loaded and executed on a computer.

The steps to producing a computer program are:

1. Define the problem.
2. Write an algorithm to solve the problem.
3. Translate the algorithm into ALGOL.
4. Run the ALGOL compiler to produce relocatable object code.
5. Run the loader to combine subroutines and produce executable machine code.
6. Execute the machine code.

1-2. HP ALGOL PROGRAMS

HP ALGOL statements and declarations do not have to begin in any particular column. (They must end by column 72.) One statement can be spread over several lines or several statements can appear on one line. Statements are separated from each other with semicolons (;).

HP ALGOL is a block structured language. An ALGOL block is a series of declaratons and/or statements that start with the word BEGIN and terminate with the word END.

As an example of HP ALGOL, here is a program to calculate a factorial:

```
PAGE  001

001 00000 HPAL,L,"FACTL"
002 00000 BEGIN
003 00001 COMMENT THIS PROGRAM READS A NUMBER FROM THE
004 00001   SYSTEM CONSOLE (LU 1) AND PRINTS ITS  FACTORIAL;
005 00001 INTEGER N, FACTL;
006 00004 READ(1, *, N);
007 00040 FACTL := 1;
008 00042 WHILE N > 0 DO
009 00045    BEGIN
010 00046    FACTL := FACTL*N;
011 00052    N := N - 1
012 00052    END;
013 00056 WRITE(1, #(" FACTORIAL =" I6), FACTL);
014 00101 END$

PROGRAM= 000105  ERRORS=000
```

Line 1 contains the control statement. The L indicates that the program is to be listed as it is compiled. The name of the program (in this case FACTL) is enclosed in quotes.

Line 2 is the first line of the program. Because a program is a block, the first statement must always be the word BEGIN.

Lines 3 and 4 contain a comment. Comments are included in programs to help the reader understand what the program does. Comments are ignored by the compiler and do not affect the object code that is generated. The semicolon terminates the comment.

Line 5 declares two integer variables: N and FACTL. Any variables used exclusively within a block must be declared at the beginning of the block.

Line 6 is the first executable statement in the program. It reads a number from logical unit 1 and puts it into N.

Line 7 initializes the factorial to one.

Line 8 is the beginning of a WHILE statement. The condition is checked (N greater than zero) and the following block is executed while the condition is true. If N is zero the first time it is checked, the block is never executed.

Line 9 is the beginning of an inner block.

Line 10 multiplies FACTL by N.

Line 11 sets up the value of N for the next iteration of the WHILE block.

Line 12 terminates the block that started in line 9. After line 11 is executed, the condition in line 8 is checked again.

Line 13 is executed only after the WHILE condition is false. The value of the factorial of N is printed on logical unit 1.

Line 14 terminates the block that began in line 2. The "$" indicates that this is the end of the program.

This program listing was formatted to clearly show ALGOL's block structure by indenting the statements. This is not required — it could have been compacted to:

```
001  00000 HPAL,L,"FACTL"
002  00000 BEGIN INTEGER N,FACTL;READ(1,*, N);FACTL:=1;WHILE N>
003  00042 0 DO BEGIN FACTL:=FACTL*N;N:=N-1 END;WRITE(1, #(
004  00064 " FACTORIAL =" I5),FACTL) END$

PROGRAM= 000105  ERRORS=000
```

Although both programs produce the same object code, the first one is easier to read and understand. Most of the program examples in this book indent blocks three spaces.

1-3. HP ALGOL AND ALGOL 60

This subsection describes the differences between HP ALGOL and ALGOL 60. If you are not already familiar with ALGOL 60, you may skip this discussion.

In addition to the major elements of ALGOL 60, HP ALGOL provides the following extensions:

1. REAL and INTEGER variables can be intermixed on the left side of assignment statements.

Example:

```
INTEGER A; REAL B;
A := B := 0;
```

2. IF statements can be nested within IF statements.

Example:

```
IF A=B THEN IF B=C THEN D=1 ELSE D=3;
```

3. All variables are treated as OWN variables (their values are not changed when execution terminates in the block where they are declared).

4. Variables and arrays can be initialized as you declare them.

5. Program constants can be given symbolic names with the EQUATE declaration.

Example:

```
EQUATE TABLESIZE := 126;
ARRAY TABLE[1:TABLESIZE];
```

6. READ and WRITE statements use FORMAT specifications and Logical Unit numbers (as in FORTRAN).

7. You can reference external routines written in HP ALGOL, FORTRAN, or Assembly Language.

8. Because HP ALGOL treats Boolean and integer expressions in the same manner, you can use NOT, AND, and OR in arithmetic expressions.

9. You can replace "STEP 1 UNTIL" with "TO" in FOR statements (only when the step value is 1).

10. You can define one word (integer) octal and ASCII constants.

11. Comments can be inserted after an ampersand (&) on any line.

12. The DO ... UNTIL, PAUSE and CASE statements have been added.

HP ALGOL also differs from ALGOL 60 in the following areas:

1. Recursion is not supported.

2. Parameters are called by reference instead of by name. (Parameters can also be called by value.)

3. The definition of a switch is not as general.

4. Lower case letters are not included in the basic character set. In addition, the following characters are substituted for ALGOL 60 characters:

ALGOL 60	HP ALGOL	MEANING
×	*	Multiplication
÷	\	Integer Division
≤	<=	Less Than or Equal
≥	>=	Greater Than or Equal
≠	#	Not Equal
¬	NOT	Logical Not
∧	AND	Logical And
∨	OR	Logical Or
$_{10}$	'	Scaling Factor for Real Constants
' '	"	String Quotes

You may use the standard ALGOL assignment operator := or the HP ALGOL assignment operator ←. The symbols ⊃ (Logical Implies) and ≡ (Logical Equate) are not included in HP ALGOL.

5. You cannot use the FOR list and WHILE form of the FOR statement.

6. Array bound values must be integer constants.

7. You must declare labels before you define them.

8. Storage for arrays and variables is not dynamic.

BASIC CONCEPTS

SECTION II

This section has been written to explain the basic elements of HP ALGOL: constants, identifiers, declarations, expressions, intrinsic functions, and program comments.

2-1. CONSTANTS

Constants are values used in your program that do not change. You can use four types of constants in HP ALGOL programs: decimal, octal, Boolean, and ASCII.

2-2. DECIMAL CONSTANTS

Decimal constants may be either real or integer numbers.

Real numbers use two words (32 bits) of memory. They have a 23 bit fraction (plus sign bit) and a seven bit (plus sign) exponent. Real numbers are significant to six or seven decimal digits, depending on the leading digit in the fraction. The largest number that can be represented is 2^{127} (approximately 10^{38}). The smallest positive number is 2^{-127} (approximately 10^{-38}).

Integer numbers require one word (16 bits) of memory. They are represented in two's complement form and can take any integer value between -32768 and 32767.

If you write a number and include a decimal point or scale factor, the compiler generates a two word real constant. Otherwise a one word integer constant is created. All decimal constants may be either signed or unsigned.

You may write real decimal constants as a number multiplied by an integral power of ten (scientific notation) by following the number with an apostrophe (') and a signed or unsigned integer. The apostrophe and power are called a scale factor.

Here are examples of decimal constants:

ALGOL Constant	Internal Value	Type
0	0	Integer
0.0	0.0	Real
-325	−325	Integer
+426	426	Integer
-.5384	−.5384	Real
-5.384'-1	−.5384	Real
200.	200.0	Real
+200.0	200.0	Real
.0002'6	200.0	Real
2.'+2	200.0	Real
'3	1000.0	Real

2-3. ASCII CONSTANTS

ASCII constants are 1 or 2 ASCII characters enclosed in quotes ("). Each character requires ½ word (8 bits) of storage. If you include only one character between the quotation marks, the character is placed in the right half of a word and the left half is filled with zeros.

ASCII constants are stored as integers. The internal representation of character values is described in Appendix C. Here are examples of ASCII constants:

ALGOL Constant	Internal Value (octal)
"HP"	044120
"A"	000101
" A"	020101
"A "	040440

2-4. OCTAL CONSTANTS

You can use octal (base 8) constants by typing a commercial at sign (@) followed by up to six octal digits. Octal constants must be in the range 0 to 177777. They are stored as integers. For negative numbers, place a minus (−) in front of the @

Examples:

```
@1230
@1030
-@121
```

(The constants "A", @101, and 65 generate the same internal value.)

2-5. BOOLEAN CONSTANTS

There are two Boolean constants: TRUE and FALSE. Internally, TRUE = −1 (all bits on) and FALSE = 0 (all bits off). Boolean constants are treated as integers.

2-6. IDENTIFIERS

Identifiers are names you can use to reference procedures, statements, variables and constants. The first character of an identifier must be a letter; succeeding characters may be letters or digits. You can have as many characters as you want in an identifier, but only the first fifteen are significant. (Additional characters are ignored.)

A number of identifiers already have special meanings in ALGOL. You may not use the identifiers in table 2-1 except as noted.

Table 2-1. Reserved ALGOL Identifiers

*ABS	ENDFILE	*LN	*SQRT
AND	*ENTIER	NOT	STEP
*ARCTAN	EQUATE	OR	SWITCH
ARRAY	*EXP	OUTPUT	*TAN
BACKSPACE	*FALSE	PAUSE	*TANH
BEGIN	FOR	*PI	THEN
BOOLEAN	FORMAT	PROCEDURE	TO
CASE	GO	READ	*TRUE
CODE	IF	REAL	UNLOAD
COMMENT	INPUT	REWIND	UNTIL
*COS	INTEGER	*ROTATE	VALUE
DO	*KEYS	*SIN	WHILE
ELSE	LABEL	*SIGN	WRITE
END		SPACE	

*These identifiers have been predeclared in that you can use them without declaring them. You may, however, override these declarations with your own.

2-7. DECLARATIONS

ALGOL programs consist of declarations and statements. Declarations are non-executable. They describe the properties of identifiers you use in your program. You must declare identifiers at the beginning of the outermost block where they are used. This section discusses five types of declarations: Equate, Type, Array, Label, and Switch. Input, Output, and Format declarations are discussed in Section IV. Procedure declarations are discussed in Section V.

Declarations are separated by semicolons.

2-8. EQUATE DECLARATION

You can assign identifiers to program constants with the EQUATE declaration. The form of the EQUATE declaration is:

EQUATE <variable> := <constant>, ..., <variable> := <constant>

Example:

```
EQUATE KEYBOARD:=1, BLANK:=" ", MASK:=@177, EPSILON:='38;
```

EQUATE identifiers may be used anywhere in a program where a constant can be used. The EQUATE declaration is particularly useful to set constants which may change if the program is redesigned.

Example:

```
EQUATE MAXLEN := 5, MAXENTRY := 100;
INTEGER ARRAY NAME[1:MAXENTRY, 1:MAXLEN];
```

NOTE

Throughout this book, the assignment operator := is used. HP ALGOL also accepts ← (_ on some terminals) as an assignment operator. You can use either one where := is shown in this book.

2-9. TYPE DECLARATION

Type declarations tell the compiler the names of simple variables and the type of data they are to contain. You can initialize variables when you declare them. The types of variables are REAL, INTEGER, or BOOLEAN. The general form of the type declaration is:

```
REAL     <identifier> := <constant>, ... ,<identifier> := <constant>
                                        or
INTEGER  <identifier> := <constant>, ... ,<identifier> := <constant>
                                        or
BOOLEAN  <identifier> := <constant>, ... ,<identifier> := <constant>
```

The ":= <constant>" portion, which is entirely optional and may be omitted, assigns an initial value to the variable.

Examples:

```
INTEGER I,J:=0,K:=@30,N:="N", START,M:=-3;
REAL X,Y,Z:=0.0;
BOOLEAN FLAG, MOREDATA;
```

If you don't specify an initial value, the HP ALGOL compiler does not provide one. The value of uninitialized variables depends upon the loader of your system.

2-10. ARRAY DECLARATION

The ARRAY declaration declares multi-dimensional arrays and gives the lower and upper bounds of each dimension. There is no limit to the number of dimensions an array may have. The bounds must be integer constants with the lower bound less than or equal to the upper bound.

In array declarations, you specify the type and write the word ARRAY followed by a list of identifiers (the names of the arrays separated by commas) and a list of bound pairs enclosed in brackets. The bound pairs are the lower and upper bound for each subscript separated by a colon. All the identifiers in the list have the number of dimensions and subscript limits given by the bound pairs.

If you want to declare several different sized arrays of the same type, follow the first declaration with a comma, give a list of new identifiers, and specify new bound pairs. There is no limit to the number of arrays you can define in an array declaration.

The general form of the ARRAY declaration is:

```
<type> ARRAY <identifier>, ... ,<identifier>[<lower bound>:<upper bound>,
                    ... ,<lower bound>:<upper bound>], ... ,
    <identifier>, ... ,<identifier>[<lower bound>:<upper bound>, ... ]
```

An array may be REAL, INTEGER, or BOOLEAN. If no type is given, REAL is assumed.

Examples:

```
REAL ARRAY TABLE[0:20, 1:12];
INTEGER ARRAY CONTROL[0:@31],CHARS[" ":"_"],UPPERCASE["A":"Z"];
BOOLEAN ARRAY USED[-321:0];
ARRAY MATRIX1, MATRIX2, MATRIX3 [1:4,1:6],
      MATRIX4, MATRIX5 [1:3,1:3,1:10];
```

In the last example, MATRIX1, MATRIX2, and MATRIX3 are REAL 4 by 6 arrays. MATRIX4 and MATRIX5 are REAL 3 by 3 by 10 arrays.

Array are stored in consecutive memory locations with the rightmost dimension increasing first.

Example:

```
ARRAY A[1:3,1:3,1:3]
```

is stored in the order
A[1,1,1], A[1,1,2], A[1,1,3], A[1,2,1], A[1,2,2], A[1,2,3], A[1,3,1], A[1,3,2], A[1,3,3], A[2,1,1], A[2,1,2], ..., A[3,3,2],A[3,3,3]

You can initialize the elements of an array when you declare it by placing an assignment operator after the bounds list and following it with a list of constants separated by commas. In an array declaration, only the last array named may be initialized. (Other arrays can be initialized in other declarations.)

Example:

```
INTEGER ARRAY DIGITS [0:20] := "0", "1", "2", "3", "4",
                               "5", "6", "7", "8", "9";
```

In this example, the first ten elements (0 through 9) of DIGITS are initialized to the ASCII representation of their index. Elements 10 through 20 are undefined.

2-11. LABEL DECLARATION

The label declaration indicates that you will use the specified identifiers as labels of statements in your program. Labels may be used in GO TO statements or passed as parameters to subroutines.

The general form of the label declaration is:

LABEL <identifier>, ... , <identifier>;

Example:

```
LABEL ERROR, RESTART, LOOP;
```

All declared labels must be defined (assigned to a statement) somewhere in your program. Defining labels is described in Section III.

2-12. SWITCH DECLARATION

A switch is a set of labels which can be entered as objects of a GO TO statement. During execution of the GO TO statement, the switch identifier uses an indexing parameter to determine which label is used.

General format:

SWITCH <identifier> := <label>, ... , <label>;

There is no limit to the number of labels which may be used in the switch. The labels are associated from left to right with the positive integers (1,2,3...).

Example:

```
LABEL L1,L2,L3,FINISH;
SWITCH SW := L1,L2,L2,L3,FINISH;
INTEGER I;
 :

I := 3;
GO TO SW[I];
 :
```

In this example the indexing parameter, I, is set to 3. Control is passed to L2, the third label in the switch list.

All the labels must be declared before the switch declaration. When the indexing parameter is less than 1 or greater than the number of labels, the GO TO statement is bypassed; execution continues with the next statement.

2-13. VARIABLES

There are two types of ALGOL variables: simple variables and subscripted (array) variables. Each of these may be real, integer, or Boolean, depending on how you declared them.

Variables must be declared before you use them. Subscripted variable names must be followed by subscript expressions enclosed in brackets. The general form is:

<identifier>[<expression>, ... , <expression>]

The number of subscript expressions must be identical to the number of dimensions specified in the array declaration. Each subscript must have an integer value which lies within the bounds specified for the array in its declaration.

Examples:

```
TABLE[I, J+3*A]
USED[ TEXT[3*X]+1 ]
```

If your program tries to access an array element beyond its declared bounds, the INDEX? error message is printed and 0 is returned as the value of the element.

2-14. ARITHMETIC EXPRESSIONS

An arithmetic expression is a mathematical formula containing constants, variables, functions, parenthesis, and arithmetic operators. (Constants, variables, functions, and parenthesized expressions are known as primaries.) The value of a variable is the last value assigned to it. The value of a function is the value resulting from the computation defining the function procedure.

The following operators are valid for arithmetic expressions:

^	Exponentiation (↑ on some terminals)
*	Multiplication
/	Real Division
\	Integer Division
MOD	Remainder from Integer Division
+	Addition
-	Subtraction
NOT	Complement bits
AND	Logical AND bits
OR	Inclusive OR bits

All operators except NOT work on two operands. NOT works on one operand.

The results of arithmetic operations can be real or integer. For +, −, and * the result is real unless both of the operands are integer. The operator / always produces a real result. The results are always integer for \, NOT, AND, and OR.

Examples of \, MOD, and logical functions:

Expression	Value
7\8	0
3\2	1
(-3)\(-2)	1
(-24)\5	-4
7 MOD 8	7
3 MOD 2	1
(-3)MOD(-2)	-1
(-24)MOD 5	-4
@777 AND @12345	@000345
@077 OR @120000	@120077
NOT @123456	@054321

WARNING

Integer and real overflow resulting from arithmetic operations may not be detected during program execution.

The operator ^ denotes exponentiation and is defined as follows for integer I, real R, and real or integer A:

A^I

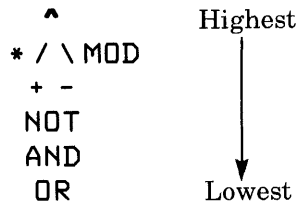
I>0: A*A* ... *A (I times). Result is the same type as A.
I=0 and A≠0: 1. Result is the same type as A.
I<0 and A≠0: 1/(A^(−I)). Result is REAL.
I≤0 and A=0: Undefined

A^R

A>0: EXP(R*LN(A)). Result is REAL.
A=0 and R>0: 0.0. Result is REAL.
A=0 and R≤0 or
A<0 and R≠0: Undefined.

You can mix integer and real operands in an expression. If a real result is assigned to an integer, the result is rounded to the nearest integer, not truncated.

The order of operations is determined by parentheses and the normal precedence of operators. The normal precedence is:



If operands have the same precedence, they are performed from left to right.

Examples of arithmetic expressions:

ALGOL	Mathematics
`2^I^J`	$(2^I)^J$
`2^(I^J)`	$2^{(I^J)}$
`A+B*C*D/E+F`	$A + \frac{B \cdot C \cdot D}{E} + F$
`(A+B)*C+D/(E+F)`	$(A + B) \cdot C + \frac{D}{E + F}$
`X/Y*Z`	$\frac{X}{Y} \cdot Z$

2-15. BOOLEAN EXPRESSIONS

A Boolean expression is a rule for computing a logical value (TRUE or FALSE). The following relational operators are valid in Boolean expressions:

<	Less Than
<=	Less Than or Equal
=	Equal
>=	Greater Than or Equal
>	Greater Than
#	Not Equal

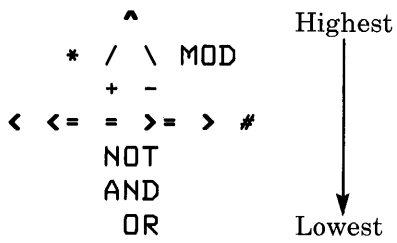
The meaning of the logical operators NOT, AND, and OR is given by the following truth table where B1 and B2 are Boolean expressions:

B1	B2	B1 OR B2	B1 AND B2	NOT B1
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	TRUE	FALSE	FALSE
FALSE	TRUE	TRUE	FALSE	TRUE
FALSE	FALSE	FALSE	FALSE	TRUE

Examples of Boolean expressions:

```
X = -2.46
(CHAR >= "A" AND CHAR <= "Z") OR (CHAR >= "0" AND CHAR <= "9")
A^2 < 100
3*Q > 4*A+2  AND  Q>0.0
A<B  AND  B#0
NOT (CHAR < "A"  OR  CHAR > "Z")
```

As shown in the examples, arithmetic expressions can be part of Boolean relations. The complete hierarchy of operations is:



As with arithmetic expressions, Boolean expressions can be parenthesized to change the precedence of operations.

In HP ALGOL, there is no difference between Boolean and integer values. As a result, Boolean and arithmetic operations can be mixed within expressions.

Integer values are considered to be true when they are negative and false when they are positive or zero. (Only the sign bit is significant in determining the logical value.) The Boolean constants TRUE and FALSE are equivalent to −1 and 0.

For all logical operations, only the sign bit is used. Because of this, evaluation of a Boolean expression may produce a value that is not −1 (TRUE) or 0 (FALSE). For example, the following code may not work:

```
  :
  :
FLAG := A>B;
IF FLAG = TRUE THEN
   <statement>;
  :
  :
```

As a result of the > operation, only the sign bit is set in FLAG, and the value may not be TRUE even when A is greater than B. Instead of this, you should use:

```
  :
  :
FLAG := A>B;
IF FLAG THEN
   <statement>;
  :
  :
```

WARNING

Integer overflow resulting from Boolean expressions is not detected during program execution. Integer overflow can result from comparing integer values that differ by more than 32767.

2-16. CONDITIONAL EXPRESSIONS

Another type of arithmetic expression is the Conditional Expression. It has the form:

IF <Boolean expression> THEN <expression 1> ELSE <expression 2>

The Boolean expression is evaluated first. If it results in a true value, the value of the expression is <expression 1>. Otherwise the value is <expression 2>. Each of the subexpressions may have the conditional form.

Examples:

```
IF A>0 THEN A ELSE -A
IF FLAG THEN N ELSE N+3
IF CHAR >= "A" AND CHAR <= "Z" THEN ALPHA ELSE NUMERIC
IF A<B THEN
   IF A<C THEN A ELSE C
   ELSE IF B<C THEN B ELSE C
```

The value of the first expression is the absolute value of A. The value of the last expression is the minimum of A, B, and C.

Conditional expressions may be enclosed in parentheses and combined with other expressions.

Example:

```
(IF A<B THEN A ELSE B) + C
```

is equivalent to

```
IF A<B THEN A+C ELSE B+C
```

A conditional expression may be used anywhere an arithmetic expression is legal.

2-17. ASSIGNED EXPRESSIONS

Arithmetic subexpressions can be assigned to variables within expressions. To do this, enclose the assignment within parentheses. For example, the expression

```
3 + (A:=3*2) + 7
```

has the value 16. In addition, it assigns the value 6 to the variable A.

The subexpression can be any arithmetic, Boolean or conditional expression.

2-18. INTRINSIC FUNCTIONS AND PREDECLARED IDENTIFIERS

HP ALGOL provides 13 intrinsic functions. The intrinsics which require a parameter will accept an expression enclosed in parentheses. (If you are passing a variable, constant, or function value as the parameter, no parentheses are needed.) The trigonometric functions use radian measure. The intrinsic functions are listed in table 2-2.

Table 2-2. ALGOL Intrinsic Functions

NAME	MEANING	TYPE OF RESULT	VALID RANGE OF PARAMETER
ABS X	Absolute Value; $\|X\|$	Same as X	
SIGN X	1 if $X > 0$ 0 if $X = 0$ -1 if $X < 0$	Integer	
SQRT X	Square Root; $\sqrt{X}$	Real	$X \geq 0$
SIN X	Trigonometric Sine	Real	$\| X/\pi + 1/2 \| \leq 2^{15}$
COS X	Trigonometric Cosine	Real	$\| X/\pi \| \leq 2^{14}$
TAN X	Trigonometric Tangent	Real	$X \leq 2^{14}$
ARCTAN X	Arctangent; $\tan^{-1} X$	Real	
TANH X	Hyperbolic Tangent	Real	
LN X	Natural Logarithm (base e)	Real	$X > 0$
EXP X	Exponential; e^x	Real	$X < 124/\log_2 e$
ENTIER X	Truncation; Largest Integer $\leq X$	Integer	$X < 32767$
ROTATE I	Rotate I 8 Bits; Swap Halfwords	Integer	Integer
KEYS	Value of Switch Register	Integer	(No Parameter)

The parameter passed to ROTATE must be an integer. KEYS does not have a parameter; it returns the value stored in the Switch Register. All other intrinsics accept an integer or real parameter.

Examples:

```
ABS SIN(3*X)
SQRT 3
EXP(4*I + 2)
```

Some of the intrinsics generate run-time errors if the value of the parameter is not in an acceptable range. (See Appendix A for error messages.)

In addition to intrinsic procedures, ALGOL has predeclared values for the identifiers PI, TRUE, and FALSE. You use them as you would use constants. Their values and types are listed in table 2-3.

Table 2-3. Predeclared ALGOL Variables

NAME	VALUE	TYPE
PI	3.14159	Real
TRUE	−1	Integer
FALSE	0	Integer

You can override the standard meaning of intrinsic functions and predeclared identifiers by declaring them in your program.

2-19. COMMENTS

Comments are statements you include to explain a program; they do not affect the code that is generated. In ALGOL programs, a comment may be included anywhere a space would be permitted. Comments have several forms in HP ALGOL programs.

All characters between the word COMMENT and the next semicolon are treated as comments. They may be continued for many lines.

Characters on a line to the right of an ampersand (&) are treated as comments.

The compiler ignores all symbols following an END statement up to the next END, ELSE, UNTIL, semicolon or dollar sign.

Here is an example of a commented procedure:

```
REAL PROCEDURE INNERPRODUCT(A,B,N);
   VALUE N; INTEGER N; ARRAY A,B;
   BEGIN COMMENT COMPUTE THE SUM OF A[I]*B[I]
                 FOR I BETWEEN 1 AND N;
   REAL SUM; & HOLDS SUM OF PRODUCTS
   INTEGER I;& INDEXING VARIABLE
   SUM := 0.0;
   FOR I := 1 TO N DO
      SUM := SUM + A[I]*B[I];
   INNERPRODUCT := SUM
   END OF INNERPRODUCT;
```

ALGOL STATEMENTS

SECTION III

This section describes the executable statements in HP ALGOL (except I/O statements). Any identifiers referenced by ALGOL statements must be declared before they are used.

Statements are separated from one another by semicolons.

3-1. LABELS

A label is an identifier used to reference a statement. As described in Section II, labels must be declared before they are defined or referenced.

You define a label by placing it and a colon (:) before the statement it references. Several labels may reference the same statement. The general form is

<label>:<label>: ... <label>:<statement>

Examples:

```
L1: A := B*3;
LOOP: L2: L4:  COUNT := COUNT + 1;
```

Labels can be placed before an END statement if the preceding statement is terminated with a semicolon.

3-2. ASSIGNMENT STATEMENTS

You can use assignment statements to assign the value of an expression to one or more variables. The general form is

<identifier 1> := <identifier 2> := ... <identifier n> := <expression>

The expression following the last assignment operator (:=) is computed and then assigned to each of the variables in turn from right to left. (HP ALGOL also accepts the symbol ← in place of :=.)

Examples:

```
AREA := 2*PI*R;
A := TABLE[1,1] := C := 0;
TERM := (IF 2 MOD K = 0 THEN 1 ELSE -1)*(X^(2*K+1))/(2*K+1)
FLAG := LEN<15;
```

The variables which precede the assignment operator are called left part variables. They do not all need to be of the same type. In any case, the effect of the statement is the same as if you wrote

```
<identifier n> := <expression>;
                :
<identifier 2> := <identifier 3>;
<identifier 1> := <identifier 2>;
```

If a real value is assigned to an integer, the result is rounded (not truncated). The result is the same as adding .5 and truncating. If the real portion is exactly .5, the number is rounded up to the next integer.

Example:

Suppose X and Y are real and I is integer. The statement

```
X := I := Y := .7;
```

causes the following assignments

```
Y := .7;
I := 1;
X := 1.0;
```

3-3. GO TO STATEMENT

The GO TO statement causes an unconditional transfer of program execution to a labeled statement. The general form is

```
GO TO <designational expression>
```

The designational expression can be a label or a switch. As explained in Section II, a switch is a set of labels used with an indexing parameter. The general form of a GO TO statement using a switch is

```
GO TO <switch identifier>[<expression>]
```

Examples:

```
GO TO FINISH;
GO TO SW3[3*I - 4];
```

You should not jump into the middle of a block from outside the block. A block should be entered only at its head. Otherwise, run time errors can occur.

The IF, CASE, WHILE, DO, and FOR statements provide efficient ways to control program execution. Good programmers usually can write entire ALGOL programs without using one GO TO statement. (The most common use for GO TO statements is to jump to the end of a block when an error is detected.)

3-4. IF STATEMENT

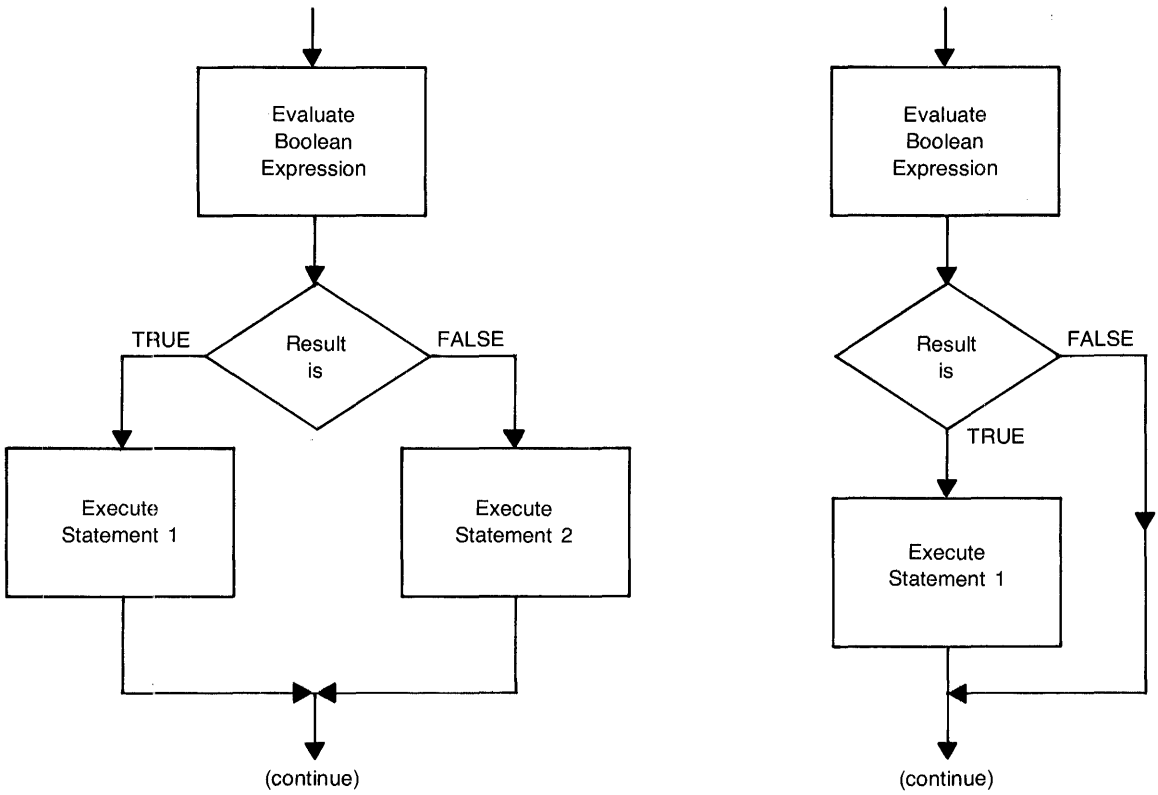
You can use the IF statement to select a statement (or block) to be executed depending upon a condition. The IF statement has two general forms:

IF <Boolean expression> THEN <statement 1> ELSE <statement 2>

or

IF <Boolean expression> THEN <statement 1>

These correspond to the following flowcharts:



In the first form, if the Boolean expression is true statement 1 is executed and statement 2 is skipped. Otherwise statement 1 is skipped and statement 2 is executed. If you use this form, you must be sure not to insert a semicolon between statement 1 and ELSE.

In the second form, statement 1 is executed when the Boolean expression is true. Otherwise it is skipped.

Examples:

```
IF A>B THEN MAX:=A ELSE MAX := B;
IF A=B THEN FOUND := TRUE;
IF NOT FOUND THEN GO TO LOOP;
```

Each of the statements following the THEN or ELSE can be an IF statement. Each ELSE is always associated with the closest preceding unmatched IF.

For example,

```
IF A<B THEN IF B<C THEN <statement 1> ELSE <statement 2>;
```

is interpreted as

```
IF A<B THEN
   BEGIN
   IF B<C THEN
      <statement 1>
   ELSE
      <statement 2>
   END;
```

not as

```
IF A<B THEN
   BEGIN
   IF B<C THEN
      <statement 1>
   END
ELSE
   <statement 2>;
```

Statement 1 is executed when A<B and B<C. Statement 2 is executed when A<B and B≥C. (In the second section of code, statement 2 is executed whenever A≥B.)

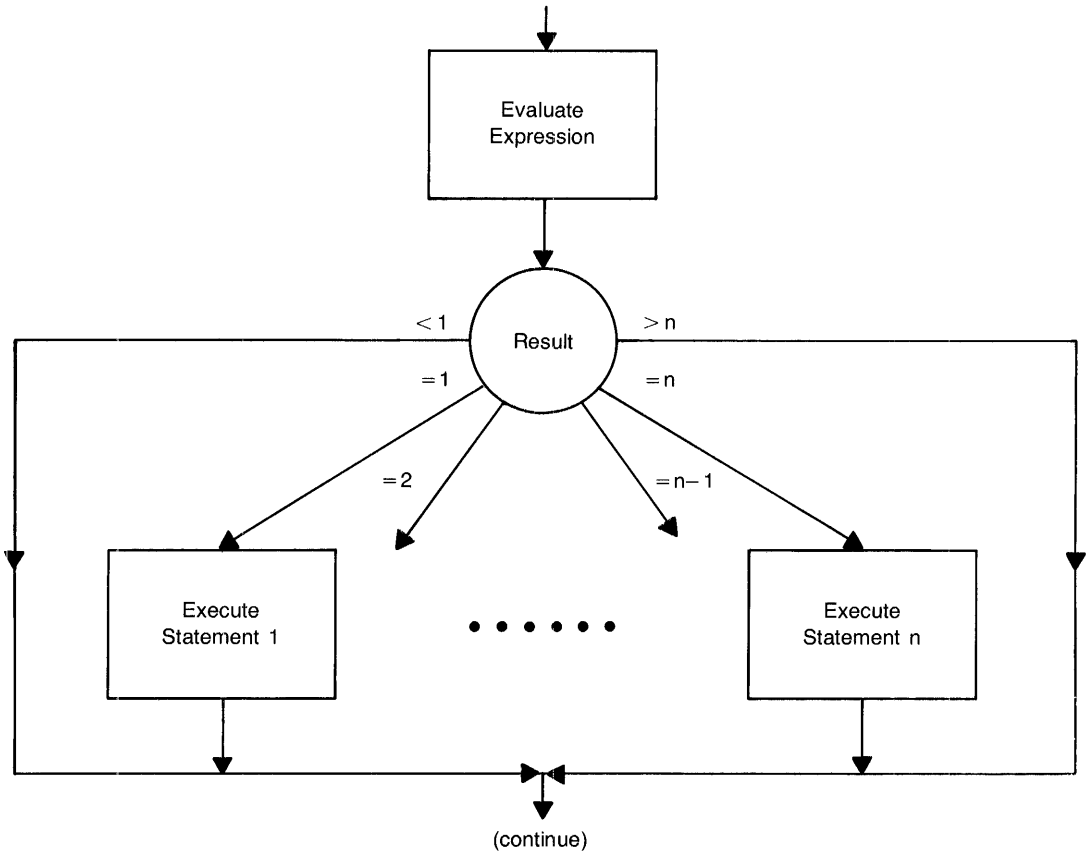
3-5. CASE STATEMENT

The CASE statement is another method you can use to select statements (or blocks). One statement from any number of ALGOL statements is chosen depending on the value of an arithmetic expression. The general form of the CASE statement is:

```
CASE <expression>
  BEGIN
  <statement 1>;
  <statement 2>;
      .
      .
      .
  <statement n>
  END
```

The expression is evaluated. Real results are rounded to the nearest integer. If the expression is between 1 and n, the statement corresponding to that value is executed. Otherwise the entire CASE statement is bypassed.

The flowchart corresponding to the CASE statement is



Example:

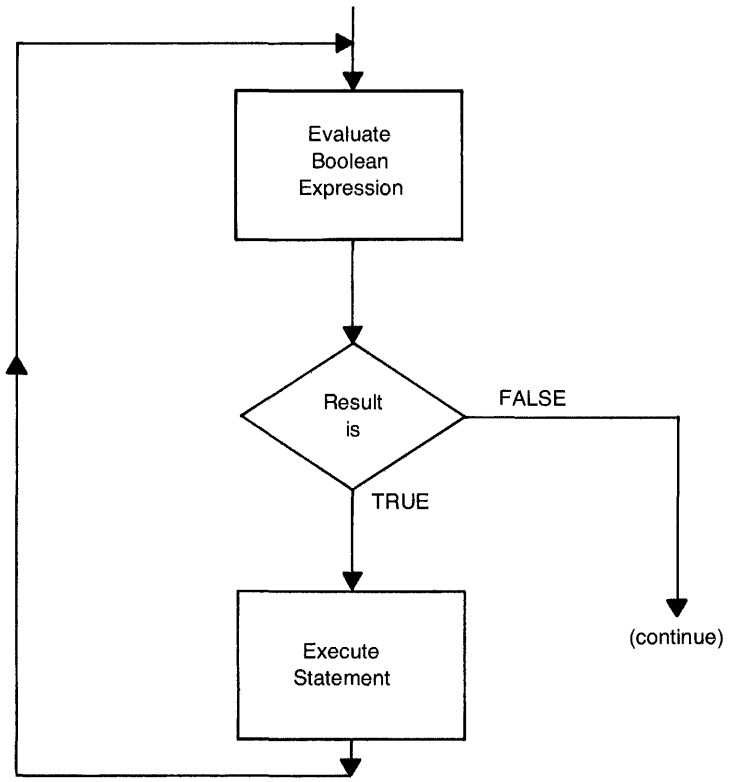
```
CASE INDEX+3
   BEGIN
   LENGTH := 0;
   DONE := TRUE;
   TERM := SIN(DEGREES*PI/180.0)
   END;
```

3-6. WHILE STATEMENT

The WHILE statement causes repetition of a statement (or block) as long as a condition is true. The general form is

WHILE <Boolean expression> DO <statement>

The flow chart for the WHILE statement is



The statement will never be executed if the condition is false for the first iteration.

Example:

```
I := 1;
WHILE X#TABLE[I] AND I<50 DO
   I := I + 1;
```

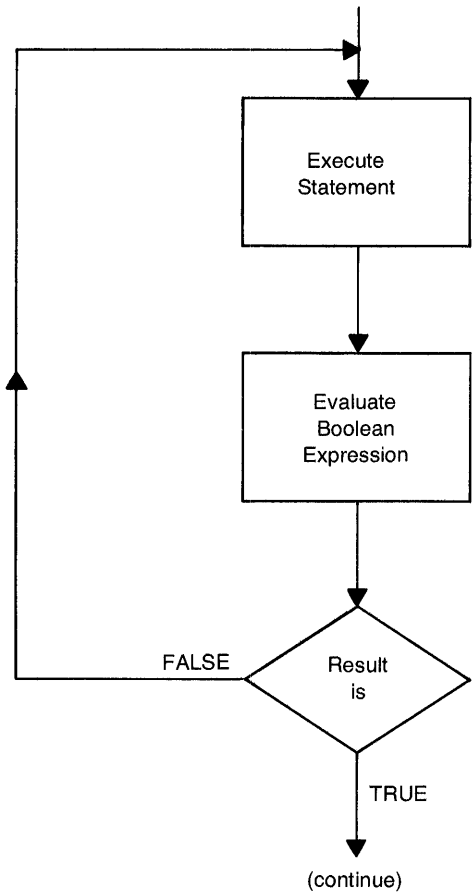
This example finds the element in the array TABLE that is equal to X (if any exists).

3-7. DO STATEMENT

The DO statement causes repetition of a statement (or block) until a condition becomes true. It differs from the WHILE statement in that the statement is executed the first time through whether the condition is true or false. The form of the statement is

DO <statement> UNTIL <Boolean expression>

The flow chart for the DO statement is



Example:

```
I := 1;
DO
   BEGIN
   READ(5, *, TABLE[I]);
   I := I+1
   END
UNTIL I>50 OR TABLE[I]=0.0;
```

This example puts numbers in an array until a zero is read or the table is full.

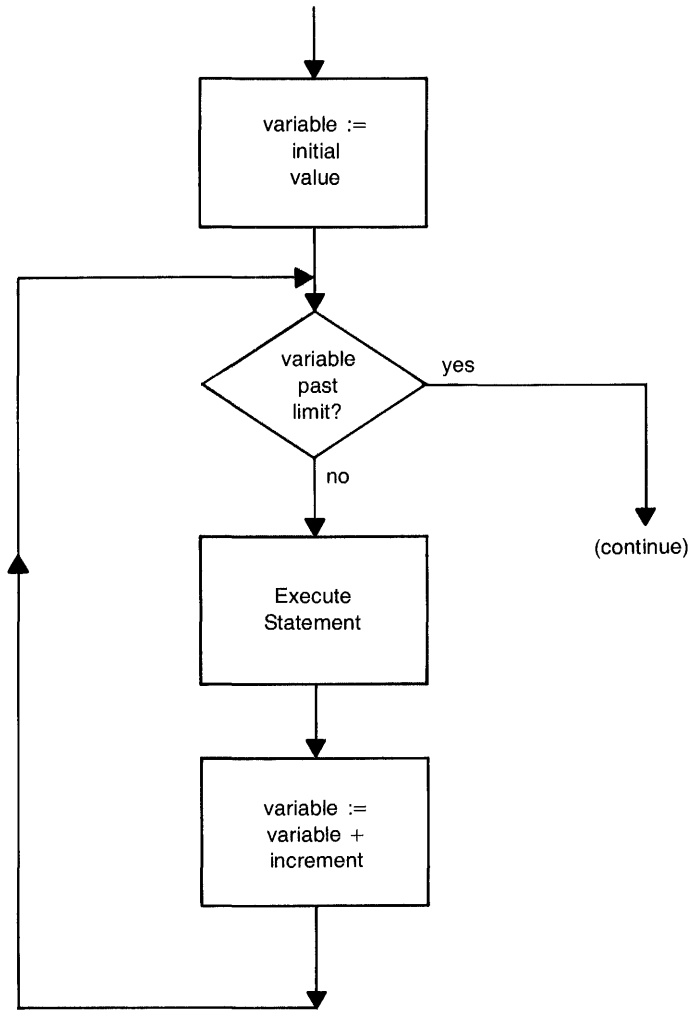
3-8. FOR STATEMENTS

You can use the FOR statement to repeat a statement (or block) while assigning a sequence of values to a control variable. The general form of the FOR statement is

```
FOR <variable> := <initial value> STEP <increment> UNTIL
    <final value> DO <statement>
```

The control variable must be a declared integer variable. The initial value, increment, and final value can be any expression, including negative ones.

The flow chart for this statement is



The control variable is assigned the initial value. Then the statement is executed while the control variable is incremented until it exceeds the final value in the direction (positive or negative) of the increment. (If the initial exceeds the final value, the statement is never executed.)

The statement

```
FOR I:=J STEP K UNTIL L DO <statement>;
```

is equivalent to

```
        I := J;
LOOP:   IF SIGN(K)*I > SIGN(K)*L THEN GO TO DONE;
        <statement>;
        I := I + K;
        GO TO LOOP;
DONE:   <continuation of program>;
```

If the increment or final value is an expression (not a simple variable) it will be calculated and saved before the statement is executed. The value will not be modified while the FOR statement is executing. (If the increment or final value is a simple variable, changes to the variable will affect the increment or final value.)

If the increment is an expression which evaluates to zero, the control variable will not change value.

If the increment is 1, you can replace "STEP 1 UNTIL" with "TO". For example,

```
FOR I:=1 TO N DO
```

is equivalent to

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

One use of FOR statements is to initialize arrays. Suppose A is an N by N array. You can set all the elements to zero with

```
FOR I := 1 TO N DO
   FOR J := 1 TO N DO
      A[I,J] := 0.0;
```

3-9. PAUSE STATEMENT

The PAUSE statement causes your program to halt. You can use it for debugging purposes or for changes to the hardware (mounting a tape, for example). The form of the statement is

```
PAUSE
```

This statement performs different functions in different operating systems:

For RTE and DOS, the PAUSE statement causes the job to be suspended until an operator enters a GO directive.

For SIO, the word "PAUSE" is printed on the system console and the computer is halted. Program execution continues when the operator presses the RUN button.

3-10. DUMMY STATEMENTS

Sometimes you may find it useful to specify a statement which causes no operation. You can do this by placing an extra semicolon in the program.

One use for a dummy statement is in a case statement when you want no operation for values of the controling expression. For example, suppose you want no operation when N is equal to three. You could write

```
CASE N
   BEGIN
   X := SIN X;
   Y := COS Y;
   ;
   X := X*X
   END;
```

You can also use dummy statements to place labels.

Example:

```
I := I+1;
EXIT1: END;
```

3-11. BLOCKS

As stated in the introduction, a block is a section of code that starts with BEGIN and terminates with END.

BEGIN and END are not considered statements; they are block brackets. No semicolon is needed after BEGIN and none is needed between the last statement of a block and END. If you insert a semicolon, it is treated as a dummy statement.

ALGOL blocks have the general form

```
BEGIN
<declaration>;
<declaration>;
        .
        .
        .
<declaration>;
<statement>;
<statement>;
        .
        .
        .
<statement>
END
```

All declarations must be specified before any executable statements. If there are no declarations, the block is known as a compound statement.

A block is a type of ALGOL statement. Each block may have any number blocks within it. Declarations made in a block are valid only within the block (and any blocks contained in the

block). You can redeclare identifiers for an inner block, as shown in this example:

```
BEGIN
INTEGER I,J,K;
REAL X,Y,Z;
   .
   .
   .
J:=1;
I:=3;
   .
   .
   .
   BEGIN                    )
   INTEGER I,X,L;           |
   .                        |
   .                        |
   .                        |
   I:=7;                    } Inner Block
   J:=4;                    |
   .                        |
   .                        |
   .                        |
   END                      |
END$                        )
```

In this example, the variables I, J, K, X, Y, and Z are declared in the outer block. The variables I, X, and L are declared in the inner block. Because the names I and X are declared in both blocks, two different values may be referenced, depending on whether the statement referencing the variable is in the inner block or not. Only one location is referenced for the variables J, K, Y, and Z throughout the blocks. The variable L is valid only in the inner block. (Such variables are said to be local.) After the outer block is executed, J has the value 4 and I has the value 3.

You can use local variables to perform functions that are contained within inner blocks. This is one way to be sure that variables in the main program are not unintentionally altered.

Blocks can be nested indefinitely.

Example:

```
BEGIN
I := 0;
IF A<0 THEN
   BEGIN
   I := 1;
   IF B<0 THEN
      BEGIN
      I := 2;
      IF C<0 THEN I:= 3
      END
   END
END
```

After executing this code, I has the value

0 if $A \geq 0$
1 if $A < 0$ and $B \geq 0$
2 if $A < 0$ and $B < 0$ and $C \geq 0$
3 if $A < 0$ and $B < 0$ and $C < 0$

INPUT/OUTPUT

SECTION IV

The basic HP ALGOL input and output statements are READ and WRITE. You can declare I/O lists and format specifications to be used with READ and WRITE. HP ALGOL also has statements for Magnetic Tape I/O.

The input list, output list, and format declarations are like other ALGOL declarations in that they must come at the beginning of a block (before any executable statements).

4-1. INPUT LIST

You can declare a list (or lists) of items that will be used in READ statements. The general form of the INPUT declaration is

INPUT <list identifier>(<list element>, ... , <list element>), ... ,
<list identifier>(<list element>, ... , <list element>)

Whenever you use the list identifier in a READ statement, you refer to all the elements in the associated list. Input list elements can be

a simple variable,
a subscripted variable,
another input list identifier, or
a FOR element.

A FOR element is similar to the FOR statement, except the statement part is replaced by one or more input list elements. If more than one element is used, they must be enclosed in brackets.

Examples of FOR elements:

```
FOR I := 1 STEP 2 UNTIL 10 DO TABLE[I]

FOR J := 1 TO N DO
    [V[J], FOR I:= 1 TO N DO [A[I,J], B[I,J]]]
```

All the elements in an input list must be previously declared.

Examples of input lists:

```
INPUT IN1(A, B[2], FOR I:= 1 TO N DO TBL[I]),
      IN2(X, Y, IN1, Z);
```

4-2. OUTPUT LISTS

You can declare a list (or lists) of items that will later be used in WRITE statements. The form is similar to the INPUT declaration:

```
OUTPUT <list identifier>(<list element>, ... , <list element>), ... ,
       <list identifier>(<list element>, ... , <list element>)
```

Output list elements may be

a simple variable,
a subscripted variable,
another output list identifier,
a FOR element, or
an expression.

Examples of output lists:

```
OUTPUT TRIG(FOR I:= 1 TO N DO
                [I, A[I], SIN A[I], COS A[I]]),
       NOTUSEFUL(3*I, PI/180.0, 14.321, X*Y+2);
```

4-3. FORMAT DECLARATIONS

You use FORMAT declarations to describe the arrangement of data that is read or written. (You will probably find that free field input is superior to formatted input for numeric data.) The general form of the FORMAT declaration is:

```
FORMAT <format identifier> (<specification> ... <specification>), ... ,
       <format identifier> (<specification> ... <specification>)
```

The format specifications are separated by commas or slashes.

Format specifications fall into three classes: Real, Integer, and Editing.

4-4. REAL FORMAT SPECIFICATIONS

You can transfer data to and from real values with the following specifications:

E	Exponent
F	Fixed Point
G	General

In addition, you can specify that real values are read or written with a scale factor.

4-5. E SPECIFICATION: EXPONENT

Format: E w. d
w = field width
d = number of digits in fraction

Input: The number in the input field is converted to a real number and stored.

The number read must have the form of an ALGOL decimal constant, except the character E is used instead of ' for a scale factor. If the exponent is signed, the E is not necessary. Thus, 12.0E+02, 12E2, and 12+02 all represent 1200.0. The number may be positioned anywhere within the field. Spaces in the field are ignored.

When no decimal point is present in the input field, d acts as a negative power of ten scaling factor (otherwise it does nothing). The internal value of the quantity will be

$$(\text{integer portion}) \times 10^{-d} \times 10^{\text{exponent portion}}$$

Example: Suppose the characters 1234+5 appear in a field read with an E12.8 specification. The result is 1.234.

Output: The output field consists of

a blank or negative sign
a decimal point
the d most significant digits of the internal value
the sign of the exponent
a two digit exponent.

The field must be wide enough to contain the sign, decimal point, d digits, and exponent. For this reason, w should be at least d+5. If the field is not large enough, dollar signs ($) are inserted in the entire field. If the field is longer than the output value, the quantity is right-justified with spaces to the left. The number printed is rounded (not truncated).

4-6. F SPECIFICATION: FIXED POINT

Format: F w. d
w = field width
d = number of digits in fraction

Input: Same as E Specification.

Output: The value occupies w positions and appears as a decimal number with d digits following a decimal point (no exponent). The quantity is right justified in the field, and rounded.

The field must be wide enough to hold the significant digits, sign (if the value is negative), and decimal point. If the field is too short, dollar signs are placed in the entire field.

4-7. G SPECIFICATION: GENERAL

Format: Gw. d
w = field width
d = number of digits in fraction

Input: Same as E Format

Output: The G specification acts like an E or F specification, depending on the magnitude of the value being output. If X is the value being printed, the G format is the same as:

Magnitude	Equivalent Specification
$.1 \leq X < 1$	F(w−4). d, 4X
$1 \leq X < 10$	F(w−4). (d−1), 4X
•	•
•	•
•	•
$10^{d-2} \leq X < 10^{d-1}$	F(w−4). 1 , 4X
$10^{d-1} \leq X < 10^{d}$	F(w−4). 0 , 4X
All other values	E w. d

4-8. SCALE FACTOR. A scale factor provides a way to normalize real values. (It has no effect on integer values.)

Format: nP
n = an integer or negative integer constant

The default scale factor is 0. During a formatted I/O operation, once a scale factor is established it remains in effect until another scale factor is read.

Input: If there is an exponent in the external field, the scale factor has no effect. Otherwise the internally represented number is equal to the external number times 10^{-n}.

Output: E specification: The real constant part is multiplied by 10^{n} and the exponent is decreased by n.

F specification: The value is multiplied by 10^{n}.

G specification: No effect if the value is in range for F representation. Otherwise it has the same result as for E.

Examples:

```
-1PE10.4, 1PF10.3, 2PG18.8, 0PF6.2
```

4-9. INTEGER FORMAT SPECIFICATIONS

You can transfer data to and from integer variables with the following specifications:

I	Decimal Integer
@, K, and O	Octal
A	ASCII
R	ASCII, Right-Justified
L	Logical (Boolean)

4-10. I SPECIFICATION: DECIMAL INTEGER

Format: I w
w = width of field

Input: The characters in the input field are read as a signed or unsigned decimal integer constant. Blanks (both leading and trailing) are treated as zeros.

Output: The internal value is converted to a decimal integer constant right justified in its field. (The sign is printed only if the value is negative.) If the field is too short, dollar signs are placed in the entire field.

4-11. @, K, AND O SPECIFICATIONS: OCTAL

Format: @w or
Kw or
Ow
w = width of field

Input: If w ≥ 6, up to six octal digits are stored; non-octal digits are ignored. The value of the field must be no greater than 177777. If w < 6 or less than six octal digits are read from the field, the number is right-justified in a computer word and filled with leading zeros.

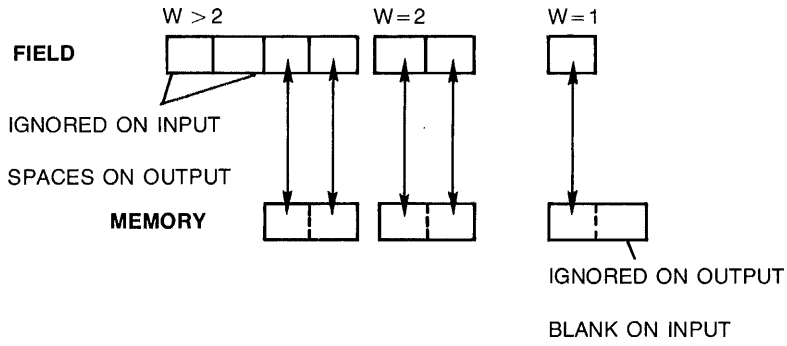
Output: If w ≥ 6, six octal digits are written right-justified in the field. (Leading positions are filled with blanks.) If w<6, the w least significant (rightmost) digits are written.

4-12. A SPECIFICATION: ASCII

Format: Aw
w = field width

Input: If w ≥ 2, the rightmost two characters are taken from the field. (The first w−2 characters are ignored.) If w = 1, one character is read and stored in the left half of a computer word; blank is stored in the right half.

Output: If w ≥ 2, two characters are written in the field with w−2 leading blanks. If w=1, the character in the left half of the word is written.



The A1 and A2 format for character I/O correspond to ASCII constants with a blank included as the second character when w= 1.

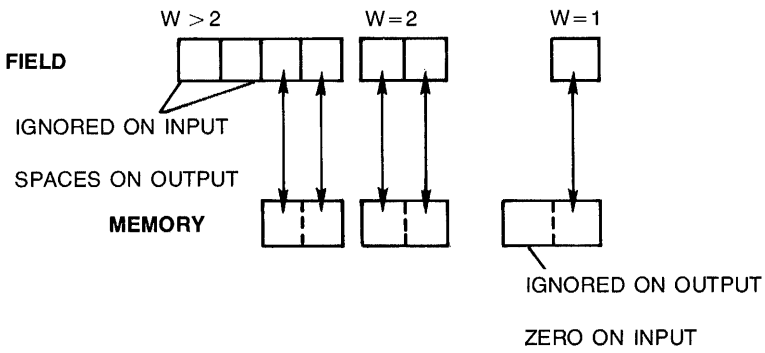
4-13. R SPECIFICATION: ASCII, RIGHT-JUSTIFIED

Format: R w
w = field width

R specifications are the same as A specifications when w > 1.

Input: When w= 1, one character is read and stored in the right half of a computer word. Binary zero is stored in the left half.

Output: When w= 1, the character in the right half of the word is written.



The R format corresponds to ASCII constant format.

4-14. L SPECIFICATION: LOGICAL (BOOLEAN)

Format: L w

w = width of field

Input: The input field may contain leading blanks, but the first non-blank character must be a T (for TRUE) or F (for FALSE). Any characters may follow the T or F.

WARNING

The L specification converts an input T into the FORTRAN .TRUE. = −32767 (octal 100000), not the ALGOL TRUE = −1. (Note that the sign bit is set correctly.) F converts to FALSE.

Output: If the internal value is negative, the character T is printed as the rightmost character in the field. Otherwise the character F is printed. The left w−1 characters are blanks.

4-15. EDITING SPECIFICATIONS

In addition to real and integer number specifications, you can use editing specifications in FORMAT declarations.

4-16. STRING SPECIFICATION

Format: " c_1c_2 ... c_w "
w = field width (number of characters)
c_i = any ASCII character except "

Input: The number of characters within the quotes are skipped on the input record.

Output: The characters between the quotes are written.

WARNING

If you omit the closing quote, the compiler will read all the text of your program up to the next quote as part of the character string. If the program ends before a second quote is read, the compiler will expect more input.

4-17. HOLLERITH SPECIFICATION

Format: w H c_1c_2 ... c_w
w = field width (default = 1)
c_i = any ASCII character

This specification is similar to the string specification. The w characters following the H are considered part of the Hollerith string. A quote can be included. You must have exactly w characters following the H as part of the Hollerith string.

Input: The characters in the external field replace the characters in the field specification.

Output: The characters in the field are written.

Examples:

```
29HTHE " SPECIFICATION IS EASIER
35HYOU MUST COUNT CHARACTERS CORRECTLY
```

4-18. X SPECIFICATION

Format: wX
w = field width (default = 1)

Input: w characters are skipped.

Output: w blanks are placed in the output field.

This specification allows you to separate fields.

4-19. SPECIFICATION SEPARATORS

The specifications in a FORMAT declaration must be separated from one another by a comma (,) or slash (/). The comma acts only as a character which separates two specifications. The slash terminates a record. A series of slashes causes records to be skipped on input or lines to be skipped on output.

Example:

```
FORMAT F1(3X, "X=", F7.2, 3X, "Y=", F7.2/"2*X+Y=", F10.2),
       F2(//1X,1PE10.3,0PF6.4///I5,@4//);
```

4-20. REPEAT COUNT

You can repeat real and integer specifications or a slash several times in a format specification by placing an integer constant (the repeat count) before the E, F, G, I, @, K, O, A, R, L, or /. For example,

```
3I3,2F5.2,3/1P2E10.3
```

is equivalent to

```
I3,I3,I3,F5.2,F5.2,///1PE10.3,E10.3
```

You can repeat groups of specifications by enclosing them in parentheses and placing a repeat count in front.

Examples:

```
2(2(F7.2,3X),F5.1)
4( 5("*"), 3X)
```

4-21. CARRIAGE CONTROL

If you are using a line printer as the output device, the first character of each line is used to determine how the printer spaces before it prints a line. The following characters have special meanings when they appear in column 1:

0	double space
1	eject page
*	suppress spacing (overprint)
blank	single spage

Any other character is treated like a blank. When the first character is used for carriage control, it is not printed.

If your output device is a terminal, when the last character is a ← or _ the normal carriage return is supressed; the next I/O operation begins where the _ would have been printed. This is useful in asking questions:

```
WRITE(1, #("WHAT IS THE LU NUMBER?_"));
READ(1, *, LISTLU);
```

4-22. FREE FIELD INPUT

Your program can read numeirc input without format specifications if you place an asterisk (*) instead of a format identifier in a READ statement. When your program reads free field input, it recognizes input values by scanning for the characters

+ or -	sign of item
.	decimal point
E	scaling factor
@	octal integer
" . . . "	comments
/	record terminator
0 to 9	digits
space or ,	delimiter

All other characters are treated as separators between data items.

DATA ITEMS. Real and integer values are represented in the same form as E and I formatted input. Octal values are represented in the same form as octal constants within the program. Any characters between quotes are ignored.

DATA ITEM DELIMITERS. During input, you usually separate data items by a space or comma. Any of the characters not listed above can also be used. The data items are read and transferred to the corresponding variables in the input list. Two consecutive commas indicate that no value is supplied for the corresponding list variable; its current value remains unchanged. (An initial comma causes the first variable to be skipped.)

RECORD TERMINATOR. A slash within a record causes the next record to be read; the remainder of the current record is skipped. If a record terminates and a slash has not been read, the input operation terminates even if all the input elements have not been assigned new values.

4-23. READ STATEMENT

The READ statement transfers values from an I/O device to program variables. The general form of the READ statement is

READ (<unit>, <format part>, <input list>)

The unit is an integer arithmetic expression which designates an I/O logical unit number.

The format part can be

the name of a format identifier

the symbol # followed by format specifications enclosed in parentheses (in-line format)

the symbol *, indicating free field input.

If the format part is left out, an unformatted record consisting of binary values is read. Consult the driver manual of your device for its binary I/O format.

The input list can be any of the input elements allowed in the INPUT specification. If the input list is omitted, a record is skipped.

For formatted input, when a READ statement is executed, one record is read. Additional records are read only as indicated by the format specification. Any unprocessed characters on the current record are ignored. If there are fewer input elements than format specifications, the unused specifications are ignored. If there are more elements than specifications, a new record is read and format control continues with the group of specifications terminated by the last preceding right parenthesis.

4-24. WRITE STATEMENT

The WRITE statement transfers internal values to an I/O device. The general form of the WRITE statement is:

WRITE (<unit>, <format part>, <output list>)

The unit is an integer arithmetic expression which designates an I/O logical unit number.

The format part can be

the name of a format identifier

the symbol # followed by format specifications enclosed in parentheses (in-line format).

(There is no free field output.)

If the format part is left out, an unformatted record consisting of binary values is written. Consult the driver manual of your device for its binary I/O format.

The output list can be any of the output elements allowed in the OUTPUT declaration. If the format specifications don't require any values, the output list is not necessary.

For formatted output, when a WRITE statement is executed the values are printed according to the corresponding format specifications. The specifications are interpreted from left to right. If there are fewer values to be printed than specifications, the extra specifications are ignored. If format control continues to the last right parenthesis and more values are to be printed, a new line is started and format control continues with the group repeat specification terminated by the last preceding right parenthesis.

4-25. EXAMPLES

These programs have been written to show the use of FORMAT specifications and I/O statements.

The first program shows the effect of a scaling factor. The four values read were all typed as 3212.5, but have different internal values due to different scaling factors.

In the second part of the program, values are printed using different scaling factors and different real specifications, all with the same basic field width.

```
PAGE  001

001 00000 HPAL,L,"FMTS"
002 00000 BEGIN
003 00001 INTEGER I;
004 00003 REAL X1,X2,X3,X4;
005 00013
006 00013 OUTPUT XS(X1,X2,X3,X4),
007 00031        X1S(X1,X1,X1,X1);
008 00047
009 00047 FORMAT
010 00047  FMT1(   E15.5,   1PE15.5,   2PE15.5,   3PE15.5,   4PE15.5),
011 00073  FMT2(1X,F14.3,1X,1PF14.3,1X,2PF14.3,1X,3PF14.3,1X,4PF14.3),
012 00126  FMT3(   G15.5,   1PG15.5,   2PG15.5,   3PG15.5,   4PG15.5);
013 00152
014 00152 READ(1, FMT1, X1,X2,X3,X4);
015 00220 WRITE(6, #(///"0NUMBERS READ:"));
016 00242 WRITE(6, #(5(1X,F14.3)), XS);
017 00261 WRITE(6, FMT1, XS);
018 00270 WRITE(6, FMT2, XS);
019 00277
020 00277 WRITE(6, #(/"07.5 ^ I"));
021 00315 FOR I := 1 STEP 5 UNTIL 20 DO
022 00323    BEGIN
023 00323    X1 := 7.5^I;
024 00331    WRITE(6, #("0I =, I3), I);
025 00350    WRITE(6, FMT1, X1S);
026 00357    WRITE(6, FMT2, X1S);
027 00366    WRITE(6, FMT3, X1S);
028 00375    END;
029 00401 END$

PROGRAM= 000405  ERRORS=000
```

Program output:

```
NUMBERS READ:
      3212.500         321.250          32.125           3.213
    .32125E+04    3.21250E+02    32.1250E+00    321.250E-02
      3212.500       3212.500       3212.500       3212.500


7.5 ^ I

I =  1
    .75000E+01    7.50000E+00    75.0000E-01    750.000E-02
         7.500          75.000         750.000        7500.000
    7.5000         7.5000         7.5000         7.5000

I =  6
    .17798E+06    1.77979E+05    17.7979E+04    177.979E+03
    177978.531    1779785.312   17797853.125  177978531.250
    .17798E+06    1.77979E+05    17.7979E+04    177.979E+03

I = 11
    .42235E+10    4.22351E+09    42.2351E+08    422.351E+07
4223513599.998 42235135999.98 422351359999.7 4223513599997.
    .42235E+10    4.22351E+09    42.2351E+08    422.351E+07

I = 16
    .10023E+15    1.00226E+14    10.0226E+13    100.226E+12
$$$$$$$$$$$$$ $$$$$$$$$$$$$ $$$$$$$$$$$$$ $$$$$$$$$$$$$
    .10023E+15    1.00226E+14    10.0226E+13    100.226E+12
```

The second program generates values for two arrays, then calls a procedure to print each array. The procedure is designed to print any possible values from any sized N by N array. The E12.4 format is used because it allows real values of any magnitude to be printed with four significant digits. Five fields are printed per line. (If the array is larger than 5 by 5, more than one line is used for each row.)

```
PAGE  001

001 00000 HPAL,L,"ARRIO"
002 00000 BEGIN
003 00001 COMMENT READ AND WRITE ARRAYS;
004 00001 INTEGER I,J;
005 00004 ARRAY A1[1:8,1:8], A2[1:3,1:3];
006 00242 FORMAT UP("1");
007 00245
008 00245    PROCEDURE PRINTARRAY(A,N);
009 00246       VALUE N; ARRAY A; INTEGER N;
010 00246       BEGIN  & PRINT AN N BY N ARRAY
011 00252       INTEGER I,J; &INDEX VARIABLES
012 00255       OUTPUT ARRAYOUT(FOR J:=1 TO N DO A[I,J]);
013 00324       FORMAT ARRAYFMT( (1P5E12.4) );
014 00332       FOR I:=1 TO N DO
015 00340          BEGIN
016 00340          WRITE(6, ARRAYFMT, ARRAYOUT);
017 00347          WRITE(6, #(" "));
018 00361          END;
019 00365       END OF PRINTARRAY;
020 00366
021 00366 FOR I := 1 TO 8 DO
022 00374    FOR J := 1 TO 8 DO
023 00402       A1[I,J] := I + J + I/J;
024 00443 FOR I := 1 TO 3 DO
025 00451    FOR J := 1 TO 3 DO
026 00457       A2[I,J] := I*J + I/J;
027 00521 WRITE(6,UP);
028 00527 PRINTARRAY(A1,8);
029 00533 WRITE(6,#(/////));
030 00546 PRINTARRAY(A2,3);
031 00552 END$

PROGRAM= 000556  ERRORS=000
```

Program output:

```
3.0000E+00  3.5000E+00  4.3333E+00  5.2500E+00  6.2000E+00
7.1667E+00  8.1429E+00  9.1250E+00

5.0000E+00  5.0000E+00  5.6667E+00  6.5000E+00  7.4000E+00
8.3333E+00  9.2857E+00  1.0250E+01

7.0000E+00  6.5000E+00  7.0000E+00  7.7500E+00  8.6000E+00
9.5000E+00  1.0429E+01  1.1375E+01

9.0000E+00  8.0000E+00  8.3333E+00  9.0000E+00  9.8000E+00
1.0667E+01  1.1571E+01  1.2500E+01

1.1000E+01  9.5000E+00  9.6667E+00  1.0250E+01  1.1000E+01
1.1833E+01  1.2714E+01  1.3625E+01

1.3000E+01  1.1000E+01  1.1000E+01  1.1500E+01  1.2200E+01
1.3000E+01  1.3857E+01  1.4750E+01

1.5000E+01  1.2500E+01  1.2333E+01  1.2750E+01  1.3400E+01
1.4167E+01  1.5000E+01  1.5875E+01

1.7000E+01  1.4000E+01  1.3667E+01  1.4000E+01  1.4600E+01
1.5333E+01  1.6143E+01  1.7000E+01



2.0000E+00  2.5000E+00  3.3333E+00

4.0000E+00  5.0000E+00  6.6667E+00

6.0000E+00  7.5000E+00 10.0000E+00
```

4-26. MAGNETIC TAPE STATEMENTS

You can perform five magnetic tape operations in HP ALGOL. In these statements the unit is an integer expression designating the magnetic tape logical unit number.

SPACE <unit>

Spaces the tape forward one record or causes one end-of-record gap.

BACKSPACE <unit>

Backspaces the tape one record.

ENDFILE <unit>

Prints an end-of-file mark.

REWIND <unit>

Returns the tape to the load point in auto mode.

UNLOAD <unit>

Puts the unit in local mode and rewinds the tape.

PROCEDURES

SECTION V

When you write an ALGOL program, you may find that sections of code performing the same operation appear in several different places. You can write one procedure to perform the computations and execute it from your program wherever it is needed.

Procedures may be written and compiled separately from your program. When you do this, procedures are similar to subroutines of FORTRAN and Assembly Language.

You can write procedures within your program. Internal procedures are more flexible than subroutines because they can reference identifiers declared for the block in which they are written.

Procedures must be declared before they are used. If procedure A calls procedure B, procedure B must be declared before procedure A (unless it is declared within procedure A). No procedure may be entered recursively, either implicitly or explicitly.

5-1. PARAMETERS

When you declare a procedure, you can specify parameters that will be passed when it is called. You include in the procedure heading a list of the formal parameters — the symbolic names for the items provided when the call is made. The formal parameters must be declared as to type in the procedure heading. Formal parameters can have any ALGOL declaration:

REAL	INTEGER ARRAY	LABEL
INTEGER	BOOLEAN ARRAY	SWITCH
BOOLEAN	REAL PROCEDURE	FORMAT
ARRAY	INTEGER PROCEDURE	INPUT
REAL ARRAY	BOOLEAN PROCEDURE	OUTPUT
	PROCEDURE	

Formal parameters are either called by value or called by reference. If you specify that a formal parameter is called by value, the parameter is computed when the procedure is called and this result is used as a local variable within the procedure. Otherwise, the address of the parameter is passed to the procedure. Only REAL, INTEGER and BOOLEAN variables or expressions (not entire arrays) can be passed by value. Only identifiers (not expressions or constants) can be passed by reference.

When you call a procedure, you can provide a list of actual parameters — the items which are substituted for the formal parameters in the procedure declaration.

Actual parameters called by value may be expressions. Actual parameters called by reference must be identifiers.

Value parameters are treated as local variables within the procedure. Assignments to these parameters have no effect on the value of the corresponding actual parameter.

Any modifications to formal parameters passed by reference affect the corresponding actual parameter.

Example:

```
      :
PROCEDURE CHANGE(X);
   REAL X;
   X := X + 1;
PROCEDURE NOCHANGE(X);
   VALUE X;REAL X;
   X := X + 1;
A := 0;
CHANGE(A);
NOCHANGE(A);
      :
```

After the procedure CHANGE is called, A has the value 1. A's value remains 1 after NOCHANGE is called because it is passed by value.

If an actual parameter is itself a procedure, all of its parameters must be called by value. (See the SUMS example in Section VII.)

5-2. PROCEDURE DECLARATIONS

A procedure declaration describes a process. The process is not executed until the procedure is called. A procedure declaration consists of two parts: the procedure heading, which gives the name of the procedure and describes any formal parameters, and the procedure body, which describes the process that takes place.

Example:

```
PROCEDURE TRANSPOSE(A,N);
   VALUE N; INTEGER N; ARRAY A;
   BEGIN
   COMMENT TRANSPOSE AN N BY N MATRIX;
   INTEGER I,J; & USED TO INDEX MATRIX
   REAL Z;      & HOLDS VALUE DURING TRANSFER
   FOR I := 1 TO N DO
      FOR J := I+1 TO N DO
         BEGIN
         Z := A[I,J];
         A[I,J] := A[J,I];
         A[J,I] := Z;
         END
   END OF TRANSPOSE;
```

This procedure's heading consists of the following parts:

1. PROCEDURE — The reserved word PROCEDURE.

2. TRANSPOSE — The procedure name. It must be a legal identifier.

3. `(A,N)`	The formal parameter part. The identifiers of the formal parameters must be enclosed in parentheses
4. `;`	A semicolon to terminate the first part.
5. `VALUE N;`	The reserved word VALUE followed by a list of any parameters passed by value. A semicolon terminates the list. (Parameters passed by reference are not listed in the procedure heading.)
6. `INTEGER N;ARRAY A;`	Specifications for each of the formal parameters. A semicolon separates identifiers of different types. The order of the identifiers is not important.

A procedure body can consist of a single statement or (as in this example) a block.

Much of the heading is not needed if no parameters are passed. For example, here is a procedure to put the next character read from logical unit 5 into the program variable CHAR.

```
PROCEDURE GETCHAR;
   BEGIN
   COMMENT GLOBAL VARIABLES USED BY GETCHAR:
      BUFFER    HOLDS INPUT TEXT
      COLUMN    POINTS TO CURRENT TEXT POSITION
      CHAR      CURRENT INPUT CHARACTER;
   IF COLUMN > 71 THEN
      BEGIN
      INTEGER I;
      READ(5, #(80R1), FOR I:=1 TO 80 DO BUFFER[I]);
      COLUMN := 1
      END;
   CHAR := BUFFER[COLUMN];
   COLUMN := COLUMN+1
   END OF GETCHAR;
```

The procedure body may refer to any formal parameters, local variables, and those identifiers which have been declared in the block containing the procedure.

5-3. CALLING PROCEDURES

You call a procedure by writing its name followed by any actual parameters enclosed in parentheses. The procedure call is treated by the compiler like any other ALGOL command.

For example, here are calls to the procedure defined in the previous subsection:

```
TRANSPOSE(TABLE,25);
GETCHAR;
```

The actual parameters must correspond in number and type to the formal parameters specified in the procedure declaration.

5-4. FUNCTION PROCEDURES

A function procedure is a procedure that returns a single value. For function procedures, the word PROCEDURE is preceded by the type of the value that is returned (REAL, INTEGER, or BOOLEAN). You use the procedure name as a simple variable within the function. The value returned is the last value assigned to the procedure name.

Example:

```
BOOLEAN PROCEDURE ALPHANUMERIC(CH);
   INTEGER CH;
   ALPHANUMERIC := (CH >= "A"  AND  CH <= "Z") OR
                   (CH >= "0"  AND  CH <= "9");
```

This function returns the value TRUE when the parameter, CH, is an alphabetic or numeric character.

You call function procedures by using them as you would use an expression. The function's value is treated like the value of a variable of the same type.

Example:

```
IF ALPHANUMERIC(CHAR) THEN TOKEN:=7;
```

5-5. CODE PROCEDURES

Procedures may be compiled or assembled separately from the main ALGOL program. You can declare such procedures by including a procedure heading in your program and replacing the procedure body with the reserved word CODE. (All the parameters must be specified, as in regular procedures.)

Examples:

```
PROCEDURE INVER(A,X,N);
   VALUE N;INTEGER N;REAL A,X;
   CODE;
REAL PROCEDURE INTEGRAL(A,B,F);
   VALUE A,B;REAL A,B;REAL PROCEDURE F;
   CODE;
```

The names of CODE procedures have a maximum of five characters. Any characters beyond the fifth are ignored.

5-6. SEPARATELY COMPILED PROCEDURES

Any CODE procedures you specify in your program can be written in ALGOL, FORTRAN, or Assembly Language. The object code segments can be linked by the Relocating Loader before the program is executed.

5-7. ALGOL PROCEDURES

When you compile ALGOL procedures separately from a main program, you must use the "P" option in the HPAL control statement (Secton VI). The first line is the procedure declaration (not BEGIN). The procedure is terminated with END; (not END$).

Example:

```
PAGE  001

001 00000 HPAL,L,P,"SUM"
002 00000 INTEGER PROCEDURE SUM(V,N);
003 00002    INTEGER ARRAY V; INTEGER N;
004 00002    BEGIN  & ALGOL PROCEDURE TO SUM ELEMENTS V[1] ... V[N]
005 00006    INTEGER I;
006 00010    SUM := 0;
007 00037    FOR I:=1 TO N DO
008 00045       SUM := SUM + V[I]
009 00045    END;

PROGRAM= 000062  ERRORS=000
```

The control statement name in quotes is used as the NAM record. The name in the procedure declaration is used as the entry point. (The two should usually be the same.) For correct execution, the procedure heading must agree with the main program's CODE procedure heading as to the number of parameters and parameter types.

5-8. CALLING FORTRAN ROUTINES FROM ALGOL

You can call subroutines written in FORTRAN the same way you call ALGOL procedures: write the name of the subroutine followed by any parameters enclosed in parentheses. FORTRAN functions are treated similarly.

Arrays are stored differently in FORTRAN and ALGOL. The name of an array in FORTRAN references the first element of the array. In ALGOL, the name of an array references a description of the array (type, number of dimensions, bounds for each dimension, and starting address).

If you wish to call a FORTRAN routine that manipulates an array, specify the formal parameter (in the heading) as being REAL or INTEGER (depending on the type of the array) rather than as ARRAY or INTEGER ARRAY. The actual parameter (in the call) should be the first element of the array. Arrays should also be handled this way in Assembly Language routines that have been written to be called from FORTRAN.

Example:

```
FTN4,L
        INTEGER FUNCTION SUM(V,N)
        INTEGER V(1)
C   FORTRAN FUNCTION TO SUM ELEMENTS V(1) ... V(N)
        SUM = 0
        DO 10 I=1,N
        SUM = SUM + V(I)
  10    CONTINUE
        RETURN
        END
        END$
```

FORTRAN Function

```
HPAL,L,"MPRG1"
BEGIN
INTEGER RESULT, J;
INTEGER ARRAY A[1:20];
INTEGER PROCEDURE SUM(V,N);
   INTEGER V,N;
   CODE;
   .
   .
   .
RESULT := SUM(A[1], J);
   .
   .
   .
END$
```

ALGOL Main Program

The following is the WRONG WAY to call the FORTRAN routine:

```
   .
   .
   .
INTEGER PROCEDURE SUM(V,N);
   ARRAY V; INTEGER N;
   CODE;
   .
   .
   .
RESULT:=SUM(A,J)
   .
   .
   .
```

You can pass VALUE parameters to FORTRAN subroutines. However, the FORTRAN routine will not treat the parameters as being called by value. Changes to parameters which are not expressions will result in changes to the corresponding actual parameters in the main program. For this reason, you must be absolutly sure a FORTRAN subroutine does not modify constants passed as parameters.

5-9. CALLING ALGOL PROCEDURES FROM FORTRAN

ALGOL procedures compiled with the P option in the control statement can be called from FORTRAN. Arrays cannot be passed as parameters.

5-10. CALLING ALGOL PROCEDURES FROM ASSEMBLY LANGUAGE

If you want to call an ALGOL procedure from Assembly Language, you must provide the return address and the address of parameters.

Suppose you want to call the ALGOL procedure that has been compiled with the heading

```
PROCEDURE TEST(A,B,C,D,E);
   VALUE A,B;INTEGER A,C;REAL B,D;LABEL E;
```

In the Assembly Language program, you would write

```
        :
       EXT TEST        DECLARE TEST EXTERNAL
       JSB TEST        JUMP TO TEST
       DEF RTNPT       ADDRESS OF RETURN
       DEF PARM1       INTEGER VALUE PARAMETER
       DEF PARM2       REAL VALUE PARAMETER
       DEF PARM3       INTEGER PARAMETER
       DEF PARM4       REAL PARAMETER
       DEF LABL1       LABEL PARAMETER
RTNPT  EQU *           RETURN POINT
        :
LABL1  EQU *
        :
PARM1  BSS 1           STORAGE FOR INTEGER VALUE PARAMETER
PARM2  BSS 2           STORAGE FOR REAL VALUE PARAMETER
PARM3  BSS 1           STORAGE FOR INTEGER PARAMETER
PARM4  BSS 2           STORAGE FOR REAL PARAMETER
        :
```

This would be equivalent to calling TEST from ALGOL with

```
TEST(PARM1,PARM2,PARM3,PARM4,LABL1);
```

This is also the standard HP calling sequence for FORTRAN and Assembly Language. Value and reference parameters are passed in the same way; the difference is the way they are treated within the subroutine.

5-11. CALLING ASSEMBLY LANGUAGE ROUTINES FROM ALGOL

You can call Assembly Language routines from ALGOL if they have been written to pass the correct parameter values and addresses.

When you write Assembly Language routines to be called from ALGOL, you can obtain parameters from the calling sequence shown above or you can use the standard ALGOL parameter processing routine .PRAM.

The general calling sequence for .PRAM is

```
JSB .PRAM
<code words (maximum = 7)>
<storage for parameters>
```

The code words tell .PRAM the number of parameters, indicate which parameters are called by reference and which are called by value, and specify whether value parameters are real or integer. Code words have the following format:

First code word:

bits 15 through 10		Number of parameters (maximum= 52)
9	8	Bit pair for first parameter
7	6	Bit pair for second parameter
•	•	
•	•	
•	•	
1	0	Bit pair for fifth parameter

Second code word:

bits 15 through 14		Bit pair for sixth parameter
•	•	
•	•	
•	•	
1	0	Bit pair for 13th parameter

Similarly, code word 3 contains bit pairs for parameters 14-21, code word 4 contains bit pairs for parameters 22-29, etc.

Each parameter's bit pair has the following meaning: The left bit is 1 if the parameter is called by value, 0 if it is called by reference. The right bit is not used for parameters called by reference; for value parameters it is 1 for real variables and 0 for integer (or Boolean) variables.

Following the code words, you must reserve exactly enough words to store the address or value of each parameter. Real parameters passed by value require two words; all others require one word.

.PRAM also places the return address in the word preceding the JSB .PRAM instruction.

Example:

If the procedure TEST given above were written in Assembly Language, the entry portion could be

```
TEST  NOP             ENTRY POINT
      JSB .PRAM
      OCT 013300      CODE WORD
A     BSS 1           VALUE (INTEGER)
B     BSS 2           VALUE (REAL)
C     BSS 1           REFERENCE (INTEGER)
D     BSS 1           REFERENCE (REAL)
E     BSS 1           REFERENCE (LABEL)
```

.PRAM places the values of A and B and the addresses of C, D, and E in the locations provided. Within the subroutine you can refer to A and B directly. C and D should be referenced indirectly. If you want to transfer control to label E (an address in the calling program) you would write

```
JMP E,I
```

To return to the calling program you would write

```
JMP, TEST,I
```

You can use the Relocatable Library routines .INDA (get address) or .INDR (get value) to access ALGOL array elements from Assembly Language. ALGOL maintains a table for each array as follows:

```
TABLE DEC number of indices (+ = real, - = integer)
      DEC size of 1st dimension
      DEC -lower bound of 1st dimension
      DEC size of 2nd dimension
      DEC -lower bound of 2nd dimension
       :
      DEC size of last dimension
      DEC -lower bound of last dimension
      DEF starting address of array elements
```

When you pass the name of an array to a subroutine, you are passing the address of the first word in the array table.

To call the indexing routines use the following code:

```
JSB .INDA  (or .INDR)
DEF array table
DEC -number of indices
DEF first subscript value
 :
DEF last subscript value
```

If you call .INDA, the address of the array element is placed in the A-Register. For .INDR, the value of the array element is placed in the A-Register (for integer arrays) or the A- and B-Registers (for real arrays). For both routines, if the subscript values are not within their bounds, the INDEX? error message is printed and the routine returns a zero.

For example, here is the subroutine SUM coded in Assembly Language:

```
 PAGE 0002 #01

0001                    ASMB,L,R
0002  00000                    NAM SUM,7
0003*  INTEGER FUNCTION CALLED FROM ALGOL WITH
0004*           SUM(V,N);
0005*  TO SUM ELEMENTS V[1] ... V[N]  WHERE V IS AN
0006*  INTEGER ARRAY AND N IS AN INTEGER VARIABLE.
0007                    ENT SUM
0008  00000 000000  SUM  NOP              ENTRY POINT
0009                    EXT .PRAM
0010  00001 016001X     JSB .PRAM        PICK UP PARAMETERS
0011  00002 004000      OCT 004000          CODE WORD
0012  00003 000000  V   BSS 1               ADDRESS OF ARRAY V
0013  00004 000000  N   BSS 1               ADDRESS OF N
0014*
0015*  INITIALIZE SUM1 AND FOR LOOP COUNTER
0016  00005 062033R     LDA ZERO
0017  00006 072031R     STA SUM1
0018  00007 062034R     LDA ONE
0019  00010 072032R     STA I            I := 1
0020*
0021*  FOR I:=1 TO N DO SUM1:=SUM1+V[I];
0022  00011 003004  TEST  CMA,INA
0023  00012 142004R     ADA N,I          CHECK FOR END
0024  00013 002020      SSA                OF FOR LOOP
0025  00014 026027R     JMP ENDFR            (I>N)
0026*
0027                    EXT .INDR
0028  00015 016002X     JSB .INDR        PUT
0029  00016 100003R     DEF V,I           V[I]
0030  00017 177777      DEC -1             IN
0031  00020 000032R     DEF I               A-REGISTER
0032  00021 042031R     ADA SUM1
0033  00022 072031R     STA SUM1         SUM1 := SUM1 + V[I]
0034*
0035  00023 062032R     LDA I
0036  00024 002004      INA              INCREMENT I FOR NEXT
0037  00025 072032R     STA I              ITERATION AND CONTINUE
0038  00026 026011R     JMP TEST             IN FOR LOOP
0039*
0040  00027 062031R ENDFR LDA SUM1        PUT SUM IN A-REGISTER
0041  00030 126000R     JMP SUM,I        RETURN TO CALLING ROUTINE
0042*
0043  00031 000000  SUM1  BSS 1
0044  00032 000000  I     BSS 1
0045  00033 000000  ZERO  DEC 0
0046  00034 000001  ONE   DEC 1
0047                    END

**  NO ERRORS *TOTAL **RTE ASMB 750420**
```

THE HP ALGOL COMPILER

SECTION VI

This section has been written to explain some of the features and requirements of the HP ALGOL compiler.

6-1. ENVIRONMENT

HP ALGOL is a one pass compiler. It exists in two versions:

SIO System — The HP ALGOL compiler provides off-line compilation in BCS based systems. A tape punch is required to provide simultaneous punching of an object tape with listing for one pass compilation.

RTE-II/RTE-III/DOS — The HP ALGOL compiler provide on-line compilation in interactive or batch mode. It uses 8K words of background area in RTE and comparable space in DOS. The relocatable programs produced by this HP ALGOL compiler can execute in DOS, RTE-II, RTE-III, and RTE memory based systems.

6-2. CONTROL STATEMENT

The first statement the ALGOL compiler reads must be an HPAL control statement. If the first four characters of the first record are anything other than HPAL, the error message HPAL?? is printed on the system console and the compiler halts.

You can use any of the following options in the control statement:

L — list source program
A — list octal code produced for each statement
B — produce object tape
P — a procedure only is to be compiled
S — sense switch register (SIO only)

Any options must be separated by commas.

The name of the program enclosed in quotes also appears in the control statement. (This name becomes the NAM record name. It must be a legitimate identifier.) For DOS and RTE, the name appears after any options; for SIO it must be listed first (after HPAL).

Examples:

```
HPAL,L,B,P,"NAME"        (DOS/RTE)
HPAL,"NAME",L,B,P        (SIO)
```

For SIO, when the S option is used the B, L, and A options are read by the compiler from the switch register. (The P option must still be placed in the control statement if it is used.) If the associated bit in the switch register is on, the following options are in effect:

Bit	Option
15	B
14	L
13	A

The switch register is read at the beginning of each line. Any option may be turned on or off partially through a compilation.

If no options are specified for a compilation, the compiler only produces diagnostic messages.

6-3. PROGRAM INPUT

The ALGOL compiler reads records from the logical unit you specify when you run the compiler. No matter how long the record is, the compiler only uses the first 72 characters. (You can use columns 73-80 on cards for identification or sequence numbers.)

The compiler accepts blank lines. However, some I/O drivers read a blank record as an end-of-file indicator and signal that no more data is expected. (If disc input is used, do not include null lines. Make sure each line contains at least one character.)

The compiler senses the end of a program in two ways: when all the BEGINs are matched with ENDs or when a $ is read after an END. If your program contains an error, the BEGIN-END pairs may not be matched and compilation may terminate before the entire source program is read.

6-4. PROGRAM LISTING

When you use the L option in the HPAL statement, the program is listed as it is read. In addition, two numbers are printed to the left of each line: the sequence number of each source line and the relative octal address of the object code produced for that line. The relative address is added to the start address printed by the loader to give the absolute address of each line in your program. This address can be useful if you have run-time errors.

When you compile with option A, the object code listing consists of the octal code and equivalent assembly language statements generated for each program statement. The first octal number is the relative location in memory that will contain the object code. The second number either gives the octal instruction code (if an instruction is placed in memory) or a numeric value (for constants). The equivalent assembly language statements are listed to the right of the octal code.

This partial listing shows the effect of the L and A options:

```
001 00000 HPAL,L,A,"DMT"
002 00000 BEGIN
  00000  000000                          NOP
003 00001 INTEGER ARRAY NAME[1:5]:="TR" "IB" "BY";
  00001                                 BSS  000001
  00002  177777    NAME                 OCT  177777   }
  00003  000005                          OCT  000005   } Array Table
  00004  177777                         OCT  177777   }
  00005  000006 R                       DEF  *+1      }
  00006  052122                         OCT  052122   }
  00007  044502                         OCT  044502   } Storage for Array
  00010  041131                         OCT  041131   }
  00011                                 BSS  000002   }
004 00013 INTEGER CHAR;
  00013             CHAR                BSS  000001
005 00014 WHILE CHAR#" " DO

  00001                                 ORG  000001
  00001  026014 R                       JMP  000014
  00014                                 ORR

  00014  062013 R                       LDA  CHAR
  00015  026042 R                       JMP  000042
  00016                                 BSS  000024
  00042  042016 R                       ADA  000016
  00043  002003                         SZA,RSS
006 00044    BEGIN
  00044                                 BSS  000001
007 00045    READ(1, #(R1), CHAR);
  00045  062017 R                       LDA  000017
  00046  006404                         CLB,INB
  00047  014001 X                       JSB  .DIO.
  00050  000053 R                       DEF  *+3
  00051                                 BSS  000001
  00052                                 BSS  000001
  00053  024122                         ASC  1,(R
  00054  030451                         ASC  1,1)
```

PROGRAM EXAMPLES

SECTION VII

This section contains ALGOL programs that were compiled and executed under the RTE-III operating system.

7-1. TAYLOR SERIES FOR EXP, SIN, AND COS

[If you do not have a mathematical background, you may want to skip this example.]

One method of evaluating logarithmic and trigonometric functions by polynomials is to use Taylor series. (If you do not know what Taylor series are, consult a basic Calculus textbook.) Three well known Taylor series are

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}$$

$$\sin x = \sum_{k=0}^{\infty} \frac{(-1)^k\, x^{2k+1}}{(2k+1)!}$$

$$\cos x = \sum_{k=0}^{\infty} \frac{(-1)^k\, x^{2k}}{(2k)!}$$

(Σ denotes summation and ! denotes the factorial function.)

All three of these infinite sums can be approximated by summations to a number of terms. The greater the number, the closer the result is to the true value.

The following program evaluates the series for 5, 10, and 30 terms. It also prints the values of the ALGOL intrinsic function for EXP, SIN, and COS.

The first part of the listing contains eight procedures that return real values when called. Lines 9-15 define a procedure (FACTL) to calculate factorials. Lines 20-28 define procedures that evaluate terms for the sums shown above. The procedures in lines 33-41 are included so intrinsics can be passed as arguments to the PRINTVALUE procedure. The last real procedure (SUM) uses the terms in lines 20-28 to calculate a sum to a given number of terms. Most of the work of the program takes place in PRINTVALUES; it writes a heading, increments X from −2 to 2, and prints three sums for each value of X. The actual call to the SUM procedure appears in the OUTPUT statement (line 63). The main program, which begins in line 81, is rather trivial. It prints headings and calls PRINTVALUES for each different summation.

```
001 00000 HPAL,L,"SUMS"
002 00000 BEGIN
003 00001 COMMENT PROGRAM TO CALCULATE VALUES FOR EXP(X), SIN(X),
004 00001         AND COS(X) BY TAYLOR SERIES WITH 5, 10, AND 30
005 00001         TERMS, AND VALUES OF X BETWEEN -2 AND 2;
006 00001 &
007 00001 &
008 00001 &
009 00001 REAL PROCEDURE FACTL(N);
010 00005    VALUE N; INTEGER N;
011 00005    BEGIN  &CALCULATE THE FACTORIAL OF N (N!)
012 00010    INTEGER I;
013 00012    FACTL := 1.0;
014 00043    FOR I := 2 TO N DO FACTL := FACTL * I;
015 00063    END;
016 00066 &
017 00066 &
018 00066 &
019 00066 & PROCEDURES TO CALCULATE TERMS FOR EXP, SIN, AND COS
020 00066 REAL PROCEDURE EXPSUM(X,K);
021 00071    VALUE X,K; REAL X; INTEGER K;
022 00071    EXPSUM := (X^K)/FACTL(K);
023 00120 REAL PROCEDURE SINSUM(X,K);
024 00123    VALUE X,K; REAL X; INTEGER K;
025 00123    SINSUM := (-1)^K*X^(2*K+1)/FACTL(2*K+1);
026 00177 REAL PROCEDURE COSSUM(X,K);
027 00202    VALUE X,K; REAL X; INTEGER K;
028 00202    COSSUM := (-1)^K*X^(2*K)/FACTL(2*K);
029 00301 &
030 00301 &
031 00301 &
032 00301 & DUMMY PROCEDURES SO INTRINSICS CAN BE PARAMETERS
033 00301 REAL PROCEDURE EXP1(X);
034 00304    VALUE X; REAL X;
035 00304    EXP1 := EXP X;
036 00317 REAL PROCEDURE SIN1(X);
037 00322    VALUE X; REAL X;
038 00322    SIN1 := SIN X;
039 00335 REAL PROCEDURE COS1(X);
040 00340    VALUE X; REAL X;
041 00340    COS1 := COS X;
```

```
042 00353 &
043 00353 &
044 00353 &
045 00353 REAL PROCEDURE SUM(K1,X,FUNCT);
046 00356    VALUE K1,X; INTEGER K1; REAL X; REAL PROCEDURE FUNCT;
047 00356    BEGIN  & SUM UP TERMS IN SERIES
048 00364    INTEGER K;
049 00366    SUM := 0.0;
050 00372    FOR K := 0 TO K1 DO
051 00400       SUM := SUM + FUNCT(X,K);
052 00414    END;
053 00417 &
054 00417 &
055 00417 &
056 00417 &
057 00417 PROCEDURE PRINTVALUES(INTRINSIC, FUNCTION);
058 00420    REAL PROCEDURE INTRINSIC, FUNCTION;
059 00420    BEGIN  & CALCULATE AND OUTPUT VALUES
060 00424    INTEGER ARRAY N[1:3] := 5, 10, 30;
061 00434    INTEGER I; REAL X;
062 00437    OUTPUT SUMS
063 00437     (X,INTRINSIC(X),FOR I:=1 TO 3 DO SUM(N[I],X,FUNCTION));
064 00475    FORMAT NUMBERS(F7.2, 1P4E14.5),
065 00505     HEADING("0   X",7X,"INTRINSIC",8X,"SUM(5)",8X,"SUM(10)"
066 00535                    7X,"SUM(30)");
067 00544    &
068 00544    WRITE(6,HEADING);
069 00552    X := -2.0;
070 00556    WHILE X <= 2.0 DO
071 00563       BEGIN
072 00564       WRITE(6, NUMBERS, SUMS);
073 00573       X := X + .33333;
074 00601       END;
075 00602    END;
076 00603 &
077 00603 &
078 00603 &
079 00603 & BEGINNING OF MAIN PROGRAM
080 00603 &
081 00603 WRITE(6, #("1EVALUATION OF INTRINSICS BY SERIES"
082 00634            //"0INTRINSIC 1 = EXP"));
083 00651 PRINTVALUES(EXP1,EXPSUM);
084 00655 WRITE(6, #(/"0INTRINSIC 2 = SIN"));
085 00700 PRINTVALUES(SIN1,SINSUM);
086 00704 WRITE(6, #(/"0INTRINSIC 3 = COS"));
087 00727 PRINTVALUES(COS1,COSSUM);
088 00733 END$

PROGRAM= 000737  ERRORS=000
```

EVALUATION OF INTRINSICS BY SERIES

INTRINSIC 1 = EXP

X	INTRINSIC	SUM(5)	SUM(10)	SUM(30)
-2.00	1.35335E-01	6.66667E-02	1.35379E-01	1.35335E-01
-1.67	1.88875E-01	1.64951E-01	1.88881E-01	1.88875E-01
-1.33	2.63595E-01	2.57063E-01	2.63596E-01	2.63595E-01
-1.00	3.67876E-01	3.66663E-01	3.67876E-01	3.67876E-01
-.67	5.13410E-01	5.13299E-01	5.13410E-01	5.13410E-01
-.33	7.16519E-01	7.16517E-01	7.16519E-01	7.16519E-01
-.00	9.99980E-01	9.99980E-01	9.99980E-01	9.99980E-01
.33	1.39558E+00	1.39558E+00	1.39558E+00	1.39558E+00
.67	1.94768E+00	1.94755E+00	1.94768E+00	1.94768E+00
1.00	2.71820E+00	2.71658E+00	2.71820E+00	2.71820E+00
1.33	3.79354E+00	3.78396E+00	3.79354E+00	3.79354E+00
1.67	5.29429E+00	5.25564E+00	5.29429E+00	5.29429E+00
2.00	7.38876E+00	7.26638E+00	7.38870E+00	7.38876E+00

INTRINSIC 2 = SIN

X	INTRINSIC	SUM(5)	SUM(10)	SUM(30)
-2.00	-9.09297E-01	-9.09296E-01	-9.09298E-01	-9.09298E-01
-1.67	-9.95408E-01	-9.95408E-01	-9.95408E-01	-9.95408E-01
-1.33	-9.71940E-01	-9.71940E-01	-9.71940E-01	-9.71940E-01
-1.00	-8.41476E-01	-8.41477E-01	-8.41477E-01	-8.41477E-01
-.67	-6.18381E-01	-6.18381E-01	-6.18381E-01	-6.18381E-01
-.33	-3.27211E-01	-3.27211E-01	-3.27211E-01	-3.27211E-01
-.00	-2.03848E-05	-2.03848E-05	-2.03848E-05	-2.03848E-05
.33	3.27172E-01	3.27172E-01	3.27172E-01	3.27172E-01
.67	6.18348E-01	6.18349E-01	6.18349E-01	6.18349E-01
1.00	8.41455E-01	8.41455E-01	8.41455E-01	8.41455E-01
1.33	9.71930E-01	9.71930E-01	9.71930E-01	9.71930E-01
1.67	9.95412E-01	9.95411E-01	9.95412E-01	9.95412E-01
2.00	9.09314E-01	9.09313E-01	9.09315E-01	9.09315E-01

INTRINSIC 3 = COS

X	INTRINSIC	SUM(5)	SUM(10)	SUM(30)
-2.00	-4.16147E-01	-4.16155E-01	-4.16147E-01	-4.16147E-01
-1.67	-9.57270E-02	-9.57279E-02	-9.57270E-02	-9.57270E-02
-1.33	2.35231E-01	2.35231E-01	2.35231E-01	2.35231E-01
-1.00	5.40294E-01	5.40294E-01	5.40294E-01	5.40294E-01
-.67	7.85879E-01	7.85879E-01	7.85879E-01	7.85879E-01
-.33	9.44951E-01	9.44951E-01	9.44951E-01	9.44951E-01
-.00	1.00000E+00	1.00000E+00	1.00000E+00	1.00000E+00
.33	9.44965E-01	9.44965E-01	9.44965E-01	9.44965E-01
.67	7.85904E-01	7.85904E-01	7.85904E-01	7.85904E-01
1.00	5.40328E-01	5.40328E-01	5.40328E-01	5.40328E-01
1.33	2.35270E-01	2.35270E-01	2.35270E-01	2.35270E-01
1.67	-9.56867E-02	-9.56876E-02	-9.56867E-02	-9.56867E-02

7-2. READ TEXT AND COUNT CHARACTERS

This program shows how to work with characters. All ASCII constants in the program are one character in the right half of their word, with the left half filled with zeroes. This corresponds to the R1 format for reading and writing. If the A1 format were used for I/O, all the ASCII constants would have to include a blank as the right character.

Note that ASCII constants can be used as array subscripts and as control values in a FOR statement. The compiler converts them to their octal equivalent.

```
PAGE  001

001 00000 HPAL,L,"COUNT"
002 00000 BEGIN
003 00001 &
004 00001 COMMENT  READ TEXT FROM TERMINAL AND COUNT EACH CHARACTER;
005 00001 &
006 00001 INTEGER I;
007 00003 INTEGER ARRAY CHAR[" ":"_"],&HOLDS COUNT FOR EACH CHARACTER
008 00107                TEXT[1:80];   &HOLDS INPUT CHARACTERS
009 00233 BOOLEAN DONE := FALSE;
010 00234 &
011 00234 & INITIALIZE CHARACTER COUNTS TO ZERO
012 00234 FOR I := " " TO "_" DO CHAR[I] := 0;
013 00302 &
014 00302 & WRITE HEADING AND READ TEXT
015 00302 WRITE(6, #("1INPUT TEXT:"/));
016 00322 WHILE NOT DONE DO
017 00325    BEGIN
018 00326    READ(1, #(80R1), FOR I:=1 TO 80 DO TEXT[I]);
019 00360    IF TEXT[1] = "/" THEN DONE := TRUE
020 00367     ELSE
021 00372       BEGIN
022 00372       WRITE(6, #(1X,80R1), FOR I:=1 TO 80 DO TEXT[I]);
023 00425       FOR I := 1 TO 80 DO
024 00433          CHAR[TEXT[I]] := CHAR[TEXT[I]] + 1
025 00456       END
026 00464    END;
027 00465 &
028 00465 & LIST CHARACTERS USED AND CHARACTER COUNT
029 00465 WRITE(6, #("0CHARACTERS USED:"));
030 00507 FOR I := " "+1 TO "_" DO
031 00516    IF CHAR[I]#0 THEN WRITE(6, #(3X,R1,I6), I,CHAR[I])
032 00546 END$

PROGRAM= 000557  ERRORS=000
```

Program Output:

```
INPUT TEXT:

THIS IS A TEST TO SEE IF THE PROGRAM "COUNT" WORKS. THE
PROGRAM READS TEXT, PRINTS IT, THEN COUNTS THE NUMBER OF
TIMES EACH CHARACTER APPEARS.  THE END OF TEXT IS SIGNALED
BY A "/" IN COLUMN 1. AFTER ALL THE TEXT IS READ, THE
CHARACTER COUNTS ARE PRINTED.

CHARACTERS USED:
  "      4
  ,      3
  .      4
  /      1
  1      1
  A     17
  B      2
  C      9
  D      5
  E     26
  F      4
  G      3
  H     11
  I     11
  K      1
  L      4
  M      5
  N     11
  O     10
  P      6
  R     17
  S     14
  T     27
  U      5
  W      1
  X      3
  Y      1
```

7-3. CALL SYSTEM ROUTINES

This program calls several routines provided with the RTE system:

RMPAR — a routine to pick up run-time parameters from the RUN, ON or GO command. You pass as a parameter the first of five consecutive words. After executing RMPAR, these five words contain the run-time parameters. Because the routine uses the B-register, you cannot pass an element of an array as the parameter. Instead, declare five consecutive integers and pass the first of these as the parameter.

EXEC — the standard RTE system executive routine that can be used for many different functions. You can pass from 1 to 9 parameters to EXEC. However, ALGOL programs must declare a fixed number of parameters for each procedure. One solution to this conflict is to define dummy external routines for each EXEC call that requires a different number of parameters (EXEC2 for 2 parameters, EXEC3 for 3 parameters, etc.). An example of an assembly language routine to patch in a call to the real EXEC routine is shown after the main program.

ABREG — A routine that returns the A- and B-registers in the two integer parameters passed. This routine is useful to pick up information left in the registers by EXEC.

The program first picks up the logical unit number for the console from RMPAR. It then gets the current time from EXEC and uses EXEC to read the user's name. ABREG obtains the length of the name from the B-register. The main loop of the program prints status information for logical unit numbers read from the console.

```
PAGE  001

001 00000 HPAL,L,"STATS"
002 00000 BEGIN
003 00001 &
004 00001 COMMENT THIS PROGRAM PERFORMS EXEC CALLS REQUIRING 2, 3,
005 00001         AND 4 PARAMETERS;
006 00001 &
007 00001 INTEGER LEN := 20,     & NAME LENGTH FOR READ
008 00003         LU,            & LOGICAL UNIT NUMBER
009 00004         EQT5,          & EQT WORD 5 STORAGE
010 00005         KEYBD,         & LU FOR CONSOLE
011 00006         P2,P3,P4,P5,   & USED FOR RMPAR
012 00012         A, B,          & VALUES OF A- AND B-REGISTERS
013 00014         I;             & COUNTER
014 00015 &
015 00015 INTEGER ARRAY TIME[1:5],NAME[1:20];
016 00056 FORMAT TIMEFMT
017 00056    ("/STATUS: CURRENT TIME: DAY",I4,3X,3(I2,":"),I2);
018 00107 OUTPUT TIMEVAL(FOR I:= 5 STEP -1 UNTIL 1 DO TIME[I]);
019 00155 &
020 00155 &
021 00155 & DUMMY EXEC ROUTINES
022 00155 PROCEDURE EXEC2(P1,P2);
023 00156    VALUE P1;INTEGER P1,P2;
024 00156    CODE;
025 00155 PROCEDURE EXEC3(P1,P2,P3);
026 00156    VALUE P1,P2;INTEGER P1,P2,P3;
027 00156    CODE;
028 00155 PROCEDURE EXEC4(P1,P2,P3,P4);
029 00156    VALUE P1,P2;INTEGER P1,P2,P3,P4;
030 00156    CODE;
031 00155 &
032 00155 & SYSTEM ROUTINE TO PICK UP RUN-TIME PARAMETERS
033 00155 PROCEDURE RMPAR(P1);
034 00156    INTEGER P1;
035 00156    CODE;
036 00155 &
037 00155 & SYSTEM ROUTINE TO GET VALUES OF A- AND B-REGISTERS
038 00155 PROCEDURE ABREG(A,B);
039 00156    INTEGER A,B;
040 00156    CODE;
041 00155 &
042 00155 &
043 00155 &
044 00155 & BEGINNING OF PROGRAM
045 00155 &
046 00155 &  PICK UP KEYBOARD LU VIA RMPAR
047 00155 RMPAR(KEYBD);
048 00160 IF KEYBD < 1 THEN KEYBD := 1;
049 00166 &
050 00166 &  GET TIME AND PRINT MESSAGE
051 00166 EXEC2(11,TIME[1]);
052 00177 WRITE(KEYBD, TIMEFMT, TIMEVAL);
```

```
053 00206 &
054 00206 &  GET USER'S NAME (LENGTH GOES IN B REGISTER)
055 00206 WRITE(KEYBD, #("/STATUS: WHAT IS YOUR NAME?"));
056 00235 P2 := KEYBD+@400;
057 00240 EXEC4(1,P2,NAME[1],LEN);
058 00253 ABREG(A,B);
059 00257 LEN := B;
060 00261 LU := KEYBD;
061 00263 &
062 00263 & PROMPT FOR LU AND PRINT STATUS WORD
063 00263 WHILE LU > 0 DO
064 00266    BEGIN
065 00267    EXEC3(13, LU, EQT5);
066 00274    WRITE(KEYBD,#("/STATUS: EQT5 OF DEVICE",I3," = ",K6),
067 00326                  LU,EQT5);
068 00333    WRITE(KEYBD, #("/STATUS: NEXT LU (END = 0):_"));
069 00362    READ(KEYBD, *, LU);
070 00371    END;
071 00372 &
072 00372 & GET TIME AND PRINT ENDING MESSAGE
073 00372 EXEC2(11,TIME[1]);
074 00403 WRITE(KEYBD, TIMEFMT, TIMEVAL);
075 00412 WRITE(KEYBD,#("/STATUS: GOODBYE ",20A2), FOR I:=1 TO LEN DO
076 00451                                              NAME[I]);
077 00455 END$

PROGRAM= 000461  ERRORS=000
```

The following assembly language subroutine is called by a jump to any of the dummy EXEC routines. It stores a JSB to the true EXEC routine over the dummy jump in the main program, then jumps back and executes the EXEC call. When the original call is executed again, it points to the true EXEC routine.

```
 PAGE 0002 #01

0001                    ASMB,L,R
0002  00000                    NAM EXEC0,7
0003** USED TO FIX ALGOL EXEC CALLS
0004                           EXT EXEC
0005                           ENT EXEC1,EXEC2,EXEC3,EXEC4,EXEC5
0006                           ENT EXEC6,EXEC7,EXEC8,EXEC9
0007  00000             EXEC1  EQU *
0008  00000             EXEC2  EQU *
0009  00000             EXEC3  EQU *
0010  00000             EXEC4  EQU *
0011  00000             EXEC5  EQU *
0012  00000             EXEC6  EQU *
0013  00000             EXEC7  EQU *
0014  00000             EXEC8  EQU *
0015  00000             EXEC9  EQU *
0016*
0017  00000 000000  EXEC0 NOP           ENTRY POINT
0018  00001 072011R       STA SAVE      SAVE A-REGISTER
0019  00002 003400        CCA
0020  00003 042000R       ADA EXEC0     A-REG POINTS TO EXEC CALL
0021  00004 072000R       STA EXEC0
0022  00005 062012R       LDA JSBEX     LOAD A-REG WITH REAL EXEC CALL
0023  00006 172000R       STA EXEC0,I   REPLACE JSB INSTRUCTION
0024  00007 062011R       LDA SAVE      RESTORE A-REGISTER
0025  00010 126000R       JMP EXEC0,I   RETURN
0026*
0027  00011 000000  SAVE   BSS 1         SAVE AREA FOR A-REG
0028  00012 016001X JSBEX  JSB EXEC      REPLACING INSTRUCTION
0029                       END

**  NO ERRORS *TOTAL **RTE ASMB 750420*
```

Output from program:

```
/STATUS: CURRENT TIME: DAY  91   11:10:13:55
/STATUS: WHAT IS YOUR NAME?
DAVE
/STATUS: EQT5 OF DEVICE  7 = 002400
/STATUS: NEXT LU (END = 0):2
/STATUS: EQT5 OF DEVICE  2 = 014400
/STATUS: NEXT LU (END = 0):1
/STATUS: EQT5 OF DEVICE  1 = 000000
/STATUS: NEXT LU (END = 0):6
/STATUS: EQT5 OF DEVICE  6 = 005000
/STATUS: NEXT LU (END = 0):0
/STATUS: CURRENT TIME: DAY  91   11:10:51:34
/STATUS: GOODBYE DAVE
```

HP ALGOL AND HP FORTRAN IV

SECTION VIII

This section has been written for the experienced FORTRAN programmer who wants to program in ALGOL. If you do not already know FORTRAN, you may skip this discussion.

Complete descriptions of ALGOL statements are not given here. Consult the proper section of this manual for further information.

This section compares HP FORTRAN IV and HP ALGOL. Comparison between other versions of FORTRAN and ALGOL is not implied.

8-1. PROGRAM FORMAT

Unlike FORTRAN statements, which have a comment column, a statement number field, a continuation column, and a statement field, ALGOL statements can begin in any column (they must end by column 72) and run for as many lines as you need. Statements are separated by semicolons (;). Several statements can appear on the same line.

Spaces act as delimiters in ALGOL. As in FORTRAN, extra spaces are ignored.

8-2. VARIABLES AND CONSTANTS

FORTRAN's real and integer number formats are the same as ALGOL's. ALGOL does not use double precision or complex numbers. ALGOL's Boolean values are similar to FORTRAN's logical values.

Any FORTRAN integer number is legal in ALGOL. ALGOL's real constants use ' for scale factors in place of E. Hollerith constants are enclosed in " instead of being preceded by 1H or 2H. The logical constants in ALGOL are TRUE and FALSE (not .TRUE. and .FALSE.). Octal constants are preceded by @ instead of being followed by B. (Any sign goes in front of the @.)

There are no default type declarations in ALGOL. Each variable must be declared explicitly before it is used. The identifiers shown in table 2-1 may not be used except as noted.

Only alphanumeric and numeric characters (not $) may be used in ALGOL identifiers, and embedded blanks are not allowed. Identifiers must begin with an alphabetic character and can continue for as many characters as you need.

8-3. ARRAYS

Arrays can have as many dimensions as you need. The bounds of each subscript, which must be defined when you declare the array, can be any integer constants, positive or negative. (The lower bound does not default to 1.)

When referencing an array element, use brackets, not parentheses, around the subscript expression. During execution of an ALGOL program, the subscript is checked every time an array is referenced. If it is not within the specified bounds, an error message is printed.

8-4. STATEMENT NUMBERS

ALGOL statements are not numbered. However, you may reference a statement with a label. Labels are defined by placing the label (any legal identifier) and a colon before the statement.

8-5. EXPRESSIONS

The FORTRAN exponentiation operator ** is replaced in ALGOL by ^. All other FORTRAN operators have the same meaning in ALGOL, except / always produces a real result. Integer division is performed with \.

The FORTRAN logical operators .OR., .AND., and .NOT. are replaced in ALGOL with OR, AND, and NOT.

The FORTRAN relational operators and their equivalent ALGOL symbols are

.LT. <	.EQ. =	.GT. >
.LE. <=	.NE. #	.GE. >=

The precedence of operators in arithmetic and logical expressions is the same in FORTRAN and ALGOL. In mixed mode expresions, the conversions between real and integer results within the expression may be different for the two languages.

8-6. EXTERNAL STATEMENT

Each external function and subroutine in ALGOL must be declared as a regular procedure. The instructions for the routine are replaced by the word CODE.

8-7. COMMON AND EQUIVALENCE STATEMENTS

There are no statements in ALGOL which perform the functions of FORTRAN's COMMON and EQUIVALENCE declarations. (ALGOL's EQUATE declaration is completely different from FORTRAN's EQUIVALENCE.)

8-8. DATA STATEMENT

There is no DATA statement in ALGOL. However, you can assign initial values to variables and arrays as you declare them.

8-9. ASSIGNMENT STATEMENTS

ALGOL assignments are the same as in FORTRAN except the symbols := or ← replace FORTRAN's =.

8-10. GO TO STATEMENT

The FORTRAN and ALGOL GO TO statements are equivalent. (In ALGOL, GO TO must be two words.) Because ALGOL provides many ways to control program execution, you can usually write entire programs without one GO TO statement.

8-11. ASSIGN TO AND ASSIGNED GO TO STATEMENTS

ALGOL does not have a construct exactly the same as the ASSIGN TO statement; they are not needed because labels can be passed to subroutines as parameters. The ASSIGNED GO TO statement of FORTRAN is similar to an ALGOL GO TO with a switch as the destination.

8-12. COMPUTED GO TO STATEMENT

An ALGOL GO TO statement with a switch as the destination is similar to FORTRAN's computed GO TO. The ALGOL CASE statement can also be used in much the same way.

8-13. ARITHMETIC IF STATEMENT

There is no ALGOL statement directly equivlent to the arithmetic IF statement. You can, however, use the ALGOL IF statement in the following way:

```
E := <expression>;
IF E<0 THEN GO TO <statement 1 label>
   ELSE IF E=0 THEN GO TO <statement 2 label>
      ELSE GO TO <statement 3 label>;
```

A better solution using blocks instead of GO TO statements can usually be found.

8-14. LOGICAL IF STATEMENT

ALGOL's IF statement is similar to FORTRAN's, but it also allows an ELSE clause.

8-15. CALL STATEMENT

There is no need for a CALL statement in ALGOL. You invoke a subroutine by writing its name as a statement. As in FORTRAN, any parameters are enclosed in parenthesis following the subroutine name.

8-16. RETURN AND STOP STATEMENTS

RETURN and STOP statements are not needed in ALGOL. Control is returned to a main program or to the operating system when the final END is reached during execution.

8-17. CONTINUE STATEMENT

ALGOL dummy statements perform the same functions as FORTRAN's CONTINUE.

8-18. PAUSE STATEMENT

The PAUSE statement performs the same function in both languages. You cannot use an octal number with an ALGOL PAUSE.

8-19. DO STATEMENT

FORTRAN's DO statement is similar to ALGOL's FOR statement. (Do not confuse it with ALGOL's DO statement.) The statements in the range of a FORTRAN DO statement are always executed at least once. The FOR statement block is not executed when the initial value exceeds the final value. ALGOL's FOR statement is more powerful — it allows negative values for the control variable, terminal parameter, and step size.

8-20. END STATEMENT

The FORTRAN and ALGOL END statements are similar, but should not be considered as equivalent. ALGOL's END is not a statement, but is used with BEGIN to delimit blocks of code.

8-21. I/O STATEMENTS

FORTRAN and ALGOL use the same formatter for I/O. The FORMAT statements of each language are equivalent. Because ALGOL does not have double precision values, you cannot use D format specifications. All other format specifications are legal in ALGOL. Unformatted input and binary I/O are the same in ALGOL as in FORTRAN.

The ALGOL READ and WRITE statements perform the functions of their FORTRAN counterparts. ALGOL can write expressions (not just variables). You cannot output an entire ALGOL array by specifying its name. FORTRAN's implied DO list is replaced by FOR elements in ALGOL.

ALGOL has magnetic tape statements that are similar to FORTRAN's.

8-22. FUNCTIONS AND SUBROUTINES

ALGOL procedures are similar to FORTRAN's functions and subroutines. You can call ALGOL, FORTRAN, or Assembly Language routines from ALGOL. ALGOL provides additional control of parameters by allowing passage by value or reference. (FORTRAN passes parameters by reference only.)

ERRORS

APPENDIX A

A-1. COMPILER ERROR MESSAGES

If the first record of your source program is not an HPAL control statement, the compiler halts and prints HPAL?? on the system console.

Errors detected in the source program are indicated by a numeric error code. An arrow (↑) is printed below the symbol the compiler was processing when it discovered the error.

Sometimes one mistake in your program will cause numerous errors. For example, if you do not declare an identifier properly, an error message appears anywhere you reference the identifier in your program. A semicolon before an ELSE or UNTIL (in a DO statement) can cause the compiler to lose track of the blocks in your program.

Here are the compiler error codes and a description of the condition causing the error:

ERROR CODE	DESCRIPTION
1	More than two characters used in an ASCII constant.
2	@ not followed by an octal digit.
3	Octal constant greater than 177777.
4	Two decimal points in one number.
5	Non-integer following apostrophe in scale factor.
6	Label declared but not defined in program.
7	Number required but not present.
8	Missing END.
10	Undefined identifier.
11	Illegal symbol.
12	Procedure designator must be followed by left parenthesis.
13	Parameter types disagree.
14	Reference parameter must be a variable.
15	Parameter must be followed by a comma or right parenthesis.

ERROR CODE	DESCRIPTION
16	Too many parameters.
17	Too few parameters.
18	Array variable not followed by a left bracket.
19	Subscript must be followed by a comma or right bracket.
20	Missing THEN.
21	Missing ELSE.
22	Illegal assignment.
23	Missing right parenthesis.
24	Proper procedure not legal in arithmetic expression.
25	Primary may not begin with this type quantity.
26	Too many subscripts.
27	Too few subscripts.
28	Variable required.
40	Too many external symbols (maximum = 255)
41	Declaration following statement.
42	No parameters declared after left parenthesis.
43	REAL, INTEGER, or BOOLEAN illegal with this declaration.
44	Doubly defined identifier or reserved word.
45	Illegal symbol in declaration.
46	Statement started with undefined identifier or illegal symbol.
47	Label not followed by colon.
48	Label is previously defined.
49	Semicolon expected as terminator.

ERROR CODE	DESCRIPTION
50	Left arrow or := expected in SWITCH declaration.
51	Label entry expected in SWITCH declaration.
52	Real number assigned to integer in declaration.
53	Constant expected following left arrow or :=.
54	Left arrow or := expected in EQUATE declaration.
55	Left bracket expected in ARRAY declaration.
56	Integer expected in array dimension.
57	Colon expected in array dimension.
58	Upper array bound less than lower bound.
59	Right bracket expected at end of array dimensions.
60	Too many values for array initialization.
61	Array size excessive (32767). Set to 1024.
62	Constant expected in array initialization.
63	Too many parameters for procedure.
64	Right parenthesis expected at end of procedure parameter list.
65	Procedure parameter descriptor missing.
66	VALUE parameter for procedure not in list.
67	Illegal type in procedure declaration.
68	Illegal description in procedure declaratives.
69	Identifier not listed as procedure parameter.
70	No type for variable in procedure parameter list.
71	Semicolon found in FORMAT declaration.
72	Left parenthesis expected after I/O declaration name.
73	Right parenthesis expected after I/O name parameters.

ERROR CODE	DESCRIPTION
74	Undefined label reference.
75	SWITCH identifier not followed by left bracket.
76	Missing right bracket in SWITCH designator.
77	THEN missing in IF statement.
78	DO missing in WHILE statement.
79	FOR variable must be in integer.
80	FOR variable must be followed by left arrow or :=.
81	STEP missing in FOR clause.
82	UNTIL missing in FOR clause or DO statement.
83	DO missing in FOR clause.
84	Parenthesis expected in READ/WRITE statement.
85	Comma expected in READ/WRITE statement.
86	Free field format (*) illegal with WRITE.
87	Unmatched left bracket in I/O statement list.
88	Missing BEGIN in CASE statement.
89	Missing END in CASE statement.
100	Program must start with BEGIN, REAL, INTEGER or PROCEDURE.
999	Table areas overflowed.

A-2. RUN-TIME ERROR MESSAGES

When you run your ALGOL program, several kinds of errors may be reported.

WARNING

Integer or real value overflow may not be recognized during execution of arithmetic operations. Integers "wrap around" (from 32767 to −32768) and real numbers are assigned the largest value (1.70141′ 38).

A-3. RELOCATABLE LIBRARY MESSAGES

During execution of a relocatable library subroutine, any error messages consist of the calling program name, a two digit subroutine identifier, and a two character error type indicator. (See *DOS-RTE Relocatable Library Reference Manual,* Part Number 24998-90001, for further details.) The message is printed in the following form:

program name nn error type address

program name is the name of your program

nn is a number in the range 02-14 (decimal) that identifies the subroutine which generated the error.

error type is OF for integer or floating point overflow,
OR for out of range, or
UN for floating point underflow.

address is the absolute octal address of the memory location where the error occured.

These error messages can occur when ALGOL intrinsics are used or during an exponentiation operation. Suppose X and Y are real values and I and J are integers. Then the following relocatable subroutines are called for these computations: X^ Y .RTOR (real to real); X^ I .RTOI (real to integer); I^ J .ITOI (integer to integer).

ERROR CODE	SUBROUTINE	CONDITION
02 UN	LN	$X \leq 0$
03 UN	SQRT	$X < 0$
04 UN	.RTOR	$X=0$ and $Y \leq 0$ or $X<0$ and $Y \neq 0$
05 OR	SIN	$\left\lvert \frac{X}{\pi} + \frac{1}{2} \right\rvert > 2^{15}$
05 OR	COS	$\left\lvert \frac{X}{\pi} \right\rvert > 2^{15}$
06 UN	.RTOI	$X=0$, $I \leq 0$
07 OF	EXP	$X \geq \frac{124}{\log_2 e}$
07 OF	.RTOR	$\lvert X*LN(X) \rvert \geq 124$
08 OF	.ITOI	$I^J \geq 2^{23}$
08 UN	.ITOI	$I = 0, J \leq 0$
09 OR	TAN	$X > 2^{14}$

A-4. INDEX? DIAGNOSTIC MESSAGE

The message INDEX? appears during execution whenever an array is accessed with an invalid index. For DOS and RTE, the absolute address of the violation is printed and exection continues. Zero is returned as the value of the array element. For SIO and BCS, the computer halts with the address in the A-register.

A-5. I/O ERROR MESSAGES

During execution of your program, the following messages may be printed on the console. For SIO and BCS the computer halts and the code which further defines the error is contained in the A-register. For DOS and RTE, the error message is printed in a form similar to the Relocatable Library messages.

ERROR CODE	EXPLANATION AND ACTION
*EQR Unit Number	SIO only. Equipment error: end of input tape or tape supply low on tape punch. B-Register contains the status word of the equipment table entry. Place the next tape into the input device or load a new reel of tape. Press RUN.
FMT ERR 1	FORMAT specification error: a. w or d field does not contain digits. b. No decimal point after w field. c. $w-d \leq 4$ for E specification. Fix the error in your source code and compile your program again.
FMT ERR 2	FORMAT specifications are nested more than one level deep or FORMAT declaration contains unbalanced parentheses. Fix the error in your source code and recompile your program.
FMT ERR 3	FORMAT declaration contains an illegal character or a repetition factor of zero or specifies more characters than possible for the device. Fix the error in your source code and recompile your program.
FMT ERR 4	Illegal character read in fixed field input or number not right-justified in field. Check your data.
FMT ERR 5	An input number has an illegal form (such as two E's, two decimal points, two signs, etc.). Check your data.

HP ALGOL BNF SYNTAX

APPENDIX B

This appendix contains a precise definition of HP ALGOL in the formal metalanguage derived from the Backus-Naur Form (BNF) of syntax definition. The BNF notation consists of "productions" or syntax equations, each of which is in the following form:

<syntactic entity> ::= <syntactic expression>

This can be read as "the entity on the left is composed of the ordered collection of one or more of the expressions on the right."

If the entity has more than one expression, they are separated by a vertical bar. These expressions represent choices for any given expansion of the entity.

The bold face characters are the "terminal" symbols; the character itself appears in the entity.

The productions are listed in a "bottom-up" fashion, that is, the simplest elements of the language are listed first, then more complex elements are defined in terms of the simpler elements. If you are more familiar with a "top-down" description, read the definitions starting with the last one and work back to the first one.

Basic Symbols

<empty> ::= the null string of symbols

<character> ::= any single character

<letter> ::=**A** | **B** | **C** | **D** | **E** | **F** | **G** | **H** | **I** | **J** | **K** | **L** | **M** | **N** | **O** | **P** | **Q** | **R** | **S** | **T** | **U** | **V** | **W** | **X** | **Y** | **Z**

<octal digit> ::= **0**|**1**|**2**|**3**|**4**|**5**|**6**|**7**

<digit> ::= <octal digit>|**8**|**9**

<logical constant> ::= **TRUE** | **FALSE**

Identifiers

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

Numbers

<unsigned integer> ::= <digit>|<unsigned integer><digit>

<integer> ::= <unsigned integer>|+<unsigned integer>|−<unsigned integer>

<octal integer> ::= @<octal digit>|<octal integer><octal digit>

<ASCII constant> ::= "<character>"|"<character><character>"

<equate identifier> ::= <identifier>

<decimal fraction> ::= .<unsigned integer>

<exponent part> ::= '<integer>

<decimal number> ::= <unsigned integer>|<unsigned integer>.|
<decimal fraction>|<unsigned integer><decimal fraction>

<unsigned number> ::= <decimal number>|<exponent part>|
<decimal number><exponent part>

<number> ::= <unsigned number>|<octal integer>|<ASCII constant>|
<logical constant>|<equate identifier>

<signed number> ::= <number>|+<number>|-<number>

Variables

<variable identifier> ::= <identifier>

<simple variable> ::= <variable identifier>

<subscript expression> ::= <expression>

<subscript list> ::= <subscript expression>|<subscript list>,<subscript expression>

<array identifier> ::= <identifier>

<subscripted variable> ::= <array identifier>[<subscript list>]

<variable> ::= <simple variable>|<subscripted variable>

Function Designators

<procedure identifier> ::= <identifier>

<actual parameter> ::= <expression>|<identifier>

<actual parameter list> ::= <actual parameter>|<actual parameter list>,<actual parameter>

<actual parameter part> ::= <empty>|(<actual parameter list>)

<function designator> ::= <procedure identifier><actual parameter part>

Arithmetic Expressions

<adding operator> ::= + | -

<multiplying operator> ::= * | / | \ | **MOD**

<math intrinsic> ::= **ABS** | **SIGN** | **SQRT** | **SIN** | **COS** | **ARCTAN** | **TANH** | **LN** | **EXP** | **ENTIER** | **ROTATE** | **TAN**

<intrinsic> ::= **KEYS** |<math intrinsic><primary>

<primary> ::= <number>|<variable>|<function designator>|**(**<expression>**)**| <intrinsic>

<factor> ::= <primary>|<factor>**^**<primary>

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

<simple expression> ::= <term>|<adding operator><term>|
<simple expression><adding operator><term>

<relational operator> ::= < | <= | = | >= | > | #

<relation> ::= <simple expression>|
<simple expression><relational operator><simple expression>

<denial> ::= <relation>| **NOT** <denial>

<conjunction> ::= <denial>|<conjunction> **AND** <denial>

<disjunction> ::= <conjunction>|<disjuction> **OR** <conjunction>

<if clause> ::= **IF** <expression> **THEN**

<expression> ::= <disjunction>|<if clause><simple expression> **ELSE** <expression>|
(<variable><assignment operator><expression>**)**

Designational Expression

<label> ::= <identifier>

<switch identifier> ::= <identifier>

<designational expression> ::= <label>|<switch identifier>[<expression>]

Format Declarations

<format identifier> ::= <identifier>

<repeat count> ::= <unsigned integer>

<field width> ::= <unsigned integer>

<fraction width> ::= <unsigned integer>

<scale factor> ::= <unsigned integer>

<basic real element> ::= **E**<field width>.<fraction width>|
F<field width>.<fraction width>| **G**<field width>.<fraction width>

<real element> ::= <basic real element>|<repeat count><basic real element>|
<scale factor>**P**<basic real element>|
<scale factor>**P**<repeat count><basic real element>

<basic integer element> ::= **I**<field width>| **L**<field width>|**O**<field width>|
K<field width>| @<field width> | **A**<field width>|**R**<field width>| **X**

<integer element> ::= <basic integer element>|<repeat count><basic integer element>

<character string> ::= <character>|<character string><character>

<string element> ::= <field width>**H**<character string>|
"<character string (not containing **"**)>**"**

<element separator> ::= , | /

<format element> ::= <real element>|<integer element>|<string element>|
(<format list>**)**|<repeat factor>**(**<format list>**)**/|<repeat factor>/

<format list> ::= <format element>|<format list><element separator><format element>

<format segment> ::= <format identifier> **(**<format list>**)**

<format declaration> ::= **FORMAT** <format segment>|
<format declaration>,<format segment>

Procedure Declarations

<formal parameter> ::= <identifier>

<formal parameter list> ::= <formal parameter>|<formal parameter list>,<formal parameter>

<formal parameter part> ::= <empty>|**(**<formal parameter list>**)**

<identifier list> ::= <identifier>|<identifier list>,<identifier>

<value part> ::= <empty>| **VALUE** <identifier list>;

<specifier> ::= <type>| **ARRAY** | <type> **ARRAY** | **LABEL** | **SWITCH** |
PROCEDURE |<type> **PROCEDURE** | **FORMAT** | **INPUT** | **OUTPUT**

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

<procedure heading> ::= <procedure identifier><formal parameter part>;
<value part><specification part>

<procedure body> ::= <statement>| **CODE**

<procedure declaration> ::= **PROCEDURE** <procedure heading><procedure body>|
<type> **PROCEDURE** <procedure heading><procedure body>

Declarations

<integer type> ::= **INTEGER** | **BOOLEAN**

<type> ::= **REAL** |<integer type>

<initialized identifier> ::= <identifier><assignment operator><signed number>

<simple variable declaration> ::= <identifier>|<initialized identifier>

<type declaration> ::= <type><simple variable declaration>|
<type declaration>,<simple variable declaration>

<lower bound> ::= <signed number>

<upper bound> ::= <signed number>

<bound pair> ::= <lower bound>:<upper bound>

<bound pair list> ::= <bound pair>|<bound pair list>,<bound pair>

<array identifier list> ::= <array identifier>| <array identifier list>,<array identifier>

<array segment> ::= <array identifier list>[<bound pair list>]

<simple array list> ::= <array segment>| <simple array list>,<array segment>

<constant list> ::= <signed number>|<constant list>,<signed number>

<array list> ::= <simple array list>| <simple array list><assignment operator><constant list>

<array declaration> ::= **ARRAY** <array list>|<type> **ARRAY** <array list>

<label declaration> ::= **LABEL** <label>|<label declaration>,<label>

<switch declaration> ::= **SWITCH** <switch identifier><assignment operator><label>|
<switch declaration>,<label>

<equate declaration> ::= **EQUATE** <initialized identifier>|
<equate declaration>,<initialized identifier>

<input identifier> ::= <identifier>

<input list element> ::= <variable>|<input identifier>|
<for clause><input list element>|[<input list>]

<input list> ::= <input list element>|<input list>,<input list element>

<input segment> ::= <input identifier> (<input list>)

<input declaration> ::= **INPUT** <input segment>| <input declaration>,<input segment>

<output identifier> ::= <identifier>

<output list element> ::= <expression>|<output identifier>|
<for clause><output list element>|[<output list>]

<output list> ::= <output list element>| <output list>,<output list element>

<output segment> ::= <output identifier> (<output list>)

<output declaration> ::= **OUTPUT** <output segment>|
<output declaration>,<output segment>

<declaration> ::= <type declaration>|<array declaration>| <label declaration>|
<switch declaration>|<equate declaration>| <input declaration>|
<output declaration>|<format declaration>| <procedure declaration>

Basic Statements

<go to> ::= **GO TO** | **GO**

<go to statement> ::= <go to><designational expression>

<dummy statement> ::= <empty>

<procedure statement> ::= <procedure identifier><actual parameter part>

<pause statement> ::= **PAUSE**

<do statement> ::= **DO** <statement> **UNTIL** <expression>

<case head> ::= **CASE** <expression> **BEGIN** <statement>

<case body> ::= <case head>|<case body>;<statement>

<case statement> ::= <case body> **END**

<basic statement> ::= <go to statement>|<dummy statement>| <procedure statement>|
<pause statement>|<do statement>| <case statement>

Assignment Statements

<assignment operator> ::= ← | :=

<left part> ::= <variable><assignment operator>

<left part list> ::= <left part>|<left part list><left part>

<assignment statement> ::= <left part list><expression>

I/O Statements

<unit> ::= <expression>

<mag tape command> ::= **ENDFILE** | **REWIND** | **UNLOAD** | **SPACE** | **BACKSPACE**

<mag tape statement> ::= <mag tape command><unit>

<free-field part> ::= *

<inline format> ::= #(<format list>)

<output format part> ::= <format identifier>|<inline format>

<format part> ::= <free field part>|<output format part>

<read statement> ::= **READ(**<unit>,<format part>,<input list>**)**|
READ(<unit>,<format part>**)** | **READ(**<unit>,<input list>**)**

<write statement> ::= **WRITE(**<unit>,<output format part>,<output list>**)**|
WRITE(<unit>,<output format part>**)**| **WRITE(**<unit>,<output list>**)**

<I/O statement> ::= <mag tape statement>|<read statement>| <write statement>

Compound Statements, Blocks, and Programs

<step specification> ::= **STEP** <expression> **UNTIL** | **TO**

<for clause> ::= **FOR** <variable><assignment operator><expression>
<step specification><expression> **DO**

<while clause> ::= **WHILE** <expression> **DO**

<closed statement> ::= <basic statement>|<assignment statement>|
<I/O statement>|<while clause><closed statement>|
<if clause><closed statement> **ELSE** <closed statement> |
<for clause><closed statement>|<compound statement>|
<block>|<label>:<closed statement>

<open statement> ::= <if clause><statement>|
<if clause><closed statement> **ELSE** <open statement>|
<for clause><open statement>|<while clause><open statement>|
<label>:<open statement>

<statement> ::= <open statement>|<closed statement>

<block head> ::= **BEGIN** <declaration>;|<block head><declaration>;

<block body> ::= <block head><statement>|<block body>;<statement>

<block> ::= <blockbody> **END**

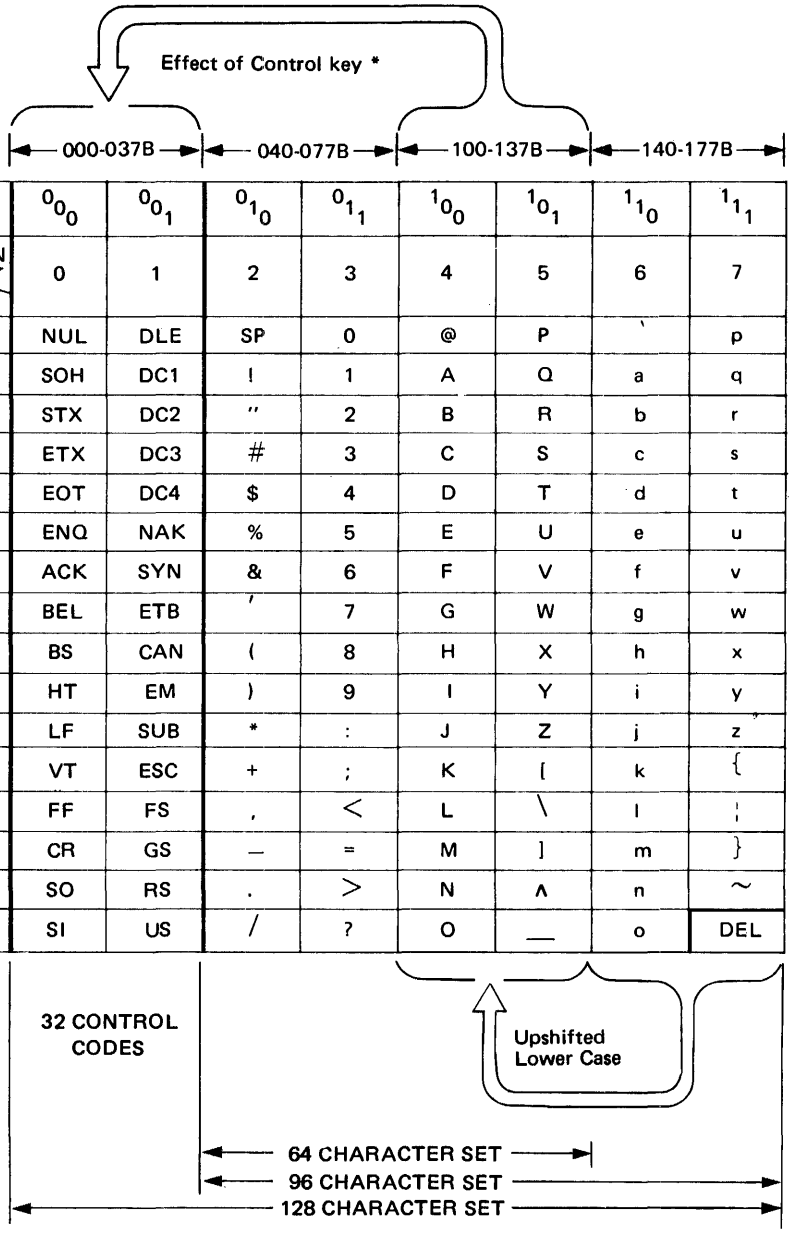
<compound head> ::= **BEGIN** <statement>|<compound head>;<statement>

<compound statement> ::= <compound head> **END**

<program> ::= <procedure declaration>;|<block>;|<compound statement>;

HP CHARACTER SET

APPENDIX C



b_7 b_6 b_5					0_{0_0}	0_{0_1}	0_{1_0}	0_{1_1}	1_{0_0}	1_{0_1}	1_{1_0}	1_{1_1}
BITS				COLUMN →	0	1	2	3	4	5	6	7
b_4	b_3	b_2	b_1	ROW ↓								
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	¦
1	1	0	1	13	CR	GS	-	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

EXAMPLE: The representation for the character "K" (column 4, row 11) is.

	b_7	b_6	b_5	b_4	b_3	b_2	b_1
BINARY	1	0	0	1	0	1	1
OCTAL	1		1			3	

* Depressing the Control key while typing an upper case letter produces the corresponding control code on most terminals. For example, Control-H is a backspace.

9206-1A

HEWLETT-PACKARD CHARACTER SET FOR COMPUTER SYSTEMS

This table shows HP's implementation of ANS X3.4-1968 (USASCII) and ANS X3.32-1973. Some devices may substitute alternate characters from those shown in this chart (for example, Line Drawing Set or Scandanavian font). Consult the manual for your device.

The left and right byte columns show the octal patterns in a 16 bit word when the character occupies bits 8 to 14 (left byte) or 0 to 6 (right byte) and the rest of the bits are zero. To find the pattern of two characters in the same word, add the two values. For example, "AB" produces the octal pattern 040502. (The parity bits are zero in this chart.)

The octal values 0 through 37 and 177 are control codes. The octal values 40 through 176 are character codes.

Decimal Value	Octal Values		Mnemonic	Graphic[1]	Meaning
	Left Byte	Right Byte			
0	000000	000000	NUL	$^{N}_{U}$	Null
1	000400	000001	SOH	$^{S}_{H}$	Start of Heading
2	001000	000002	STX	$^{S}_{X}$	Start of Text
3	001400	000003	ETX	$^{E}_{X}$	End of Text
4	002000	000004	EOT	$^{E}_{T}$	End of Transmission
5	002400	000005	ENQ	$^{E}_{Q}$	Enquiry
6	003000	000006	ACK	$^{A}_{K}$	Acknowledge
7	003400	000007	BEL		Bell, Attention Signal
8	004000	000010	BS	$^{B}_{S}$	Backspace
9	004400	000011	HT	$^{H}_{T}$	Horizontal Tabulation
10	005000	000012	LF	$^{L}_{F}$	Line Feed
11	005400	000013	VT	$^{V}_{T}$	Vertical Tabulation
12	006000	000014	FF	$^{F}_{F}$	Form Feed
13	006400	000015	CR	$^{C}_{R}$	Carriage Return
14	007000	000016	SO	$^{S}_{O}$	Shift Out } Alternate
15	007400	000017	SI	$^{S}_{I}$	Shift In } Character Set
16	010000	000020	DLE	$^{D}_{L}$	Data Link Escape
17	010400	000021	DC1	D_1	Device Control 1 (X-ON)
18	011000	000022	DC2	D_2	Device Control 2 (TAPE)
19	011400	000023	DC3	D_3	Device Control 3 (X-OFF)
20	012000	000024	DC4	D_4	Device Control 4 ($\overline{\text{TAPE}}$)
21	012400	000025	NAK	$^{N}_{K}$	Negative Acknowledge
22	013000	000026	SYN	$^{S}_{Y}$	Synchronous Idle
23	013400	000027	ETB	$^{E}_{B}$	End of Transmission Block
24	014000	000030	CAN	$^{C}_{N}$	Cancel
25	014400	000031	EM	$^{E}_{M}$	End of Medium
26	015000	000032	SUB	$^{S}_{B}$	Substitute
27	015400	000033	ESC	$^{E}_{C}$	Escape[2]
28	016000	000034	FS	$^{F}_{S}$	File Separator
29	016400	000035	GS	$^{G}_{S}$	Group Separator
30	017000	000036	RS	$^{R}_{S}$	Record Separator
31	017400	000037	US	$^{U}_{S}$	Unit Separator
127	077400	000177	DEL	▩	Delete, Rubout[3]

Decimal Value	Octal Values		Character	Meaning
	Left Byte	Right Byte		
32	020000	000040		Space, Blank
33	020400	000041	!	Exclamation Point
34	021000	000042	"	Quotation Mark
35	021400	000043	#	Number Sign, Pound Sign
36	022000	000044	$	Dollar Sign
37	022400	000045	%	Percent
38	023000	000046	&	Ampersand, And Sign
39	023400	000047	'	Apostrophe, Acute Accent
40	024000	000050	(	Left (opening) Parenthesis
41	024400	000051	)	Right (closing) Parenthesis
42	025000	000052	*	Asterisk, Star
43	025400	000053	+	Plus
44	026000	000054	,	Comma, Cedilla
45	026400	000055	-	Hyphen, Minus, Dash
46	027000	000056	.	Period, Decimal Point
47	027400	000057	/	Slash, Slant
48	030000	000060	0	} Digits, Numbers
49	030400	000061	1	
50	031000	000062	2	
51	031400	000063	3	
52	032000	000064	4	
53	032400	000065	5	
54	033000	000066	6	
55	033400	000067	7	
56	034000	000070	8	
57	034400	000071	9	
58	035000	000072	:	Colon
59	035400	000073	;	Semicolon
60	036000	000074	<	Less Than
61	036400	000075	=	Equals
62	037000	000076	>	Greater Than
63	037400	000077	?	Question Mark

Decimal Value	Octal Values		Character	Meaning
	Left Byte	Right Byte		
64	040000	000100	@	Commercial At
65	040400	000101	A	
66	041000	000102	B	
67	041400	000103	C	
68	042000	000104	D	
69	042400	000105	E	
70	043000	000106	F	
71	043400	000107	G	
72	044000	000110	H	
73	044400	000111	I	
74	045000	000112	J	
75	045400	000113	K	
76	046000	000114	L	
77	046400	000115	M	
78	047000	000116	N	Upper Case Alphabet, Capital Letters
79	047400	000117	O	
80	050000	000120	P	
81	050400	000121	Q	
82	051000	000122	R	
83	051400	000123	S	
84	052000	000124	T	
85	052400	000125	U	
86	053000	000126	V	
87	053400	000127	W	
88	054000	000130	X	
89	054400	000131	Y	
90	055000	000132	Z	
91	055400	000133	[	Left (opening) Bracket
92	056000	000134	\	Backslash, Reverse Slant
93	056400	000135	]	Right (closing) Bracket
94	057000	000136	^ ↑	Caret, Circumflex; Up Arrow[4]
95	057400	000137	_ ←	Underline; Back Arrow[4]

Decimal Value	Octal Values		Character	Meaning
	Left Byte	Right Byte		
96	060000	000140	`	Grave Accent[5]
97	060400	000141	a	
98	061000	000142	b	
99	061400	000143	c	
100	062000	000144	d	
101	062400	000145	e	
102	063000	000146	f	
103	063400	000147	g	
104	064000	000150	h	
105	064400	000151	i	
106	065000	000152	j	
107	065400	000153	k	
108	066000	000154	l	
109	066400	000155	m	
110	067000	000156	n	Lower Case Letters[5]
111	067400	000157	o	
112	070000	000160	p	
113	070400	000161	q	
114	071000	000162	r	
115	071400	000163	s	
116	072000	000164	t	
117	072400	000165	u	
118	073000	000166	v	
119	073400	000167	w	
120	074000	000170	x	
121	074400	000171	y	
122	075000	000172	z	
123	075400	000173	{	Left (opening) Brace[5]
124	076000	000174	¦	Vertical Line[5]
125	076400	000175	}	Right (closing) Brace[5]
126	077000	000176	~	Tilde, Overline[5]

9206-1C

Notes: [1]This is the standard display representation. The software and hardware in your system determine if the control code is displayed, executed, or ignored. Some devices display all control codes as "|", "@", or space.

[2]Escape is the first character of a special control sequence. For example, ESC followed by "J" clears the display on a 2640 terminal.

[3]Delete may be displayed as "__", "@", or space.

[4]Normally, the caret and underline are displayed. Some devices substitute the up arrow and back arrow.

[5]Some devices upshift lower case letters and symbols (` through ~) to the corresponding upper case character (@ through ^). For example, the left brace would be converted to a left bracket.

INDEX

READER COMMENT SHEET

02116-9072 Nov 1976

HP ALGOL

We welcome your evaluation of this manual. Your comments and suggestions help us improve our publications. Please use additional pages if necessary.

Is this manual technically accurate?

Is this manual complete?

Is this manual easy to read and use?

Other comments?

FROM:

Name _____

Company _____

Address _____

_____

_____

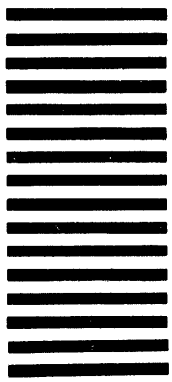
FOLD FOLD

FIRST CLASS
PERMIT NO. 141
CUPERTINO
CALIFORNIA

BUSINESS REPLY MAIL

No Postage Necessary if Mailed in the United States Postage will be paid by

Manager, Technical Publications
Hewlett-Packard Company
Data Systems Division
11000 Wolfe Road
Cupertino, California 95014



FOLD FOLD

PART NO. 02116-9072
Printed in U.S.A. 11/76

HEWLETT hp PACKARD

Sales and service from 172 offices in 65 countries.
11000 Wolfe Road, Cupertino, California 95014