60481600

CONTROL DATA CORPORATION

ALGOL-60 VERSION 5 REFERENCE MANUAL

CDC® OPERATING SYSTEMS
NOS 1
NOS/BE 1
SCOPE 2

REVISION RECORD	
REVISION	**DESCRIPTION**
A	Original release.
(1-9-79)	
B	Revised to reflect the release of ALGOL-60 Version 5.1. New features include sequenced input, a
(7-20-79)	reserved word option, and the VIRTUAL comment for declaring virtual arrays.
C	Implementation of ALGOL 5.1 under SCOPE 2.1 operating system. Miscellaneous technical corrections.
(8-22-79)	
Publication No. 60481600	

REVISION LETTERS I, O, Q AND X ARE NOT USED

Address comments concerning this manual to:

CONTROL DATA CORPORATION
Publications and Graphics Division
215 MOFFETT PARK DRIVE
SUNNYVALE, CALIFORNIA 94086

or use Comment Sheet in the back of this manual

LIST OF EFFECTIVE PAGES

New features, as well as changes, deletions, and additions to information in this manual are indicated by bars in the margins or by a dot near the page number if the entire page is affected. A bar by the page number indicates pagination rather than content has changed.

Page	Revision
Cover	C
Title Page	C
ii	C
iii/iv	C
v/vi	C
vii thru ix/x	C
xi	A
1-1	B
1-2	A
1-3	A
1-4	C
2-1	C
2-2	B
2-3	C
2-4	B
2-5	B
3-1	C
3-2	A
3-3	A
3-4	C
3-5	A
3-6	B
3-7 thru 3-10	A
4-1	A
4-2	B
4-3	C
4-4 thru 4-6	A
4-7	B
4-8	C
5-1	C
5-2 thru 5-4	B
6-1	B
6-2	A
6-3 thru 6-5	B
6-6	A
7-1	B
7-2	C

Page	Revision
7-3	C
7-4	B
8-1	A
8-2	B
8-3 thru 8-5	A
8-6	B
8-7 thru 8-10	A
8-11	B
8-12	A
8-13 thru 8-15	B
8-16 thru 8-22	C
9-1	A
9-2 thru 9-4	C
10-1	B
11-1 thru 11-3	C
11-4 thru 11-6	B
12-1	C
12-2	B
12-3	A
12-4	C
12-5	C
12-6	A
12-7	B
13-1	C
13-2	C
14-1	A
14-2	C
14-3 thru 14-5	A
15-1	C
15-2	A
15-3	C
15-4	C
A-1	C
A-2	B
A-3	C
B-1	A
B-2	C

Page	Revision
B-3	A
B-4	A
C-1 thru C-3	A
D-1 thru D-3	A
E-1 thru E-3	B
E-4	C
E-5	B
E-6	C
E-7	C
E-8 thru E-10	A
E-11	B
E-12	A
Index-1 thru -3	C
Comment Sheet	C
Mailer	–
Back Cover	–

PREFACE

ALGOL 5.1 is an implementation of the ALGOL-60 programming language. The version of the language that it implements is described in the following two publications:

"Modified Report on the Algorithmic Language ALGOL 60", Computer Journal, Volume 19, Number 4, November 1976, pages 364 through 379.

"A Proposal for Input-Output Conventions in ALGOL 60", Communications of the ACM, Volume 7, Number 5, May 1964.

For the most part, ALGOL 5 conforms to the standards defined in these documents. Any features permitted in ALGOL 5 but not addressed in these publications is indicated by shading. CDC implementations of features of the standard are not shaded; for example, separately compiled procedures and CDC-defined standard procedures.

ALGOL 5 operates under control of the following operating systems:

NOS 1 for the CONTROL DATA® CYBER 170 Series; CYBER 70 Models 71, 72, 73, 74; and 6000 Series Computer Systems.

NOS/BE 1 for the CDC® CYBER 170 Series; CYBER 70 Models 71, 72, 73, 74; and 6000 Series Computer Systems.

SCOPE 2 for the CDC 7600; CYBER 70 Model 76; and CYBER 170 Model 176 Computer Systems.

This manual is designed for programmers familiar with the ALGOL language and the operating system under which the ALGOL compiler executes.

Related material is contained in the following publications.

Publication	Publication Number
NOS Version 1 Reference Manual, Volume 1	60435400
NOS Version 1 Reference Manual, Volume 2	60445300
NOS/BE Version 1 Reference Manual	60493800
CYBER Record Manager Basic Access Methods Version 1.5 Reference Manual	60497500
COMPASS Version 3 Reference Manual	60492600
FORTRAN Extended Version 4 Reference Manual	60497800
SCOPE 2.1 Reference Manual	60342600
SCOPE 2 Record Manager Reference Manual	60454690
SCOPE 2 Loader Reference Manual	60454780

CDC manuals can be ordered from Control Data Corporation, Literature and Distribution Services, 308 North Dale Street, St. Paul, Minnesota 55103.

This product is intended for use only as described in this document. Control Data cannot be responsible for the proper functioning of undescribed features or parameters.

CONTENTS

APPENDIXES

INDEX

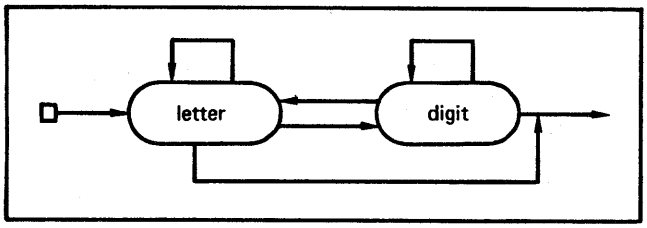
FIGURES

TABLES

NOTATIONS USED IN THIS MANUAL

The syntax of ALGOL entities shown in this manual is explained by means of drawings called state diagrams. A state diagram should be read like a flow chart. To construct a syntactically valid entity of the type defined, start at the left of the diagram and follow any path to an exit at the right. Whenever the path goes through an oval or rectangle, write down one of the entities described by the oval or rectangle. The ovals contain entities that are not further defined; the rectangles contain entities for which further state diagrams are provided or which are defined in the text. Since some of the arrows point to the left, some sub-entities can be indefinitely repeated within an entity.

Example:



State Diagram for Identifiers

According to this diagram, any of the following is a valid identifier:

A

ABCD

A12345

X9E8V7B6N5

Examples and syntax in this manual use graphics from the CDC character set. The ASCII equivalents can be found in appendix A. The symbol △ indicates a blank.

ALGOL 5 is an implementation of the ALGOL-60 programming language. ALGOL is an algorithmic language suited for writing programs to implement a wide variety of applications, especially mathematical and other scientific applications.

This section provides a brief overview of ALGOL through two different approaches: examination of a sample program and comparison of ALGOL with FORTRAN. It also describes the format of comments in an ALGOL program.

SAMPLE ALGOL PROGRAM

Figure 1-1 shows a complete ALGOL program, designed to find prime numbers through the familiar Eratosthenes' sieve method. The output is shown in figure 1-2. This particular program (named ERATO) finds all primes less than 1000, but it could be easily altered to find all primes less than any given number. The specific actions taken by the program cannot be explained without a detailed discussion of the features of ALGOL, but the general method used is as follows:

1. At the beginning of the program, all numbers from 2 through 1000 are considered candidates for primeness.

2. Beginning with 2, the program checks numbers until one is found that is still a candidate for primeness. Let N be this number. The first such number is 2.

3. Starting with N+N, every N^{th} number is eliminated from the list, since all such numbers are divisible by N. The first numbers eliminated are 4, 6, and so forth.

4. Steps 2 and 3 are repeated until only prime numbers less than 1000 remain.

Figure 1-1 also illustrates some facts about the format of an ALGOL program, and the way it is listed by the ALGOL 5 compiler:

- ALGOL is a free-format language. That is, the interpretation of an ALGOL program is not dependent on the number of characters per line or the columns of the source line in which certain entities fall. (This is in contrast to languages such as FORTRAN and COBOL.) The characters of an ALGOL program are grouped into larger entities such as statements, declarations, and specifications, which under certain circumstances are terminated by semicolons, but a single source line might contain more than one of these entities or only part of a single entity.

- The compiler provides a line number for each source line of a program numbered consecutively from 1. This number appears to the left of the line.

- To the right of the source line, the compiler indicates whether the line is part of a comment, and also the depth of nesting of blocks at that line. Comments are defined below, and block nesting in section 4.

- An installation, or any group of programmers, might define coding conventions to make programs easier to read. Typical conventions are that the ≠BEGIN≠ symbol of a block should begin in the same column as the corresponding ≠END≠ symbol, and that the other lines of that block should be indented. Other conventions might relate to the wording, location, and formatting of comments, or location of blanks.

- Blanks are ignored in an ALGOL program, except in the following circumstances:

  1. Blanks appearing in strings represent the blank character.

  2. Blanks are not permitted to interrupt the string quote symbols ≠(≠and≠)≠.

  3. In reserved word mode (RES option specified on ALGOL5 control statement), blanks delimit reserved words. See section 2.

  Elsewhere, blanks can be used freely to improve readability.

Table 1-1 lists some of the components of an ALGOL program, along with examples of where they can be found in program ERATO, and which section of the manual describes each component.

FORTRAN COMPARISON

Many programmers new to ALGOL have already programmed in FORTRAN, so a comparison of the two languages might help explain some of the characteristics of ALGOL.

The major differences between ALGOL and FORTRAN are as follows:

- FORTRAN: Subprograms are written and compiled independent of the main program, and require no declaration.

  ALGOL: Procedures (equivalent to subprograms) are declared within the main program, although their text can be compiled separately.

- FORTRAN: Each program unit consists of a series of individual statements.

  ALGOL: Statements can be grouped into larger units such as compound statements and blocks, each of which is bracketed by the ≠BEGIN≠ symbol and the ≠END≠ symbol. Since compound statements and blocks are also statements, they can appear anywhere a statement is allowed. This can result in the nesting of blocks to several levels.

- FORTRAN: No subprogram can reference itself, either directly or indirectly.

  ALGOL: A procedure can reference itself either directly or indirectly; this is called recursion.

```
        ERATO                    * SOURCE LISTING *            ALGOL  5.0 78318     11/18/78  14.15.51.     PAGE        1


      1.            ERATO:                                                                                    0  COMMENT
      2.            ≠BEGIN≠                                                                                   1
      3.            ≠COMMENT≠ FIND PRIME NUMBERS THROUGH ERATOSTHENES↑ SIEVE METHOD;                           1
      4.               ≠INTEGER≠ ≠PROCEDURE≠ NEXT(LAST,LIMIT,PRIME);                                          1
      5.                  ≠VALUE≠ LIMIT;                                                                      1
      6.                  ≠INTEGER≠ LAST,LIMIT; ≠BOOLEAN≠ ≠ARRAY≠ PRIME;                                      1
      7.                  ≠BEGIN≠                                                                             2
      8.                     ≠INTEGER≠ I; ≠BOOLEAN≠ FOUND;                                                    2
      9.                     FOUND := ≠FALSE≠;                                                                2
     10.                     ≠FOR≠ LAST := LAST + 1 ≠WHILE≠ ¬FOUND ∧ LAST ≠LE≠ LIMIT ≠DO≠                     2
     11.                        ≠IF≠ PRIME [LAST] ≠THEN≠ FOUND := ≠TRUE≠;                                     2
     12.                     ≠IF≠ FOUND ≠THEN≠                                                                2
     13.                     ≠BEGIN≠                                                                          3
     14.                        LAST := LAST - 1;                                                             3
     15.                        ≠FOR≠ I := LAST ≠STEP≠ LAST ≠UNTIL≠ LIMIT ≠DO≠                                3
     16.                           PRIME [I] := ≠FALSE≠;                                                      3
     17.                        NEXT := LAST;                                                                 3
     18.                     ≠END≠                                                                            2
     19.                     ≠ELSE≠ NEXT := 0;                                                                2
     20.                  ≠END≠;                                                                              1
     21.            ≠BOOLEAN≠ ≠ARRAY≠ NUMBS [1:1000];                                                         1
     22.            ≠INTEGER≠ LATEST,I;                                                                       1
     23.               ≠FOR≠ I := 1 ≠STEP≠ 1 ≠UNTIL≠ 1000 ≠DO≠                                                1
     24.                  NUMBS [I] := ≠TRUE≠;                                                               1
     25.               LATEST := 1;                                                                          1
     26.               ≠FOR≠ LATEST := NEXT (LATEST,1000,NUMBS) ≠WHILE≠ LATEST > 0 ≠DO≠                      1
     27.                  OUTPUT (61,≠(≠7ZD≠)≠,LATEST);                                                      1
     28.            ≠END≠ ERATO                                                                               0
PROGRAM LENGTH     0001208 WORDS
REQUIRED  CM 037000. CP   .539 SEC.
```

Figure 1-1. Sample ALGOL Program

```
  2     3     5     7    11    13    17    19    23    29    31    37    41    43    47    53    59
 61    67    71    73    79    83    89    97   101   103   107   109   113   127   131   137   139
149   151   157   163   167   173   179   181   191   193   197   199   211   223   227   229   233
239   241   251   257   263   269   271   277   281   283   293   307   311   313   317   331   337
347   349   353   359   367   373   379   383   389   397   401   409   419   421   431   433   439
443   449   457   461   463   467   479   487   491   499   503   509   521   523   541   547   557
563   569   571   577   587   593   599   601   607   613   617   619   631   641   643   647   653
659   661   673   677   683   691   701   709   719   727   733   739   743   751   757   761   769
773   787   797   809   811   821   823   827   829   839   853   857   859   863   877   881   883
887   907   911   919   929   937   941   947   953   967   971   977   983   991   997
```

Figure 1-2. Output from Sample Program

TABLE 1-1. SAMPLE PROGRAM COMPONENTS

Entity	Line Number	Example	Section
Special symbol	2	≠BEGIN≠	2
	27	≠(≠	2
Comment	3	≠COMMENT≠ FIND . . . METHOD	1
	28	ERATO:	1
Number	21	1000	2
Identifier	8	FOUND	2
String	27	≠(≠ 7ZD≠)≠	2
Variable	25	LATEST (Simple)	2
	16	PRIME [I] (Subscripted)	2
Array	21	NUMBS	2
Function Designator	17	NEXT	2
Expression	14	LAST - 1 (Arithmetic)	3
	26	LATEST> 0 (Boolean)	3
Assignment Statement	25	LATEST := 1	4
Procedure Statement	27	OUTPUT 61, ≠(≠ 7ZD≠)≠ ,LATEST)	6
Dummy Statement	27	Null string after semicolon	4
FOR Statement	15,16	≠FOR≠ I := LAST . . . ≠FALSE≠	4
Conditional Statement	11	≠IF≠ PRIME . . . ≠TRUE≠	4
Compound Statement	13 - 18	≠BEGIN≠ . . . ≠END≠	4
Block	1 - 28	Entire program	4
	7 - 20	Body of procedure NEXT	4
Type Declaration	22	≠INTEGER≠ LATEST, I	5
Array Declaration	21	≠BOOLEAN≠≠ARRAY≠ NUMBS [1:1000]	5
Procedure Declaration	4 - 20	Declaration of NEXT	6
Formal Parameter	4	LAST	6
Actual Parameter	26	LATEST	6
Value Part	5	≠VALUE≠ LIMIT	6
Specification	6	≠BOOLEAN≠ ≠ARRAY≠ PRIME	6
Standard Procedure (Input/Output)	27	OUTPUT	8

- FORTRAN: Each statement (except assignment statements and statement functions) begins with a keyword that identifies the statement. However, there are no reserved words; keywords can be used in other contexts within a program.

  ALGOL: One of two modes for detection of keywords is selected by the RES option on the ALGOL5 control statement. If RES is selected, keywords are reserved words, and cannot be used in any other context. They are delimited by blanks and cannot have internal blanks. If RES is not selected, keywords are delimited by the ≠ character, and can be used freely elsewhere in the program.

- FORTRAN: The data types are real, integer, logical, double precision, and complex.

  ALGOL: The data types are real, integer, and Boolean (equivalent to logical).

- FORTRAN: Arrays can have up to three dimensions, and the lower bound of each dimension must be 1.

  ALGOL: Arrays can have up to 254 dimensions. Array bounds can be any arithmetic expression (positive, negative, or zero; integer or real) as long as the upper bound is greater than or equal to the lower bound.

- FORTRAN: Variables need not be declared if they have default type. Declarations occur at the beginning of a program unit.

  ALGOL: All variables must be explicitly declared. Declarations can occur at the beginning of any block.

- FORTRAN: Maximum size of an array must be determined before execution begins.

  ALGOL: Array size can be specified dynamically during execution.

- FORTRAN: Arguments to subprograms are passed by the call-by-address method, whereby the address of an argument is passed, and the argument can be used or reset freely. In FORTRAN literature, this method is often referred to as call-by-name.

  ALGOL: Parameters to procedures are passed either by the call-by-name method (which differs from that used in FORTRAN) or the call-by-value method. These are explained in section 6.

COMMENTS

A comment in an ALGOL program can be placed either after the ≠BEGIN≠ symbol, after the ≠END≠ symbol, or after any semicolon. The format of a comment appearing after the ≠BEGIN≠ symbol or a semicolon is identical: the format is as follows:

≠COMMENT≠ sequence ;

where sequence is any sequence of characters not containing a semicolon.

After an ≠END≠ symbol, a comment consists of any characters not including a semicolon, the ≠END≠ symbol, the ≠ELSE≠ symbol, or the ≠EOP≠ symbol.

Comments are listed with the rest of the program. They have no effect on execution, unless they include comment directives and the CD parameter has been included in the ALGOL5 control statement. Comment directives are discussed in section 10. The comment directive ≠VIRTUAL≠ is always in effect, regardless of any control statement options.

Examples:

```
≠BEGIN≠ ≠COMMENT≠ MUCH ADO ABOUT NOTHING;

I := 12; ≠COMMENT≠ LET≠S HOPE IT STAYS THAT WAY;

≠END≠ OF PROGRAM

≠END≠≠COMMENT≠ END OF PROGRAM
```

SEQUENCED INPUT

When the SEQ option is selected on the ALGOL5 control statement (section 11), the input to the compiler is assumed to be in sequenced format. In this case, each line should have the following components in the order shown:

seqnum source

seqnum	Sequence number of 1 to 10 digits. The line number can have leading zeros or blanks. The line numbers must be in sorted order; if not, a diagnostic is issued. If a line number is missing or too large, the compiler generates a line number.
source	ALGOL source line.

A blank is optional between seqnum and source.

Example:

The following program segment is written to be compiled with both the SEQ and RES options:

```
090 TEST:
100 BEGIN
110 INTEGER I;
120 REAL A,B;
140 I := SQRT (A+B);
150 OUTREAL (61,I)
160 END
```

The characters in an ALGOL program are grouped and identified lexically in the following categories:

Identifiers
Numbers
Special symbols
Strings
Comments (see section 1)

Identifiers, in turn, are further categorized in one of the following syntactic categories:

Variable
Array
Label
Switch
Procedure identifier
Formal parameter

A procedure identifier is used either as a function designator or as a procedure statement. Function designators are described in this section; procedure statements are described in section 6.

LEXICAL ELEMENTS

All the characters in a syntactically correct ALGOL program are either symbols or components of symbols. A complete list of characters is given in appendix A. Some of the symbols are grouped into more inclusive entities, such as identifiers, numbers, strings, and comments. The remaining symbols are called special symbols.

SPECIAL SYMBOLS

Special symbols are predefined keywords and special characters whose appearance in an ALGOL program has a specific meaning. The special symbols are shown in appendix A.

The format of a keyword special symbol depends on the selection of the RES option on the ALGOL5 control statement. If RES is selected, keywords are reserved words, and cannot be used in any other context. They are delimited by blanks and cannot have internal blanks. If RES is not selected, keywords are delimited by the ≠ character, and can be used freely elsewhere in the program.

For example, the keyword GOTO is represented as GOTO in reserved word mode, and ≠GOTO≠ if RES is not selected. An internal blank is allowed in nonreserved word mode, but not in reserved word mode.

Allowed:

Reserved word mode	Nonreserved word mode
GOTO,≠GOTO≠,≠GO TO≠	≠GOTO≠, ≠GO TO≠

Not allowed in reserved word mode:

GO TO

In reserved word mode, the ≠ character is still required in the string quote symbols ≠(≠and≠)≠, and in the alternate form of the integer divide operator ≠/≠.

In reserved word mode, the ≠ character can still be used to delimit any keyword.

In this manual, the meaning of each special symbol is discussed in the context in which the symbol can appear.

IDENTIFIERS

Identifiers are used as names for arrays, labels, switches, simple variables, and procedures. They are also used as formal parameters of procedures, in which case they can denote any of these entities as well as strings and expressions. They must begin with a letter, which can be followed by any number of letters and digits, as diagrammed in figure 2-1.

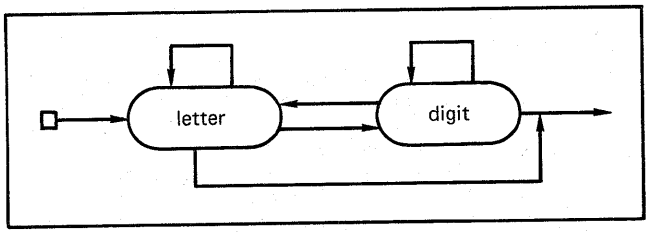


Figure 2-1. State Diagram for Identifiers

Every identifier has a scope, which is determined by the declarations and specifications in effect in the block in which it appears. The scope of an identifier is the block in which it is declared, excluding any inner blocks in which it is redeclared or respecified. Scope is explained more fully in section 4.

Examples of correctly formed identifiers:

Q
SOUP
V17A
AVERYLONGNAME
P1346790
SO UP (blanks are ignored)
FOR (not allowed as identifier in reserved word mode)

Examples of incorrectly formed identifiers:

3RD	(must begin with a letter)
P134.679	(must contain only letters and digits)

External identifiers are names required in separately compiled procedures (section 9). Figure 2-2 shows the syntax of an external identifier. An external identifier must begin with a letter, which can be followed by up to six letters and digits. If a longer identifier is used in a context requiring an external identifier, the compiler truncates the identifier to the first seven characters, and issues a diagnostic.

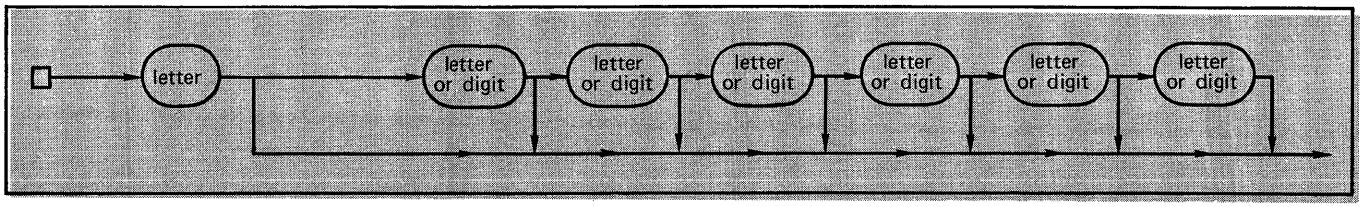


Figure 2-2. State Diagram for External Identifiers

Examples of correctly formed external identifiers:

A
ABCDEFG
X214X5

Examples of incorrectly formed external identifiers:

214X5	(must begin with a letter)
ABCDEFGH	(must be no longer than seven characters)
REAL	(legal in nonreserved word mode; illegal in reserved word mode because it conflicts with the special symbol REAL)

NUMBERS

Numbers have their conventional mathematical meanings and do not require declaration. The value of a number is determined solely by the characters it comprises. Decimal notation is used in representing numbers. A plus or minus sign, a decimal point, and an exponent are all optional parts of a number. Positive numbers can be unsigned, although a plus sign is permitted. If a number contains only digits (and possibly a sign), it is an integer; otherwise, it is a real number. Negative numbers must be prefixed with a minus sign. The syntax for integers and real numbers is given in figures 2-3 and 2-4.

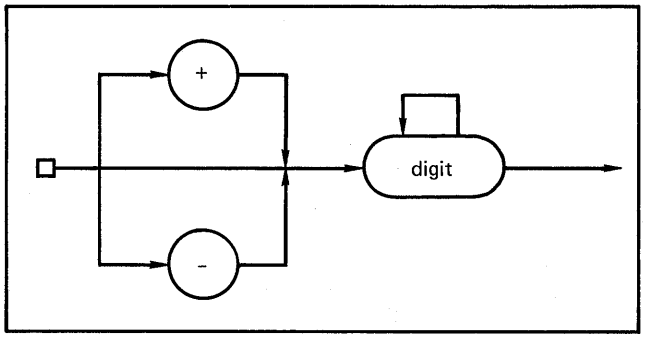


Figure 2-3. State Diagram for Integer Numbers

The exponent part of a real number consists of a not equal sign (≠), an optional plus or minus sign, and one or more digits. It signifies that the number to which it is appended is to be multiplied by an integer power of ten. For instance, the number:

.3879≠+4

represents the quantity .3879x10^4, or 3879. Fractional exponents are not allowed. An exponent can appear by itself. The value of a lone exponent is 10 raised to that power. For example, ≠+6 is 1000000; -≠+6 is -1000000.

The range of an integer constant is from $-(2^{48}-1)$ to $2^{48}-1$. An integer exceeding this range is replaced with ±MAXINT, and a diagnostic is issued. If a variable exceeds this range, the results are unpredictable. An integer constant used in a subscript must have an absolute value no greater than $2^{29}-1$. The range of absolute values of real numbers is from 10^{-293} to 10^{322}. A real number whose absolute value is less than MINREAL is replaced by zero, with no diagnostic, and a real number whose absolute value is greater than MAXREAL is replaced by ±MAXREAL, with a diagnostic. (MAXINT, MINREAL, and MAXREAL are defined in section 7.)

Examples of integers:

+123 -6 0 114090 +0 -0

Examples of real numbers:

322 114 090.125	(blanks can be inserted to improve readability, but are ignored)
3.221141≠+8	(rounded equivalent of first example)
≠-21	
5 ≠ -15	

Example of an incorrectly formed real number:

114,090 (commas are not allowed)

STRINGS

A string is either a character sequence bracketed by string quotes, or a sequence of strings and characters that is bracketed by string quotes; the latter case occurs when one string is nested inside another. Left and right string quotes are ≠(≠and≠)≠. The three characters making up the string quote must not be interrupted by blanks. The following is not a valid string:

≠(ΔΔΔ≠STRING≠)≠

(Δstands for a blank.)

A string can appear only as an actual parameter in a procedure call. Strings are limited to 131 071 characters.

Examples of strings:

≠(≠≠)≠	(Null string, with a length of zero.)
≠(≠THIS IS A STRING...≠)≠	
≠(≠AND≠(≠THIS≠(≠IS≠(≠ONE≠(≠TOO≠)≠≠)≠≠)≠≠)≠≠)≠	(Nested string)
≠(≠4ZV3D,5B,12D≠)≠	(Format string (section 8))

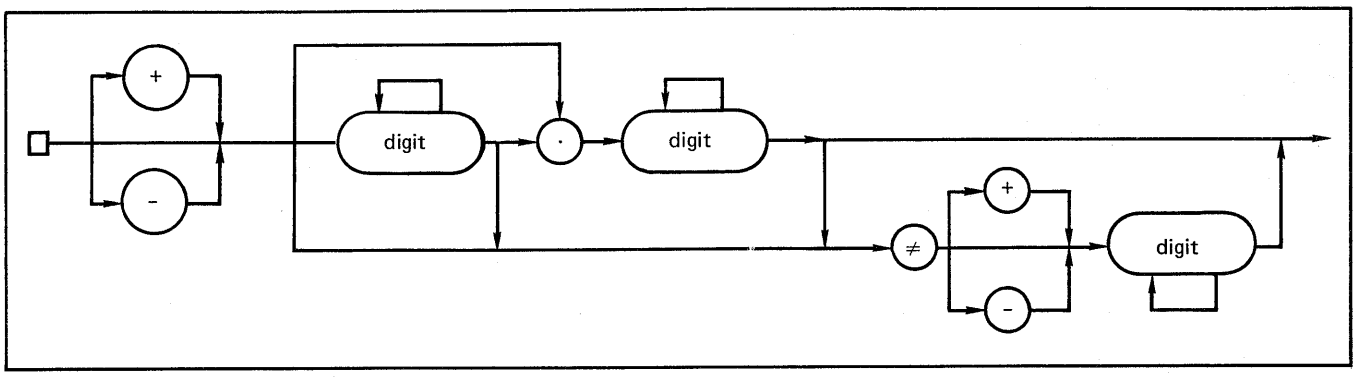


Figure 2-4. State Diagram for Real Numbers

Concatenated strings are treated as one string. That is, two strings that are separated by no characters other than blanks are the same as the string formed by deleting the right string quote of the first string, the left string quote of the second string, and any blanks that lie between the two. Concatenation only applies to outermost strings. For example, the strings:

≠(≠MOON,SUN,≠)≠≠(≠STARS,PLANETS≠)≠

and the string:

≠(≠MOON,SUN,STARS,PLANETS≠)≠

are the same. However, the string:

≠(≠ ≠(≠A≠)≠ ≠(≠B≠)≠ ≠)≠

is not the same as the string:

≠(≠ ≠(≠AB≠)≠ ≠)≠

because inner strings are not concatenated.

SYNTACTIC ELEMENTS

Special symbols, numbers, strings, and comments all represent themselves in an ALGOL program. An identifier, on the other hand, has no intrinsic meaning, but is defined by the context in which it appears. An identifier represents a variable, array, switch, label, procedure name, or formal parameter. Variables, arrays, switches and procedures must appear in declarations. Formal parameters appear in specifications. A label is implicitly declared by its usage. Some procedure identifiers do not require declaration in an ALGOL program because they are standard procedures defined in the standard circumlude (section 7).

VARIABLES

A variable is a quantity, referred to by name, whose value can be changed any number of times during program execution. Some of the actions that change the value of a variable are:

1. Executing an assignment statement in which the variable name occurs in the left part of the statement
2. Using it as an argument to a procedure that changes the argument value (call-by-name only)
3. Reading a value into it through a call to an input procedure (this is a special case of point 2).

In general, the value of a variable is retrieved or replaced wherever the name appears in the statements of a program.

A variable is either simple or subscripted. A simple variable is referred to using an identifier. It has a single value which is numeric if the identifier has been declared integer or real, and Boolean if the identifier has been declared Boolean.

A subscripted variable is an array element. (See Arrays, below.)

Every variable identifier, except for formal parameters in a procedure, must appear in a type declaration or array declaration. Formal parameter identifiers must appear in a specification in the procedure heading. The value of the variable is not defined (the variable has no value) until one is assigned to it with an assignment or procedure statement (exception: own variables, described in section 5). If the variable is used (its value is required) before it is defined, the results might not be meaningful. The DB=P option on the ALGOL 5 control statement (section 11) can be used to preset variables on entry to a block.

Examples of variables:

EPSILON (Simple variable)

DETA

A17

Q [7, 2] (Subscripted variable)

X [SIN (N* P1/2), Q [7, N]]

ARRAYS

An array is an ordered set of values. An array can have from 1 to 254 dimensions. The elements of the array are called subscripted variables and take the form:

identifier [subscript, subscript, . . .]

where identifier is the name of the array and the number of subscripts is the same as the number of dimensions in the array. Each subscript is an arithmetic (real or integer) expression. If the expression is integer, the value of the subscript is equal to the value of the expression. If the expression is real, the value of the subscript is the value of the expression, rounded to the nearest integer. Values ending in .5 are rounded up. 2.5 is rounded to 3, rather than 2. -2.5 is rounded to -2.

Each subscripted variable selects an element from the array; the location of the element is calculated from the values of the subscript expressions and the upper and lower bounds for each dimension declared for the array. The formula for computing the location of the subscripted variable is given under Array Declarations, section 5.

Array elements are ordered in memory in such a way that the rightmost subscript varies the fastest, and the leftmost subscript varies the slowest. For example, if the array declaration is A[1:3,1:2], then A[1,2] is immediately followed by [A 2,1]. (By contrast, in FORTRAN the leftmost subscript varies the fastest, and the rightmost the slowest.)

An array name without a subscript list appears only in a procedure heading or procedure call. Every array, except for formal parameters of a procedure, must be declared in an array declaration that indicates the type, the number of dimensions, and the upper and lower bounds for each dimension. An array appearing as a formal parameter of a procedure must be specified in a specification which indicates the type but not the number of dimensions or the bounds. However, the number of dimensions is determined implicitly by any array references in the procedure, and must match the number of dimensions in the actual parameter.

The value of an array is established or changed by establishing or changing the value of its elements. Since these are subscripted variables, their value is changed in the same ways that the value of any variable is changed.

A virtual array is an array which can only be accessed as a block. The principal use of this feature is to store arrays in extended memory, but a virtual array can be stored in central memory instead. A virtual array can only be used as an actual parameter. It cannot be subscripted; therefore, its individual elements cannot be accessed. By using the standard procedure MOVE (section 7), the contents of a virtual array can be transferred to a non-virtual array, after which its individual elements can be accessed. A virtual array cannot be an actual parameter for any standard procedure other than MOVE.

Virtual arrays are indicated by the comment delimiter ≠VIRTUAL≠ (section 10) appearing before the array declaration or specification (sections 5 and 6). This delimiter is always honored. The control statement option V (section 11) can be used to specify the residence of all virtual arrays in a compilation unit as central memory or extended memory. The V option only applies to a given compilation; if program units from more than one compilation are mixed (using separately compiled procedures or circumludes), it is not necessary for all the virtual arrays to have the same residence. Thus, if a separately compiled procedure using virtual arrays is compiled with the V option selected, and the procedure is subsequently called by a program that was compiled with the V option not selected, the virtual arrays in the separately compiled procedure reside in extended memory, and the virtual arrays in the program reside in central memory.

If an array is allocated to extended memory, the extended memory field length is increased as much as necessary to hold the array, up to the maximum allowed for the job.

A program using virtual arrays allocated in extended memory should not also use the standard procedures READECS and WRITEECS. No diagnostic is issued, but the results are undefined.

LABELS AND SWITCHES

A label is a quantity referred to by name whose value gives access to the position of a single statement. A label is used only as the destination of a GOTO statement. Any statement, including compound statements and blocks, can be prefixed with one or more constructs of the form:

```
label:
```

where label is any unique identifier. Several labels, such as:

```
BOSTON: NEW YORK: R2P2:
```

can precede a single statement. A label must not appear in front of a statement (or as the name of any other sort of quantity) anywhere else within its scope.

A switch is a mechanism by which a program can dynamically select a label. A switch is named and has a value, defined by a switch list, which is a list of designational expressions. A switch designator has a value which is one of the elements of the switch list, and is referred to using a name of the form:

```
switchid [subscript]
```

where switchid is the name of the switch and where the subscript identifies the position of the element in the switch list. The subscript is an arithmetic expression that reduces to an integer value in the range 1 to n, where n is the number of elements in the switch list. For example, if the value of the subscript is 1, the first element in the identified switch is evaluated and the value is assigned to the switch designator; if the value of the subscript is 2, the second element in the identified switch becomes the value of the switch designator, and so on. Switches are defined recursively in that the value of a designational expression can itself be another switch designator.

Every switch, except for formal parameters, must be defined in a switch declaration. A formal parameter switch is defined in a specification. The switch declaration specifies the labels and switch designators that are denoted by the switch identifier, and establishes their sequence. Labels are not declared in the procedure body, but when used as formal parameters must be specified in the procedure heading.

Example:

```
≠SWITCH≠ SW := L1, L2, L3;
N := 3;
≠GOTO≠ SW[N];
```

Control is transferred to label L3.

PROCEDURE IDENTIFIERS

A procedure identifier is used to call for execution of a procedure. It is also used in the procedure declaration to name the procedure. Outside the procedure, the procedure identifier appears either in a procedure statement or as a function designator. Procedure statements and function designators are identical in form but differ in how they are used; procedure statements are discussed in section 6.

A function designator is an operand in an expression whose value is the result of execution of a procedure. It differs from a procedure statement in that a value is returned through a function designator, but not through a procedure statement. The syntax of function designators is the same as procedure statements, as shown in figure 6-3.

An actual parameter can be a string, expression, array name, switch name, or procedure name. Rules applying to actual parameters are detailed under Procedure Statements, section 6.

When procedure execution is invoked within an expression, the procedure computes a single value and assigns it to the procedure name as the procedure result. This value is then used in computing the value of the expression.

Examples of function designators:

```
SIN(A-B)

J(V, N, NVAN)

R

SIMP(S+5)TEMP: (T) PRESSURE: (P)
Equivalent to SIMP(S+5,T,P)

TOLL (A) X: (P1, P2, P3) Y: (P4)
Equivalent to TOLL(A,P1,P2,P3,P4)

COMPILE (≠(≠:=≠)≠) STACK: (Q)
Equivalent to COMPILE(≠(≠:= ≠)≠,Q)
```

In addition to function designators derived from user-defined procedures, function designators provided by the system are available to invoke standard procedures. These are listed in section 7.

An expression is a series of symbols which, when evaluated at execution time, yields a single value. Expressions are of two kinds, simple and conditional.

A simple expression is a sequence of operands, together with operators that indicate the operations to be performed on the operands. The expression is evaluated by performing the indicated operations in a particular hierarchical order.

A conditional expression is a compound structure containing several simple expressions. It is of the form:

≠IF≠ Boolean expression ≠THEN≠ simple expression

≠ELSE≠ expression

If the Boolean expression yields the value true, then the value of the whole conditional expression is the value of the simple expression following ≠THEN≠; otherwise, it is the value of the expression following ≠ELSE≠. The expression following ≠ELSE≠ can also be a conditional expression, requiring a continuation of the same process.

Simple and conditional expressions can both be one of three kinds:

- Arithmetic. When evaluated, these yield a numeric value, of type real or integer.
- Boolean. When evaluated, these yield a Boolean value, true or false.
- Designational. When evaluated, these yield a label.

SIMPLE EXPRESSIONS

Simple expressions are of three types; arithmetic, Boolean, and designational. Since arithmetic expressions are components of the other two types, they are described first.

SIMPLE ARITHMETIC EXPRESSIONS

An arithmetic expression is a sequence of real and integer operands, separated by arithmetic operators, which when evaluated yields a value of type real or integer.

The operands in an expression can be any of the following:

Unsigned number

Variable (simple or subscripted)

Function designator

A parenthesized arithmetic expression:

(exp)

Any of these standing alone is also an expression. Thus, evaluation of an expression does not always require performing arithmetic operations, but can be simply the retrieval of the value of a number or variable.

An unsigned number represents its own value, and requires no operations for evaluation. A signed number, on the other hand, is considered not an operand by itself, but a unary operator (+ or -) followed by an operand. Thus the evaluation of a signed number requires the operation of addition or subtraction of the unsigned number to or from an implied zero.

A unary operator cannot be immediately preceded by another operator; the subexpression with the unary operator must be parenthesized.

Invalid:

X / -Y

Valid:

X / (-Y)

The operands in a simple arithmetic expression can be preceded by unary operators or separated from each other (in pairs) by binary operators. The unary operators are + and -, and yield, respectively, the value of the number and its negative. The binary operators are +, -, *, /, //, **, and ↑. The operations performed by these operators are described in table 3-1.

All of the operations can have either real or integer operands, except // (integer divide), which can only have integer operands. The operations *, +, and - (multiplication, addition, and subtraction) are defined as in mathematics. For all three, the result is type integer if both operands are integers; otherwise, the result is type real.

Unary + and - are treated like addition and subtraction with an implied operand to the left equal to zero.

Example:

-2 ** 2

is evaluated as 0 - (2 ** 2), not as (-2) ** 2

/ (division) is also defined as in mathematics, but the result is always type real. If one of the operands is an integer, its value is converted to real before the division is performed. A fatal error occurs if the divisor has a value of zero, and the result is subsequently used. On a CDC 7600, CYBER 70 Model 76, or a CYBER 170 Model 176, a fatal error occurs as soon as the division is performed.

// is integer division; both of its operands must be integers. It represents integer division with truncation of the remainder. I // J is equivalent to the integer whose absolute value is as great as possible without exceeding the value of I/J.

TABLE 3-1. RULES OF OPERATION PRECEDENCE

Operator	Operation	Precedence	Category
** ↑	Exponentiation Exponentiation	First	Arithmetic
* / //	Multiplication Division (real result) Division (integer operands and result)	Second	
+ -	Addition; unary plus Subtraction; unary minus	Third	
< ≤ = ¬= ≥ >	Less than Less than or equal to Equal to Not equal to Greater than or equal to Greater than	Fourth	Relational
¬	Not	Fifth	Logical
∧	And	Sixth	
∨	Or	Seventh	
→	Implies	Eighth	
≡	Is equivalent to	Ninth	

If I/J is positive, I//J is less than or equal to I/J. If I/J is negative, I//J is greater than or equal to I/J. Division by zero is not allowed. The result of integer division is of type integer.

Example:

17//4 is equal to 4.
- 17//4 is equal to -4.
- 20//4 is equal to -5.

** (exponentiation) is defined as in mathematics; that is, X**Y is X^Y. Some restrictions exist as to allowable combinations of operands, both with regard to their type (real or integer) and their value (positive, negative, or zero). Table 3-2 shows the type combinations allowed for base and exponent, depending on their value. Combinations not allowed by the table are diagnosed with a fatal error.

Association of exponentiation is to the left. That is, X**Y**Z is evaluated as (X**Y)**Z, not as X**(Y**Z). If both operands of exponentiation are integers, the result is type integer; otherwise it is type real.

If SAE1 is a simple arithmetic expression, and SAE2 is a simple arithmetic expression not preceded by unary + or -, then the following are simple arithmetic expressions:

+ SAE2

- SAE2

SAE1 + SAE2

SAE1 - SAE2

SAE1 * SAE2

SAE1 / SAE2

SAE1 // SAE2

SAE1 ** SAE2 (same as SAE1 ↑ SAE2)

This syntax is summarized in figure 3-1.

If the second operand is preceded by a unary + or -, it can appear in an expression if it is parenthesized. For example:

A+(-2.0)
-(-B)

Evaluation of an expression depends on its form, subject to the following rules:

1. An operand must be evaluated before the operation of which it is a component can be performed. Evaluation of a number follows directly from the characters it comprises. Evaluation of a simple variable involves retrieving its most recently assigned value. Evaluation of a subscripted variable requires first the evaluation of its subscript, which is in turn an expression whose evaluation is subject to all these rules; after the subscript is evaluated, the value of the indicated array element is retrieved. Evaluation of a function designator requires evaluation of its actual arguments and execution of the specified function procedure. Such execution can result in side effects (that is, operations unrelated to the computation of the result of the procedure); these side effects might cause unpredictable results, as explained below.

TABLE 3-2. TYPE COMBINATIONS ALLOWED FOR BASE-EXPONENT PAIRS

Base	Exponent		
	Value: zero	Value: negative	Value: positive
value: zero	none	none	any
value: negative	real**integer integer**integer	real**integer	real**integer integer**integer
value: positive	any	real**integer integer**real real**real	any

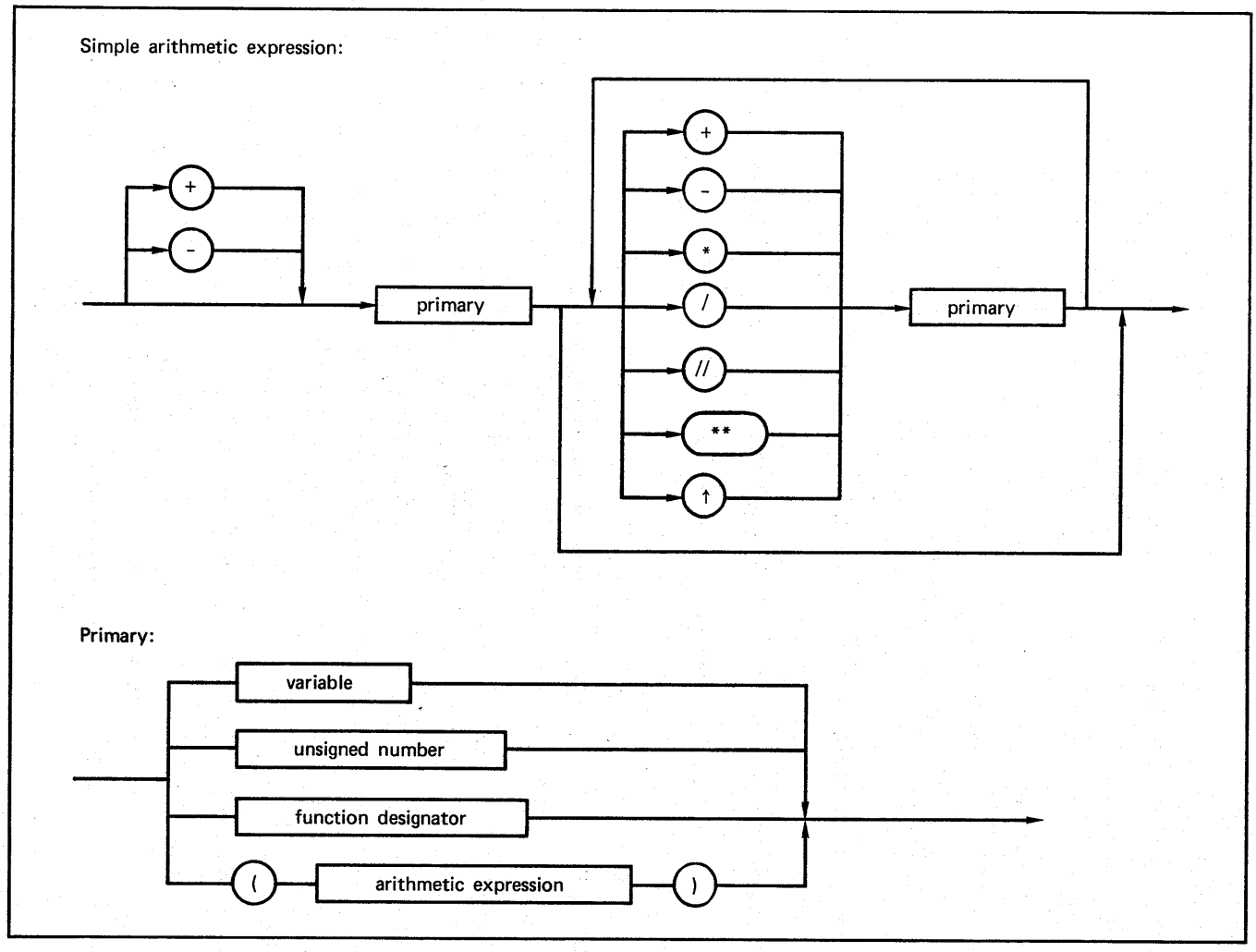


Figure 3-1. Syntax of Simple Arithmetic Expression

If a procedure is invoked by a function designator, and an exit is made from the procedure by means of a goto statement, evaluation of the expression containing the function designator is abandoned, and the value of the expression is undefined.

A parenthesized expression must be evaluated before its value can be used in a larger expression.

In the following expression:

(A + B) * (C - D)

the two operands are parenthesized expressions. Therefore, both A+B and C-D must be evaluated before the multiplication is performed.

2. When two or more consecutive operations are indicated, and the order of execution is not defined by parentheses, the order in which the operations are performed is defined by the precedence of the operators. The operation with the highest precedence is performed first. The precedence of the operators is shown in table 3-1.

   Example:

   In the expression:

   A + B * C

   two consecutive operations at the same level of parentheses are indicated. According to the precedence of operators, the multiplication is performed before the addition. Thus, the value is the same as that of the expression:

   A + (B * C)

   If the programmer wants the addition to be performed first, the expression can be rewritten:

   (A + B) * C

   In this case, rule 1 is applied and the parenthesized expression is evaluated before the multiplication is performed.

3. If two consecutive operations at the same level of parentheses have the same precedence, the operation to the left is performed first.

   Example:

   A * B / C

   The multiplication is performed before the division. The two cases in which different order of evaluation of operators with the same precedence affects the value of the result are integer division and exponentiation.

   Example:

   13 // 6 // 2 is evaluated as (13 // 6) // 2, which equals 1, not as 13 // (6 // 2), which equals 4.

   Example:

   2 ** 2 ** 3 is evaluated as (2 ** 2) ** 3, which equals 64, not as 2 ** (2 ** 3), which equals 256.

The preceding rules establish the interpretation of the expression, so as to uniquely compute its mathematical value. They do not determine the order in which operations are performed in practice. The compiler avails itself of the commutative and associative laws of arithmetic to reorder operations, as long as the reordering does not affect the mathematical result.

Example:

(A + B) * (C - D)

The value of this expression is the same whether the addition or the subtraction is performed first; therefore, the compiler might perform either operation first. In order to obtain the correct mathematical value, however, the addition and the subtraction must be performed before the multiplication.

If T1 and T2 are intermediate results, this expression might be computed in either of the following ways:

1. T1 := A + B
   T2 := C - D
   RESULT := T1 * T2

2. T1 := C-D
   T2 := A + B
   RESULT := T1 * T2

When the mathematical result of evaluation of an expression does not depend on the order in which the operations are performed, the compiler might evaluate the expression in any order that produces the correct results. The compiler freely uses the associative and commutative laws to reorder operations. If the results of the program depend on a particular ordering of operations, the program is undefined; the results might not be the same each time the program is compiled. This situation particularly arises when one of the operands of an expression is a function, and execution of the function produces results other than those necessary to compute the value of the function designator. Such results are known as side effects.

In the example shown in figure 3-2, because addition is commutative, either A(3) or B(4) could be evaluated first; therefore, the order in which the two strings are output cannot be determined. (Output procedures are explained in section 8.)

The sequence of statements shown in figure 3-3 might not produce correct results, because the procedure F changes the value of its argument, which is X in this case. If F(X) is evaluated before X (which is permitted according to the commutative law), the expression might have a different value.

Different orders of evaluation can also produce different results because of accumulated rounding errors.

Examples of simple expressions are shown in table 3-3.

```
C := A (3) + B (4)
     .
     .
     .
≠REAL≠ ≠PROCEDURE≠A(INT);
     .
     .
     .
OUTSTRING (61,≠(≠FIRST LINE≠)≠);
     .
     .
     .
≠END≠;
≠REAL≠ ≠PROCEDURE≠B(INT2);
     .
     .
     .
OUTSTRING (61,≠(≠SECOND LINE≠)≠);
     .
     .
     .
≠END≠
```

Figure 3-2. Order of Evaluation Example 1

```
Y := X + F(X)
     .
     .
     .
≠REAL≠ ≠PROCEDURE≠F(A)
     .
     .
     .
A:=12;
     .
     .
     .
≠END≠
```

Figure 3-3. Order of Evaluation Example 2

SIMPLE BOOLEAN EXPRESSIONS

A Boolean expression yields a Boolean value (true or false). Simple Boolean expressions use both relational and Boolean operators. The operand values used in a relational operation can be real or integer; mixing operand types is acceptable. The operand values used in a Boolean operation must be Boolean. Relations are constructed from arithmetic expressions, and Boolean expressions are constructed from relations and Boolean variables, constants, and function designators.

Relations

A relation is a comparison between the values of two arithmetic expressions. The result of the comparison is either true or false.

The relational operators are shown in table 3-1. Relational operators have a higher precedence than any of the logical operators, so that in a Boolean expression, relations are evaluated first.

If SAE1 and SAE2 are simple arithmetic expressions, then the following are relations:

SAE1 = SAE2

SAE1 ¬ = SAE2

SAE1 < SAE2

SAE1 ≤ SAE2

SAE1 > SAE2

SAE1 ≥ SAE2

To evaluate the relation, the expressions SAE1 and SAE2 are first evaluated. Then the specified comparison is performed. If SAE1 and SAE2 are in the specified relationship to each other (SAE1 is equal to SAE2, not equal, less than, and so forth), then the result of the relational operation is true; otherwise it is false.

Logical Operations

A logical operation is a truth-valued operation performed on operands of type Boolean. The logical operators are shown in figure 3-1. Logical operators have the lowest precedence of all ALGOL operators, which means that they are the last to be evaluated in any particular expression. The truth tables illustrating the effect of these operations on all combinations of logical values are listed in table 3-4.

A logical operand can be any of the following:

- Relation
- Boolean value
- Simple or subscripted variable of type Boolean

TABLE 3-3. SIMPLE ARITHMETIC EXPRESSION EVALUATIONS

Expression	Value
10	The integer number 10.
2**(-16.0)	Integer 2 raised to the -16. The result is type real.
N	Current numeric value of the simple variable N (type must have been declared to be real or integer).
N + 10 * CDL	Value of CDL multiplied by integer 10, added to the value of N.
B(N,R+1,S)	Value yielded by the procedure B when applied to the current values of the parameters N, R+1, and S (B must have been declared to be a function procedure and must be assigned a value).
A[N+10*CDL]	Current value of the array element (in array A) identified by the value of the subscript expression that is inside the square brackets (A must have been declared to be an integer or real array).
1/(P/(2**(-16.0)-A[N+10*CDL]))	Integer 1 divided by the value of (P/(2**(-16.0)-A[N+10*CDL])), which is the value of P divided by the value of (2**(-16.0)-A [N+10*CDL]) which is the difference of the value of 2**(-16.0) and the value of A[N+10*CDL].

TABLE 3-4. LOGICAL VALUES YIELDED BY LOGICAL OPERATIONS

P	Q	¬Q	P∧Q	P∨Q	P→Q	P≡Q
TRUE	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE
FALSE	TRUE	FALSE	FALSE	TRUE	TRUE	FALSE
FALSE	FALSE	TRUE	FALSE	FALSE	TRUE	TRUE

- Function designator of type Boolean
- Parenthesized Boolean expression:

  (bexp)

Any of these standing alone is a simple Boolean expression.

If SBE1 and SBE2 are simple Boolean expressions, then the following:

SBE1 ∧ SBE2

SBE1 ∨ SBE2

SBE1 → SBE2

SBE1 ≡ SBE2

are also simple Boolean expressions. If SBE3 is a simple Boolean expression not preceded by the ¬ operator, then ¬SBE3 is a simple Boolean expression. The syntax of simple Boolean expressions is shown in figure 3-4.

Example:

A*2.997<B∧C [2,N]

For this expression to be valid, the variables A, B, and N must be real or integer, and C must be a Boolean array. The relation is evaluated first to yield a value true or false, depending on whether the value of B is greater than A*2.997 or not, respectively. For the expression value to be true, both the relation and the Boolean array element C[2,N] must have values of true; if either or both have the value false, the expression value is false too.

The ¬ operator can appear adjacent to itself only with intervening parentheses, as in the following constructs:

¬(¬P)

¬(¬(¬P))

The ¬ operator can appear adjacent to any other logical operator, but only as the operator on the right, as in the following constructs:

P∨¬Z

¬P≡¬Z

The operators ∧, ∨, ≡, and → cannot appear adjacent to each other. This corresponds to the conventional usage of logical disjunction and conjunction.

Some simple Boolean expression evaluations are shown in table 3-5.

SIMPLE DESIGNATIONAL EXPRESSIONS

A simple designational expression is simply a label or a switch designator. No operator symbols are used.

A switch designator has the form:

switchid [subscript]

where switchid is the name of a switch and subscript is an arithmetic expression. Switch designators look like subscripted variables, except that, whereas a switch designator can have only one subscript, a subscripted variable can have more than one subscript (when the subscripted variable refers to a multidimensional array).

The subscript in a switch designator is an arithmetic expression that reduces to an integer value in the range 1 to n, where n is the number of items in the switch declaration list. The current value of the subscript selects one of the designational expressions listed in the switch declaration by counting these from left to right until the count is the same as the subscript value.

Figure 3-5 shows the syntax for simple designational expressions. Using designational expressions is discussed in section 4, under the GOTO statement.

CONDITIONAL EXPRESSIONS

A conditional expression is an expression whose value is one of two or more alternatives, depending on the value of a Boolean expression.

The general form for conditional expressions is:

≠IF≠ bexpr ≠THEN≠ simple expression ≠ELSE≠ condexpr

where condexpr is either a simple or conditional expression, and bexpr is a Boolean expression. The syntax for conditional expressions is shown in figure 3-6.

The expression following ≠ELSE≠ must be the same kind as the simple expression; that is, both must be arithmetic, Boolean, or designational. The value yielded by the entire expression is then of the same kind.

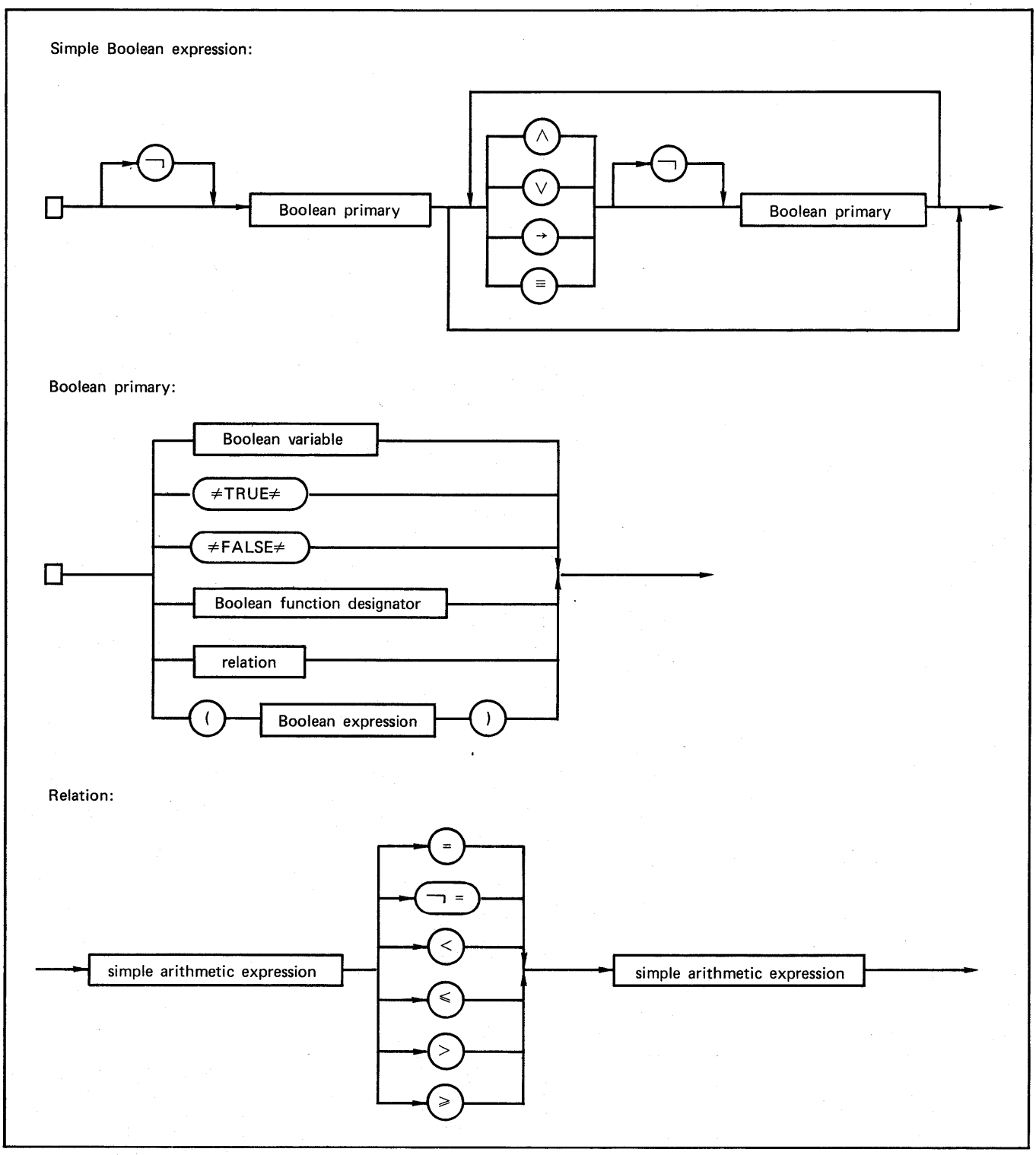


Figure 3-4. Syntax of Simple Boolean Expressions

EVALUATING A CONDITIONAL EXPRESSION

Evaluating a conditional expression consists of selecting one of two or more simple expressions and evaluating the selected expression. Selection is based on the value of the Boolean expression following the ≠IF≠ symbol. If the Boolean expression is true, the value of the whole expression is the value of the simple expression following ≠THEN≠. Otherwise, it is the value of the expression following ≠ELSE≠. Since both the Boolean expression and the expression following ≠ELSE≠ can be conditional expressions, evaluation might be in several stages.

Some conditional expressions and their values are shown in table 3-6.

TABLE 3-5. SIMPLE BOOLEAN EXPRESSION EVALUATIONS

Expression	Value
≠TRUE≠	The logical value true.
T	Current logical value of the simple variable T (the type of T must have been declared to be Boolean).
8*R¬=S	true if the current numeric value of the arithmetic expression S is greater than or less than the current numeric value of the arithmetic expression 8*R, false otherwise.
¬Q	true if the current logical value of the simple variable Q is false, and false otherwise.
8*R¬=S ≡ (¬ Q)	true if the current logical value of the relation 8*R¬=S is the same as the current logical value of the logical expression (¬Q), and false otherwise.
A∧B∧C∧D∧E→R→T	true if the current logical value of the simple variable T is true, or if the current logical values of T and of the logical expression: A∧B∧C∧D∧E→R are both false; and false otherwise. The current value of the expression: A∧B∧C∧D∧E→R is false if the current value of R is false and the current value of the logical expression: A∧B∧C∧D∧E is true. The value of the expression: A∧B∧C∧D∧E is true if the current values of the simple variables A,B,C,D, and E are all true.
8*R¬=S≡ (≠IF≠ FLAG ≠THEN≠ Q ≠ELSE≠¬Q)	true if the current logical value of the relation 8*R¬=S is the same as the current logical value of the conditional expression: ≠IF≠FLAG ≠THEN≠ Q ≠ELSE≠ ¬Q and false otherwise.

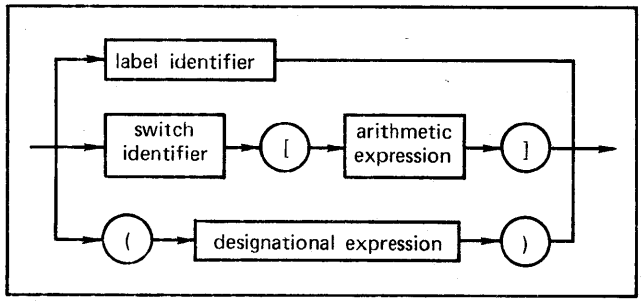


Figure 3-5. State Diagram for Simple Designational Expressions

EXPRESSION TYPE

The type of a conditional expression depends on the types of the simple expression and condexpr. If they are both Boolean, then the type of the conditional expression is Boolean. If the simple expression is type real, then the type of the conditional expression is real. If the type of the simple expression is integer, then the type of the conditional expression is that of condexpr: if condexpr is real, the conditional expression is real; if condexpr is integer, the conditional expression is integer.

Example:

Given the following declarations:

B is type Boolean.

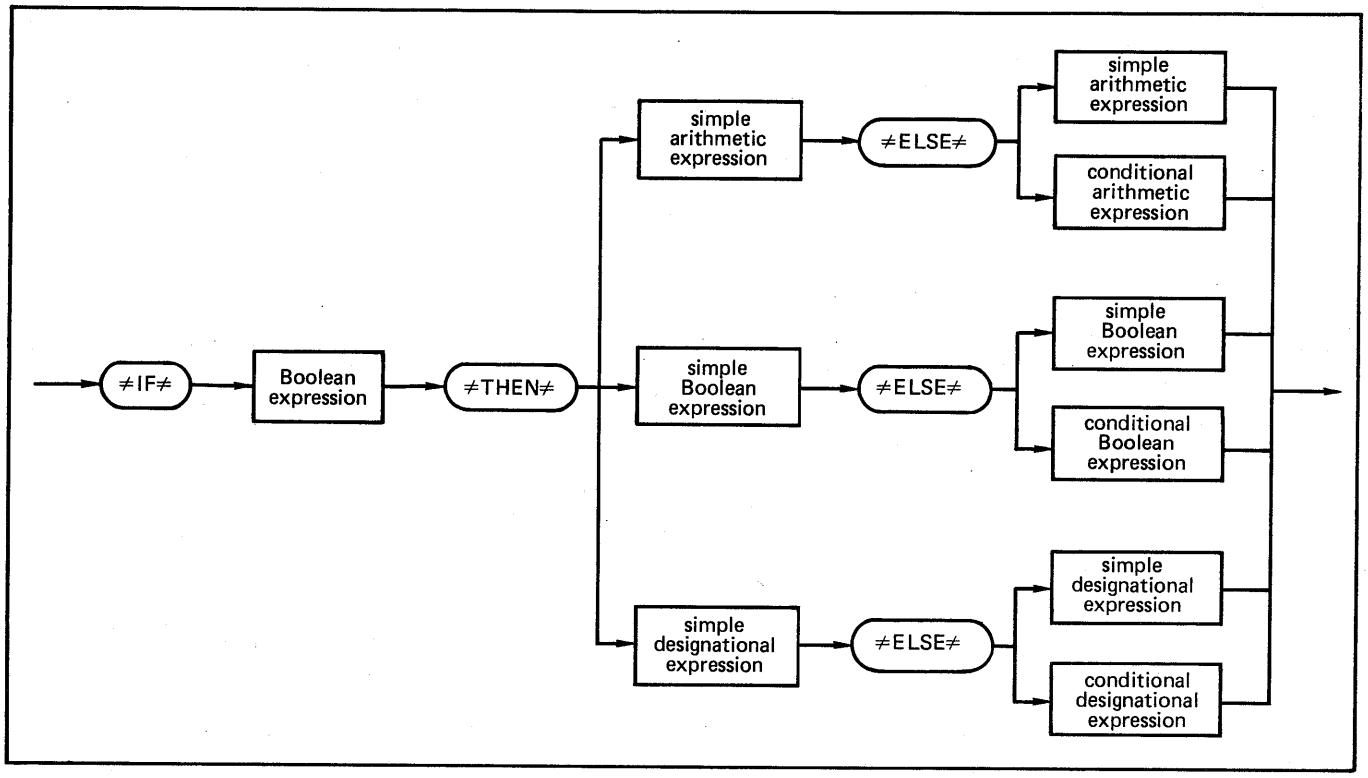


Figure 3-6. State Diagram for Conditional Expressions

TABLE 3-6. CONDITIONAL EXPRESSION EVALUATIONS

Expression	Value
≠IF≠ Q ≠THEN≠ N-1 ≠ELSE≠ N	Value of the arithmetic expression N-1 if value of the Boolean variable Q is ≠TRUE≠; otherwise, the current value of N.
≠IF≠ Q<0 ≠THEN≠ S+3*Q/A ≠ELSE≠ F9(Q)	Value of the arithmetic expression S+3*Q/A if the value of the Boolean expression Q<0 is true; otherwise, the procedure F9 is called with the value of Q as the argument.
≠IF≠ K<1 ≠THEN≠ S>W ≠ELSE≠ A≤C	Value of the Boolean expression S>W if the value of K is less than 1; otherwise, the value of the Boolean expression A≤C.
≠IF≠ A<0 ≠THEN≠ U+V ≠ELSE≠ ≠IF≠ A*B>17 ≠THEN≠ U/V ≠ELSE≠ ≠IF≠ K¬=Y ≠THEN≠ V/U ≠ELSE≠ 0	Value of one of the arithmetic expressions: U+V U/V V/U 0 depending on whether the first Boolean expression (scanning from left to right) that is found to have the value true is: A<0 A*B>17 K¬=Y none of the above respectively.
≠IF≠ AB≤C ∨ AB≥D ≠THEN≠ L17 ≠ELSE≠ L27	The label L17 if the Boolean expression: AB≤C ∨ AB≥D has a current value of true; otherwise, the label L27.
≠IF≠ T ≠THEN≠ CAIRO ≠ELSE≠ TOWN[≠IF≠Y<0≠THEN≠N ≠ELSE≠N+1]	The label CAIRO if the current value of T is true; the label designated by the switch designator: TOWN[≠IF≠Y<0≠THEN≠N≠ELSE≠N+1] if the current value of T is false.

S is type integer.
R is type real.

the type of the conditional expression:

≠IF≠ B ≠THEN≠ S ≠ELSE≠ R

is real, even if the current value of B is true, so that S (which is type integer) is selected. The integer value of S, converted to real, is the value of the conditional expression in this case.

Statements describe the actions of an ALGOL program. Since sequences of statements can be grouped together into compound statements and blocks, which are also statements, the definition of statement is recursive. Also, since declarations enter fundamentally into the structure of most statements, the definition of statements in this section presumes that appropriate declarations have already been made. (See section 5 for declarations.)

Normally, the sequence in which statements are executed is defined implicitly: the statements in a program are executed consecutively in the order in which they appear in the program. However, the normal execution sequence can be altered by any of the following statements:

- GOTO — Defines its successor statement explicitly.
- Procedure — Causes the specified procedure to be executed.
- Conditional — Can cause specified statements to be skipped.
- FOR — Can cause specified statements to be repeated.

Some ALGOL statements contain other kinds of statements. The simplest statements, which do not contain other statements, are:

- Assignment statements
- GOTO statement
- Procedure statements

These statements are the first to be described in this section. They perform some of the basic actions of any program: giving quantities values, and unconditionally altering the normal execution sequence. Later in the section, the statements that contain other statements are described:

- Compound statements and blocks
- FOR statements
- Conditional statements

These perform the more complex functions of grouping statements, controlling repeated execution of groups of statements, and conditionally altering the normal execution sequence.

Dummy statements, which are written as zero or more blank characters, perform no action and are generally used to carry labels. A dummy statement, like any other ALGOL statement, must be separated from a preceding statement (if there is one) by a semicolon.

Since the last statement before the ≠END≠ symbol of a block cannot be followed by a semicolon, a semicolon immediately before an ≠END≠ symbol indicates a dummy statement.

Any statement can optionally be labeled with one or more labels in the following format:

```
label: label: ... label: statement
```

This also applies to statements that are components of other statements.

Example:

In the program segment:

```
A:=B;
L:  ;
.
.
.
≠GOTO≠ L;
≠END≠
```

the symbols:

```
L:
```

constitute a labeled dummy statement to which a subsequent GOTO statement specifies that control is to be transferred. The semicolon after ≠GOTO≠ L is followed by another dummy statement.

ASSIGNMENT STATEMENT

An assignment statement gives a value to one or more variables and procedure names. The syntax is shown in figure 4-1. The statement has either of the following forms:

```
dest := expr
dest := assignment statement
```

where dest, the destination of the assignment, is a variable (either simple or subscripted) or a procedure name; and expr is an arithmetic expression if the type of dest is real or integer, and a Boolean expression if dest is of type Boolean. A destination can be a procedure name only within the body of a function procedure having that procedure name.

The second form allows multiple assignments such as:

```
A:=B:=C:=D:=0
```

where all of the destinations in the statement must be the same type.

The assignment operator := is not to be confused with the relational operator = (the equal to operator).

Examples:

```
A := B/C-V-Q*S
SF[V,I+2]:= 3-ARCTAN(S*ZETA)
V:=Q>U^Z
N:=N+1
S:=P[0] :=N:=N+1+S
A:=≠IF≠FLG1=FLG2≠THEN≠ B/C-V-Q*S ≠ELSE≠ 0
```

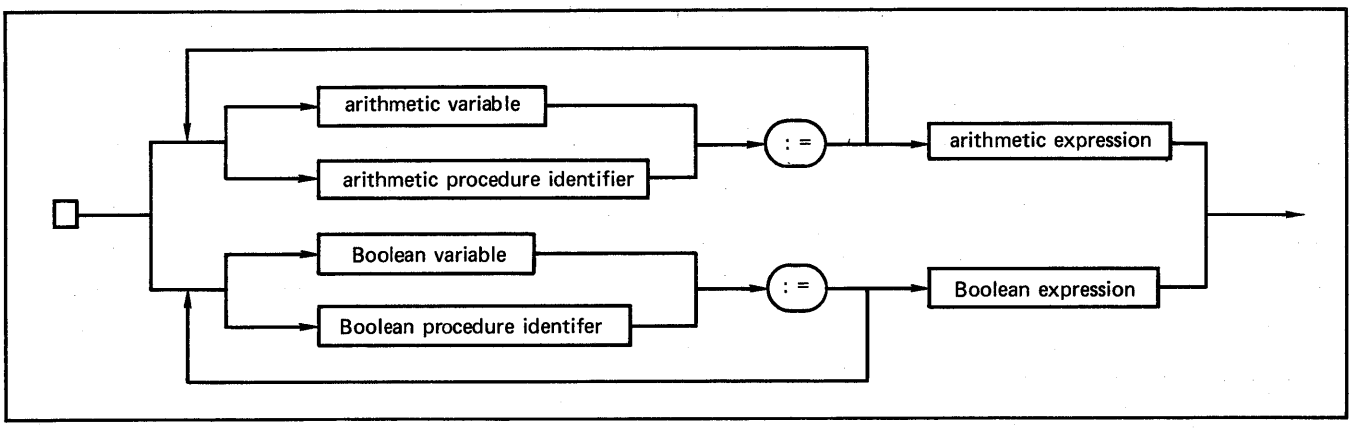


Figure 4-1. Syntax of Assignment Statement

The process of assigning the new value proceeds as follows:

1. Any subscripts occurring among the destinations of an assignment statement are evaluated in sequence from left to right in the statement.

2. The arithmetic or Boolean expression (the right part of the statement) is evaluated.

3. The value of the expression is assigned to all of the destinations, with any subscripts that appear in the left part list having the values that were obtained in step 1. If the type of the right part expression differs from that of the destination (the left part), and both are arithmetic, the type of the expression value is converted to that of the destination. Rounding is performed on conversion from a real to an integer value. The result of rounding is the largest integral quantity not exceeding n+0.5, where n is the value of the expression. That is, the value is rounded to the nearest integer. For example, 2.8 is rounded to 3, and 3.5 is rounded to 4; -3.5, however, is rounded to -3.

Example:

The statement:

```
N:=A[N]:=(N+1)*100
```

is evaluated as follows. Suppose that the current value of N is 0. Then the left part list refers to array element A[0]. The expression is evaluated to 100, which is then assigned as the new value of the array element A[0] and as the new value of N.

Example:

The statement:

```
M:=N:=A:=B[0]:=0
```

assigns the value 0 to each of M, N, A, and B[0]. In this case, the set of assignment statements:

```
M:=0; N:=0; A:=0; B[0]:=0;
```

would have a result identical to that of the multiple assignment.

A multiple assignment statement, however, does not always specify the same actions as would a separate assignment statement for each of the destinations. The expression of the right side of the assignment statement is evaluated only once in a multiple assignment, and more than once if separate assignments are made.

Example:

Assuming that in every case, N initially has a value of 0, the assignment statement:

```
N:=A[N]:=(N+1)*100
```

results in a value of 100 for N and for A[0]. However, the corresponding pair of assignment statements:

```
N:=(N+1)*100;
A[N]:=(N+1)*100;
```

would assign a value of 100 to N as before, but this time would assign a value of 10,100 to A[100].

GOTO STATEMENT

A GOTO statement interrupts the implicit execution sequence, causing control to transfer unconditionally to a labeled statement. The syntax is shown in figure 4-2. In reserved word mode, the characters GOTO must be entered with no intervening blanks. The GOTO statement has the form:

≠GOTO≠ designational expression

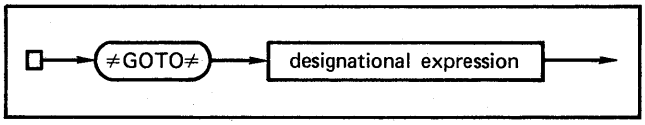


Figure 4-2. Syntax of GOTO Statements

The labeled statement whose label has the value of the designational expression is the next statement to be executed. Control does not return to the statement immediately following the GOTO statement unless another GOTO statement is executed that transfers control to that point. Therefore, if the statement following the GOTO statement is to be executed, it must be labeled.

Examples:

```
≠GOTO≠ EXIT
≠GOTO≠ SWITCH [M+1]
≠GOTO≠ L
```

Labels are local to a block (as explained under Compound Statements and Blocks in this section), so no GOTO statement can transfer control from outside a block into the block. A GOTO statement can, however, lead from outside into a compound statement (unless the compound statement is a procedure body or the statement part of a FOR statement).

For example, in the program segment shown in figure 4-3, when the statement ≠GOTO≠ L is executed, control passes unconditionally to the statement N:=N+1 (which is part of a compound statement) without the Boolean expression X<2 being evaluated. Execution proceeds (the next statement to be executed after the compound statement is R:=A[N]) until it reaches the statement ≠GOTO≠ M. At that point, control passes unconditionally back up to the statement R:=0 so that next time execution reaches the IF statement, the entire IF statement is evaluated.

```
        ≠GOTO≠ L;
M:      R:=0;
          .
          .
          .
        ≠IF≠X<2 ≠THEN≠
        ≠BEGIN≠
L:      N:=N+1;
          .
          .
          .
        ≠END≠;
        R:=A[N];
          .
          .
          .
        ≠GOTO≠ M;
```

Figure 4-3. Branch Into a Compound Statement

PROCEDURE STATEMENT

A procedure statement is written as a procedure identifier optionally suffixed by an actual parameter list. The statement is used to invoke execution of a procedure body; control is passed to the procedure named in the procedure statement. If the called procedure exits through the final ≠END≠, control returns to the statement immediately following the procedure statement. If the called procedure exits through a GOTO statement, control is transferred to the label referenced in the GOTO statement.

Procedure statements are described in detail in section 6.

COMPOUND STATEMENTS AND BLOCKS

ALGOL enables the user to group any number of statements and declarations, precede the group with a ≠BEGIN≠ symbol, follow it with an ≠END≠ symbol, and then use this group as a single statement wherever an ALGOL statement is allowed. The group is called a block if it begins with declarations; otherwise, it is called a compound statement. The syntax of a compound statement is shown in figure 4-4; the syntax of a block is shown in figure 4-5.

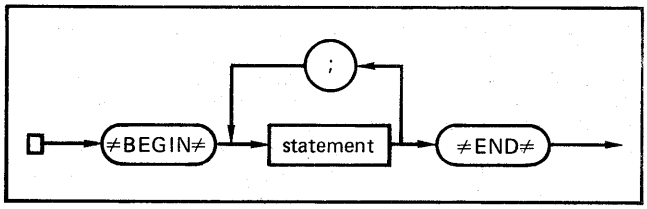


Figure 4-4. Syntax of Compound Statements

A compound statement has the form:

≠BEGIN≠ s; s; . . . s; s ≠END≠

where s is any kind of ALGOL statement, including possibly another compound statement or a block. A block has the form:

≠BEGIN≠ d; d; . . . d; d; s; s; . . . s; s ≠END≠

where d is a declaration, and s is any kind of ALGOL statement, including possibly another block or a compound statement. A block or compound statement must contain at least one statement, although that statement can be a dummy statement. A block must contain at least one declaration. Semicolons must be used to separate the statements in a block or compound statement containing two or more statements. Semicolons can precede the ≠END≠ symbol, but are not required. A semicolon must follow an ≠END≠ symbol unless it terminates a program, or is followed by an ≠END≠ or ≠ELSE≠.

Examples of compound statements:

≠BEGIN≠ A:=A+2; ≠GOTO≠ R ≠END≠
≠BEGIN≠ A:=B ≠END≠

The example in figure 4-6 shows a block containing a compound statement. The compound statement begins on line 4 and ends on line 7.

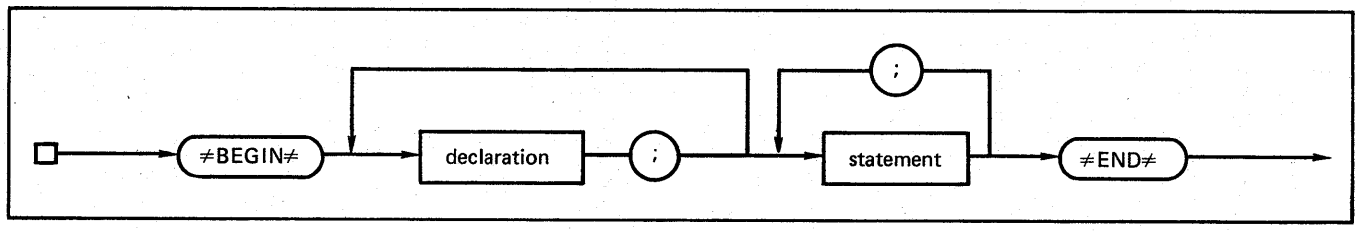


Figure 4-5. Syntax of Blocks

```
1.  ≠BEGIN≠ ≠REAL≠ W; ≠INTEGER≠ I,K;
2.    ≠FOR≠ I:=1 ≠STEP≠ 1 ≠UNTIL≠ N ≠DO≠
3.      ≠FOR≠ K:=1+I ≠STEP≠ 1 ≠UNTIL≠ N ≠DO≠
4.        ≠BEGIN≠ W:=A [I,K];
5.                    A [I,K]:=A [K,I];
6.                    A [K,I]:=W
7.        ≠END≠
8.  ≠END≠
```

Figure 4-6. Block Example

Blocks have a special status among ALGOL statements in that blocks define the scope of identifiers by means of the declarations at the head of every block. That is, the quantity to which an identifier refers depends on the static block-structure of the procedure in which it is used. The block in which an identifier declaration appears, along with any other blocks contained within that block, but not including any inner blocks in which the identifier is redeclared, is the scope of that identifier. Outside of that block, the identifier has other meanings (if declared in more inclusive blocks) or else no meaning at all.

The identifiers that are used inside a block and which have been declared at the beginning of that block are said to be local to the block. This means two things:

- The quantity referred to by a local identifier has no existence outside of the block.
- Any quantity represented by this identifier outside of the block is inaccessible inside the block.

Identifiers, other than those representing labels, that are used inside the block, but which have not been declared at the beginning of the block, are said to be nonlocal to the block. This means that the quantity referred to by a nonlocal identifier represents the same quantity inside the block and in the level immediately outside it. (The concepts of local and nonlocal to a block apply only to blocks, and not to compound statements.)

Since a statement in a block can itself be a block, the concepts of local and nonlocal to a block are recursive. An identifier that is nonlocal to block A may or may not be nonlocal to the block B in which A is one statement.

In example A of figure 4-7, the identifier A referred to in the assignment statement is the variable local to the inner block, BLOCK2. This variable does not exist in the outer block, BLOCK1.

In example B of figure 4-7, the variable A referred to in BLOCK2 is nonlocal to BLOCK2 but local to BLOCK1. Therefore, the assignment statement assigns a value to the variable declared in BLOCK1.

In example C of figure 4-7, a variable named A is declared in BLOCK1. Then, when BLOCK2 is entered, another variable with the same name is declared. The previous declaration ceases to have any effect, and any references to A, such as the one in the assignment statement A := 14, are to the variable declared in BLOCK2, not to the variable declared in BLOCK1. When BLOCK2 is exited through the ≠END≠ symbol, the previous declaration of A becomes effective again, and the variable referenced in the assignment statement A := 25 is the variable declared in BLOCK1.

```
(A) BLOCK1:  ≠BEGIN≠ ≠REAL≠ X;
               .
               .
               .
             BLOCK2:  ≠BEGIN≠ ≠INTEGER≠ A;
                                A:=12;
                                  .
                                  .
                                  .
                      ≠END≠
               .
               .
               .
             ≠END≠

(B) BLOCK1:  ≠BEGIN≠ ≠INTEGER≠ A;
               .
               .
               .
             BLOCK2:  ≠BEGIN≠ ≠REAL≠ X;
                      A:=13;
                        .
                        .
                        .
                      ≠END≠
               .
               .
               .
             ≠END≠

(C) BLOCK1:  ≠BEGIN≠ ≠INTEGER≠ A;
               .
               .
               .
             BLOCK2:  ≠BEGIN≠ ≠INTEGER≠ A;
                      A:=14;
                        .
                        .
                        .
                      ≠END≠
    A:=25
    ≠END≠
```

Figure 4-7. Nested Block Example

Nonlocal identifiers representing labels behave as though they were local identifiers. A label identifier is implicitly declared at the beginning of the smallest block in which the label identifier appears as a statement label (that is, appears suffixed with a colon in front of a statement). Figure 4-8 diagrams three control structures involving blocks (represented by the rectangles) and GOTO statements.

In block diagram A, the GOTO statement in the outer block transfers control to the statement labeled L in the outer block, while the GOTO statement in the inner block transfers control to the statement labeled L in the inner block.

In block diagram B, the GOTO statements in the outer block and the inner block both transfer control to the statement labeled L in the outer block.

In block diagram C, the GOTO statement in the inner block transfers control to the statement labeled L in the inner block, but the GOTO statement in the outer block

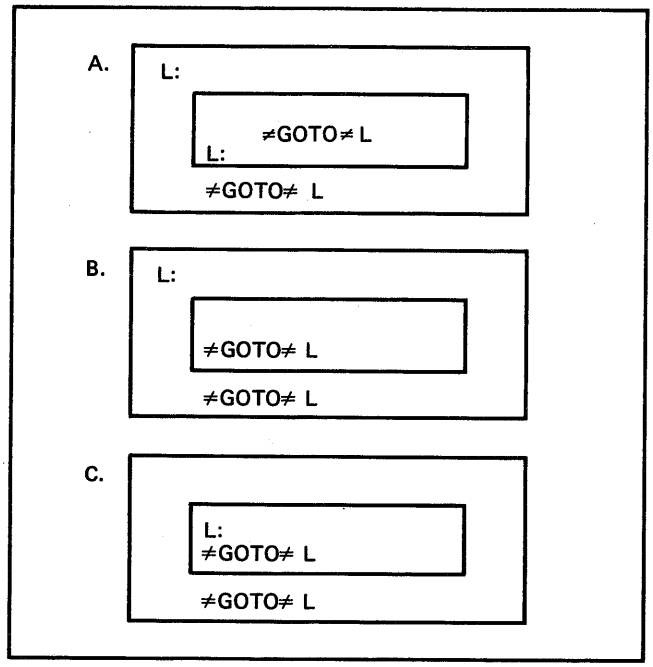


Figure 4-8. Three Block Structures

produces an error. The label in the inner block has a scope limited to the inner block, and has no meaning outside the inner block.

Certain syntactic structures in ALGOL are said to act like blocks. This means that the scopes of the labels involved in these structures are limited. In particular, the statement part of a FOR statement and the procedure body of a procedure (when the body is not a block) act as if they were bracketed by ≠BEGIN≠ and ≠END≠ as far as the scope of labels is concerned. Further explanations are given in the discussions of the FOR statement and procedure declarations.

A program is a block or a compound statement.

FOR STATEMENT

A FOR statement causes a specified statement to be executed repeatedly. The syntax is shown in figure 4-9. Any FOR statement consists of two parts:

- A FOR clause
- A statement

The FOR clause part immediately precedes the statement part. It contains specifications for controlling the iterative execution of the statement part. The statement part of the FOR statement can be any kind of statement, including another FOR statement. The statement part is always treated as though it were a block even if it does not have the form of one (this affects the scope of labels that appear in the statement part).

The form of the FOR clause is:

≠FOR≠ var:=forlist ≠DO≠

where var (called the control variable) is a real or integer simple variable, and forlist is a list of one or more elements each of which has one of the forms:

aexpr
aexpr ≠STEP≠ aexpr2 ≠UNTIL≠ aexpr3
aexpr ≠WHILE≠ bexpr

where aexpr, aexpr2, and aexpr3 are arithmetic expressions and bexpr is a Boolean expression. The list elements of forlist must be separated by commas.

Examples:

1. ≠FOR≠ I:=IVAL ≠DO≠ A[I]:=B[I]
2. ≠FOR≠ I:=IVAL, IVAL+1, IVAL+2, IVAL+3 ≠DO≠ A[I,1] :=B [1,I]
3. ≠FOR≠K := K*2 ≠WHILE≠K N ≠DO≠
   ≠FOR≠ J:= 1 ≠STEP≠ 1 ≠UNTIL≠ N ≠DO≠
   A[K,J] :=B[K,J]

MULTIELEMENT FORLIST

A FOR statement containing n list elements in the forlist part of the ≠FOR≠ clause has the same meaning as n FOR statements, each having a forlist consisting of only one of the n list elements. For the correspondence to be exact, the order of the n statements would be dictated by the order of the n list elements, and the n statements would have to be a block.

Example:

The FOR statement:

≠FOR≠ I := A,B,C ≠DO≠ statement

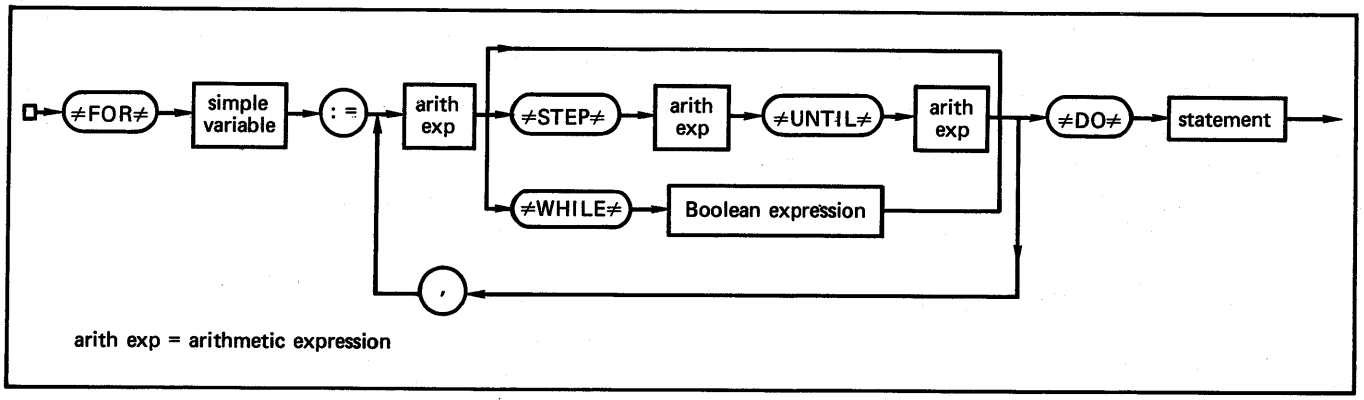


Figure 4-9. Syntax of FOR Statements

and the compound statement:

```
≠BEGIN≠
    ≠FOR≠ I:=A ≠DO≠ statement;
    ≠FOR≠ I:=B ≠DO≠ statement;
    ≠FOR≠ I:=C ≠DO≠ statement;
≠END≠
```

perform the same actions, assuming the statement is the same in each case.

EXPRESSION FORLIST ELEMENTS

A FOR clause of the form:

```
≠FOR≠ ident:=aexpr ≠DO≠
```

performs a single assignment of the arithmetic expression aexpr to the variable ident. This form initializes the variable ident before the statement part is executed. The control variable ident may or may not occur in the statement part.

The compound statement:

```
≠BEGIN≠
ident:=aexpr; statement
≠END≠
```

and the FOR statement:

```
≠FOR≠ ident:=aexpr ≠DO≠ statement
```

are effectively the same. With this type of FOR statement, the statement part is executed once for each execution of the FOR statement.

Example:

The FOR statement:

```
≠FOR≠ I:= IVAL ≠DO≠ A[I]:=B[I]
```

first assigns the value of IVAL to I, then assigns the value of array element B[I] to the array element A[I]. Suppose that the value of IVAL is 10. Then, the value of I becomes 10 and the value of A[10] becomes that of B[10].

STEP/UNTIL FORLIST ELEMENTS

A FOR statement of the form:

```
≠FOR≠ ident := aexpr1 ≠STEP≠ aexpr2
    ≠UNTIL≠ aexpr3 ≠DO≠ statement
```

causes the statement part of the FOR statement to be performed zero, one, or more times with ident typically having a different value each time. The simple variable ident may or may not occur in the statement part or in aexpr. Execution of a FOR statement containing this kind of FOR clause causes the following sequence of operations:

1. The variable ident is assigned the value of aexpr1.

2. The value of aexpr2 is computed and assigned to a temporary variable; for example, tvar.

3. The identifier ident is compared to aexpr3. The following expression is evaluated:

   (ident - aexpr3) * SIGN (tvar) ≤ 0

   where SIGN is a function that returns -1 for negative values, +1 for positive values, and 0 for zero values. When tvar is positive, the expression is true if ident does not exceed aexpr3. When tvar is negative, the expression is true if ident is not less than aexpr3. When tvar is zero, the expression is always true. If the expression is true, the statement part of the FOR statement is executed. If the expression is false, the statement part is not executed, and control passes to the statement after the FOR statement.

4. The value of aexpr2 is recomputed and assigned to tvar.

5. The value of tvar is added to that of ident (that is, ident:=ident+tvar).

6. The sequence of operations repeats starting at step 3.

Example:

```
≠FOR≠ I := 3 ≠STEP≠ -6
      UNTIL  -16 ≠DO≠ statement
```

The statement is executed with I equal to 3, -3, -9, and -15.

The statement part of a FOR statement is permitted to change the value of the control variable. This feature can be used to prematurely terminate the execution of the loop.

Example:

```
≠REAL≠ XYZ,A;
≠FOR≠ XYZ := 3.2 ≠STEP≠ 1.57
    ≠UNTIL≠ 13.69 ≠DO≠
    ≠IF≠ BOOL ≠THEN≠ PERF (BOOL,A)
    ≠ELSE≠ XYZ := 19.79
```

In this example, the loop continues executing as long as BOOL is true and XYZ does not exceed 13.69. If PERF changes BOOL to false, however, then the assignment statement is executed, and the next time XYZ is tested it exceeds 13.69, ending execution of the FOR statement. Note that XYZ is incremented before being tested, so that the value of XYZ after the FOR statement is 21.36, not 19.79.

Upon completion of the FOR statement, either through a GOTO statement or by reaching the terminating condition, the control variable ident retains the last value assigned to it at step 5. It is possible for the statement part of a FOR statement containing this kind of FOR clause never to be executed. Also, if aexpr2 (the step value) is 0, the statement executes endlessly unless a GOTO statement transfers control out of the statement.

Example:

The FOR statement:

```
≠FOR≠ J:=1 ≠STEP≠ 1
    ≠UNTIL≠ 4 ≠DO≠ A[1,J]:=B[1,J]
```

performs the following series of assignments:

```
J:=1;A[1,1] :=B[1,1];
J:=2;A[1,2] :=B[1,2];
J:=3;A[1,3] :=B[1,3];
J:=4;A[1,4] :=B[1,4];
J:=5;
```

After execution has completed, J has a value which is the sum of the step value and the terminating value given in the FOR clause.

WHILE FORLIST ELEMENTS

A FOR statement of the form:

```
≠FOR≠ ident := aexpr
    ≠WHILE≠ bexpr ≠DO≠ statement
```

causes the statement part of the FOR statement to be performed zero, one, or more times. The control variable ident may or may not occur in the statement part or in bexpr. Execution of a FOR statement containing this kind of FOR clause causes the following sequence of operations:

1. The variable ident is assigned the value of aexpr.

2. If the current value of bexpr is false, the statement part of the FOR statement is not executed, and control passes to the statement following the FOR statement. If the current value of bexpr is true, the statement part of the FOR statement is executed. The statement can change the value of ident.

3. The sequence of operations repeats starting at step 1.

Upon completion of the FOR statement, either through a GOTO statement or by reaching the terminating condition, the control variable ident retains the last value assigned to it. Either the value of bexpr must always be false whenever execution of the FOR statement begins (in which case the statement part would never execute), or else at some point during execution of the FOR statement it must become false. Otherwise, a FOR statement having a WHILE clause could not terminate normally; a termination by means of a GOTO statement would be required.

Example:

The FOR statement:

```
≠FOR≠ K:=K*2 ≠WHILE≠ K≤M ≠DO≠
    ≠FOR≠ J:=1 ≠STEP≠ 1
    ≠UNTIL≠ N≠DO≠ A[K,J]:=B[K,J];
```

performs the following series of assignments, if when execution begins, K has a value of 1, N has a value of 2, and M has a value of 10:

```
K:=2;   J:=1;  A[2,1] :=B[2,1] ;
        J:=2;  A[2,2] :=B[2,2] ;  J:=3 ;

K:=4;   J:=1;  A[4,1] :=B[4,1] ;
        J:=2;  A[4,2] :=B[4,2] ;  J:=3 ;

K:=8;   J:=1;  A[8,1] :=B[8,1] ;
        J:=2;  A[8,2] :=B[8,2] ;  J:=3;  K:=16;
```

Example:

If the initial value of the Boolean variable FLAG is true, then the FOR statement:

```
≠FOR≠ RESULT:=0 ≠WHILE≠ FLAG ≠DO≠
    ≠BEGIN≠
    PROC(RESULT,FLAG); OUTINTEGER
    (61,RESULT)
≠END≠
```

causes the variable RESULT to be repeatedly set to 0, and the procedure named PROC to be repeatedly called until PROC alters the value of FLAG to false. Each time that control returns from PROC, the variable RESULT is output, reset to 0, and passed again to PROC. The variable FLAG must be passed by name, as explained in section 6; if it is passed by value, this FOR statement can never complete executing, because PROC cannot alter the value of FLAG. Loop execution could also be terminated by a GOTO statement within PROC.

THE STATEMENT PART

The statement part of a FOR statement can be any kind of ALGOL statement, including another FOR statement.

The statement part might not, in some cases, be executed at all. The test for the terminating condition of a WHILE or STEP clause is performed before the statement part has been executed even once. If the condition is met at that time, the statement part is not executed. The statement part is always executed if the FOR clause has the form which merely initializes the control variable.

The statement part is treated like a block (whether or not it has the form of one), so that the labels of the statement part itself and of labeled statements in the statement part have no meaning outside of the statement part. Hence, a GOTO statement can transfer control out of, but not into, a statement part. The fact that the statement part is treated like a block does not affect the scope of any other identifiers appearing in it.

CONDITIONAL STATEMENT

A conditional statement either dynamically selects the execution of a statement from among a group of two or more statements, or else dynamically selects whether or not a specified statement is to be executed at all. The selection depends on the current value of one or more Boolean expressions. The IF statement has either of the following forms:

```
≠IF≠ bexpr ≠THEN≠ statement 1
≠IF≠ bexpr ≠THEN≠ statement 2 ≠ELSE≠ statement 3
```

where bexpr is a Boolean expression; statement 1 is any statement except another conditional statement; statement 2 is any statement except a conditional or FOR statement; and statement 3 is any statement. The first form allows a selection between performing and not performing statement 1. The second form allows a selection between statement 2 and statement 3, which might be another conditional statement.

Figure 4-10 shows the syntax of conditional statements.

FIRST FORM

Evaluation of a statement of the form:

```
≠IF≠ bexpr ≠THEN≠ statement 1
```

causes the Boolean expression bexpr to be evaluated. If the value of bexpr is true, statement 1 is executed. If the value of bexpr is false, statement 1 is not executed and control passes to the next statement.

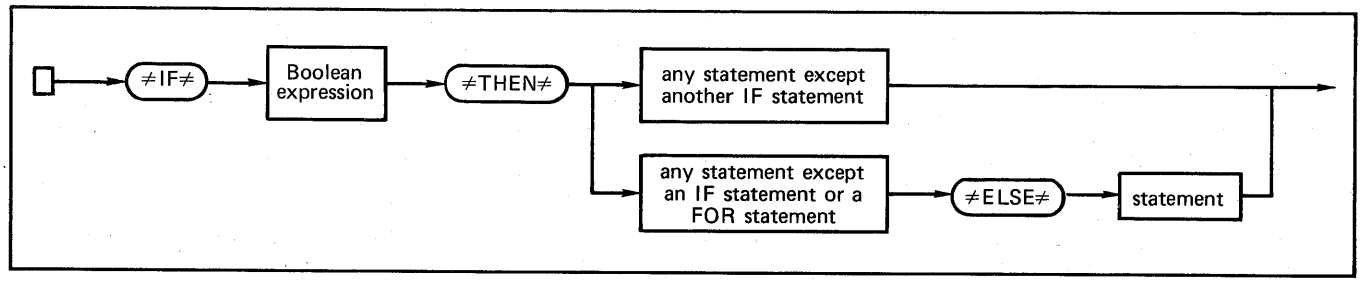


Figure 4-10. Syntax of Conditional Statements

Example:

In the program segment:

```
≠IF≠ X>0 ≠THEN≠ N:=N+10;
A[N]:=NUM=30.0;
```

the Boolean expression X>0 is evaluated. If the value of X is greater than 0, N is incremented by 10 before the array A is accessed in the next statement. If the value of X is less than or equal to 0, N retains the value it had before the IF statement was evaluated; and control passes to the next statement, which assigns a value to the Boolean array element A[N].

Although statement 1 cannot be a conditional statement, there is no prohibition against its being a compound statement (or block) that contains a conditional statement as its only statement element.

Example:

The statement:

```
≠IF≠ BOOL1 ≠THEN≠
    L:≠BEGIN≠
        ≠IF≠ BOOL2 ≠THEN≠
        A:=B
        ≠END≠
```

is syntactically correct. If the compound statement were not labeled, a simpler way to describe the same actions would be:

```
≠IF≠ BOOL1∧BOOL2 ≠THEN≠ A:=B
```

Bexpr is not evaluated when statement 1 is labeled, and a GOTO statement transfers control to it.

If statement 1 is a FOR statement, it is treated as a block even if it does not have the form of one.

SECOND FORM

Conditional statements of the second form are similar to conditional expressions: both contain the symbols ≠IF≠, ≠THEN≠, and ≠ELSE≠. The difference between the two is that the conditional objects of a conditional statement are statements to be executed instead of expressions to be evaluated. The Boolean conditions in both cases are evaluated only at execution time and the selection of the operations to be performed is made then.

Evaluation of a statement of the form:

```
≠IF≠ bexpr ≠THEN≠ statement 2 ≠ELSE≠ statement 3
```

causes the Boolean expression bexpr to be evaluated. Depending on the value of bexpr, either statement 2 or statement 3 is executed, but not both. If bexpr is true, statement 2 is executed and, unless statement 2 contains a GOTO statement, control then passes directly to the statement that follows the conditional statement. If bexpr is false, statement 3 is executed instead of statement 2. Statement 3 can be any type of statement, including another conditional statement. Statement 2 can be any type of statement except a conditional statement or a FOR statement.

Example:

In the program segment:

```
≠IF≠ N≤V
    ≠THEN≠ Z:=N+M
        ≠ELSE≠ Z:=N;
A[Z]:=NUM-30.0;
```

the Boolean expression N≤V is evaluated. If the value of N is less than or equal to the value of V, the assignment Z:=N+M is performed and control passes to the assignment statement A[Z]:=NUM-30.0. If, instead, the value of N is greater than that of V, N is assigned to Z and control passes to the next statement.

Although statement 2 can be neither a conditional statement nor a FOR statement, there is no prohibition against its being a compound statement (or a block) that contains either a conditional statement or a FOR statement as its only statement.

Example:

The statement:

```
≠IF≠ BOOL1 ≠THEN≠
    ≠BEGIN≠≠FOR≠ Q:=Q+L
        ≠WHILE≠ A[Q]>0
            ≠DO≠ A[Q]:=N/A[Q]
    ≠END≠
    ≠ELSE≠  GOTO  EXIT
```

is syntactically correct. Omitting the ≠BEGIN≠ and ≠END≠ symbols around the FOR statement would yield an incorrect statement.

Declarations serve to define properties of quantities (arrays, switches, procedures, and simple variables) and to associate the quantities with identifiers. Every quantity used in a program must be explicitly declared within the program, except that labels and quantities declared in the standard circumlude are implicitly declared, while formal parameters must be described in the specification part of the procedure heading (section 6).

Declarations appear only at the head of a block. (See Compound Statements and Blocks in section 4.) The scope of a declared quantity is the block at the head of which the declaration appears, minus any inner blocks in which the identifier is redeclared. The quantity is said to be local to that block. Outside of the block in which a declaration appears, that declaration is irrelevant. In effect, the quantity does not exist outside of the block, although its associated identifier can appear outside of the block with a different meaning, since any identifier can be declared in any block. No identifier can be declared explicitly or implicitly more than once at the head of a block.

During execution of a block, when the block is entered through the ≠BEGIN≠, all identifiers declared for the block assume the significance implied by the nature of the declarations given. If these identifiers have already been declared outside, they are for the time being given a new (local) significance. At the time of an exit from a block through an ≠END≠ symbol or by a GOTO statement, all identifiers declared for the block lose their local significance. If these identifiers had already been declared outside, they regain their former significance.

If a declaration is preceded by the ≠OWN≠ symbol, then upon a reentry into a block, the values of quantities declared own have the values they had at the last exit from the block. The values of local quantities not declared own are undefined at reentry. The values of quantities nonlocal to the block are unchanged by entry into or exit from the block. Own variables are initialized to zero if they are arithmetic, and false if they are Boolean.

The ALGOL5 control statement option DB=P (section 11) and the execution control statement option Z (section 13) can be used to initialize non-own variables (including array elements).

Not every identifier within an inner block of the program need be declared at the head of that block. Those that are not have the same meanings that they possess outside of the block.

TYPE DECLARATION

A type declaration must be used to specify:

- The type of a simple variable (integer, real, or Boolean)
- The identifier of the simple variable in that block

The variable can have only one type in any particular block. In the declaration, the variable is referred to by the name that must be used to refer to it throughout the block.

The form of a simple variable type declaration is any of the following:

```
≠REAL≠ list
≠INTEGER≠ list
≠BOOLEAN≠ list
≠OWN≠≠REAL≠ list
≠OWN≠≠INTEGER≠ list
≠OWN≠≠BOOLEAN≠ list
```

where list is one or more single variable identifiers separated by commas.

The symbol ≠REAL≠ indicates that any identifier in the list following it can only assume a real value (it cannot be given the value true, for instance). The symbol ≠INTEGER≠ indicates that any identifier in the list following it can only assume an integral value. The symbol ≠BOOLEAN≠ indicates that any identifier in the list following it can only assume a value of true or false.

Examples:

```
≠INTEGER≠ X

≠REAL≠ A, BOOBY, CATS

≠OWN≠ ≠INTEGER≠ K, M, W

≠BOOLEAN≠ TRUE, F, FLAG
    (The variable TRUE is independent of the symbol
    ≠TRUE≠ unless reserved word mode has been
    specified by the RES option on the ALGOL5
    control statement, in which case TRUE cannot
    be used as an identifier.)
```

Figure 5-1 shows the syntax of type declarations.

The symbol ≠OWN≠ indicates that any identifier in the list following will upon reentry into the block retain whatever value it possessed at the last exit from the block. Any variable declared own is also initialized to zero (for types integer and real) or false (for type Boolean), at the first entry to the block.

Example (in reserved word mode):

```
BEGIN
    OWN   INTEGER  I;
  I := I + 1;
    IF  I > N  THEN   GO TO  BYE
END
```

This example consists of one block that increments the value of I by 1 each time the block is executed. The first time the block is entered, I has the value zero when execution reaches the statement I:=I+1, and the value 1 when the block ends. The second time the block is entered, I has the value 1 on entry and 2 on exit, and so forth.

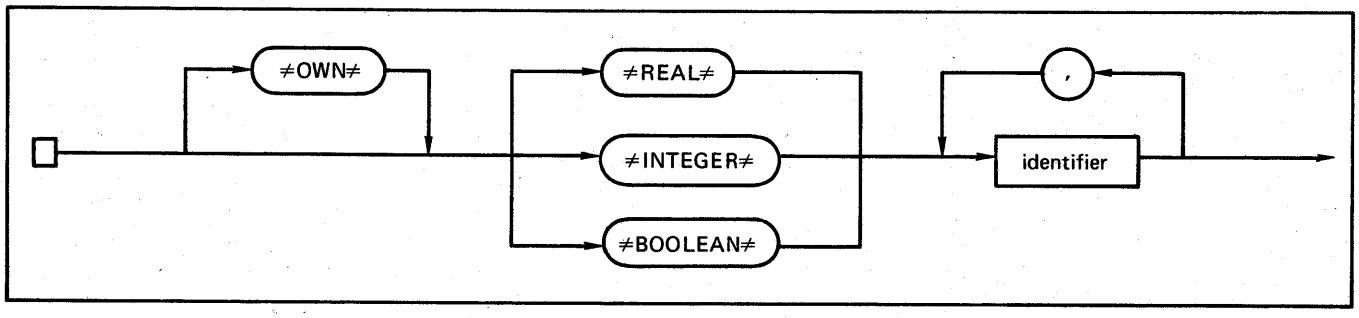


Figure 5-1. State Diagram for Variable Declarations

ARRAY DECLARATION

An array declaration must be used to specify:

- The type of an array
- The identifier of the array in that block
- The number of dimensions of the array
- The bounds of the array (that is, the size of each dimension)

From these specifications the number of elements in the array and the order of subscripted variables identifying the array elements can be derived. The array can have only one type in any particular block. In the declaration, the array is referred to by the name that must be used to refer to it throughout the block.

The form of an array declaration is one of the following:

≠ARRAY≠ list
type ≠ARRAY≠ list
≠OWN≠ type ≠ARRAY≠ list
≠OWN≠ ≠ARRAY≠ list

where type is the type of the array, real, integer, or Boolean, and list is one or more items of the following form, separated by commas:

names [limits]

Names is a list of one or more array identifiers separated by commas, and limits is a list of pairs of arithmetic expressions giving the upper and lower bounds of each dimension of the array. Limits has the form:

low:high, low:high, . . . , low:high

where low is the lowest value and high is the highest value that a subscript in the corresponding position in a bracketed subscript list can have. The values of all bound pairs are evaluated once at each entry into the block. The maximum absolute value for a lower or upper bound is $2^{29} - 1$ (= 536 870 991).

The number of pairs in the bracketed list is the number of dimensions in the array. The first pair gives the bounds for the first dimension, the second pair for the second dimension, and so forth.

If no type is given in the declaration, real is assumed.

Example:

The following declarations are identical in effect:

≠REAL≠ ≠ARRAY≠ M[2:20]

≠ARRAY≠ M 2:20

Figure 5-2 shows the syntax of array declarations.

If the array declaration is immediately preceded by a comment of the following form:

≠COMMENT≠ ≠VIRTUAL≠;

then all the arrays declared are assumed to be virtual arrays. The comment directive only applies to one array declaration; thus, either all the virtual arrays in a block must be declared in the same declaration, or the same comment must precede each declaration of a virtual array. Virtual arrays are discussed in section 2, under Arrays.

Virtual arrays cannot be declared to be own arrays, regardless of the setting of the V control statement option.

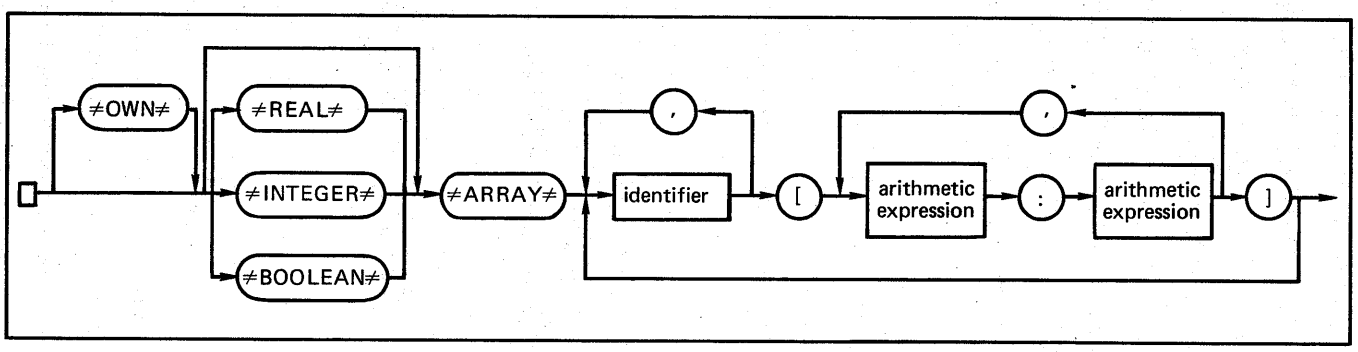


Figure 5-2. State Diagram for Array Declarations

The array is undefined when the value of any upper bound is less than the value of the accompanying lower bound. The comment directive ≠CHECKON≠ can be used to check that array references are within bounds (see section 10).

When more than one array name precedes a bracketed list in an array declaration, the dimensions apply to each of the arrays in the list. The type and own specifications apply to every array in the declaration.

Example:

The declaration:

≠ARRAY≠ A, B, C[7:N, 2:M], S[-2:10]

specifies the properties of four arrays: A, B, C, and S. All four arrays are type real. A, B, and C have two dimensions, with subscripts ranging from 7 to N and from 2 to M, respectively; S has one dimension, with subscripts ranging from -2 to 10. None of the arrays is an own array.

Example:

The declaration:

≠OWN≠ ≠BOOLEAN≠ ≠ARRAY≠ STOP[1:3],

FLAG, SWITCHES 0:15

specifies the properties of three arrays: STOP, FLAG, and SWITCHES. All three arrays are Boolean and own, and have one dimension. The subscripts for STOP range from 1 to 3, and for FLAG and SWITCHES from 0 to 15.

Example:

The declaration:

≠REAL≠ ≠ARRAY≠ Q[1: ≠IF≠ C<0
≠THEN≠ 10.3 ≠ELSE≠ 20.6]

specifies the properties of an array Q. The array is real and non-own, with one dimension. The lower subscript bound is 1, and the upper bound is either 10 (rounded from 10.3) or 21 (rounded from 20.6), depending on whether or not C is negative.

The expressions that define the upper and lower bounds of the dimensions of an array must not include any identifier that is declared, implicitly or explicitly, in the same block head as the array. Also, the expressions that define the bounds of an own array must be integer numbers.

The relative location of a subscripted variable within an array can be determined from its subscript and the array declaration. Table 5-1 shows the relative offset for a given subscripted variable when the array declaration is shown. The table shows the offset for one, two, and three dimensions; the case for a higher number of dimensions can be calculated inductively. For the sake of simplicity, the table assumes that all subscripts and lower and upper bounds are integers. The offset is the ordinal of the array element; thus, the offset of A[7] is 7 when the declaration is A[1:10].

SWITCH DECLARATION

A switch declaration must be used to specify:

- The identifier of a switch in the block
- The switch elements and their sequence

The switch is referred to in the declaration by the name that must be used to refer to it throughout the block at the head of which the declaration appears.

The form of a switch declaration is as follows:

≠SWITCH≠ switchid := list

where switchid is a switch identifier, and list is one or more designational expressions separated by commas. The list of designational expressions is the switch list for switchid. One switch is defined per declaration; to define several switches, several declarations need to be made. A switch cannot be declared own.

Example:

The declaration:

≠SWITCH≠ TRAINS := SOUTH, NORTHEAST,
MIDWEST

specifies that the switch named TRAINS has three elements. The switch designator SWITCH[1] has the value SOUTH; SWITCH[2] has the value NORTHEAST; SWITCH[3] has the value MIDWEST; SWITCH[4] and SWITCH[0] are undefined.

Figure 5-3 shows the syntax of switch declarations.

A designational expression in the switch list is evaluated when a switch designator identifying one of the expressions is evaluated during program execution. The current values of any variables in the expression are used.

TABLE 5-1. RELATIVE OFFSET OF SUBSCRIPTED VARIABLES

Array Declaration	Subscripted Variable	Offset
A [L1:U1]	A [M]	M - L1 + 1
A [L1:U1 L2:U2]	A [M, N]	N - L2 + 1 + (M - L1) * (U2 - L2 + 1)
A [L1:U1, L2:U2, L3:U3]	A [M, N, P]	P - L3 + 1 + (N - L2 + (M - L1) * (U2 - L2 + 1)) * (U3 - L3 + 1)

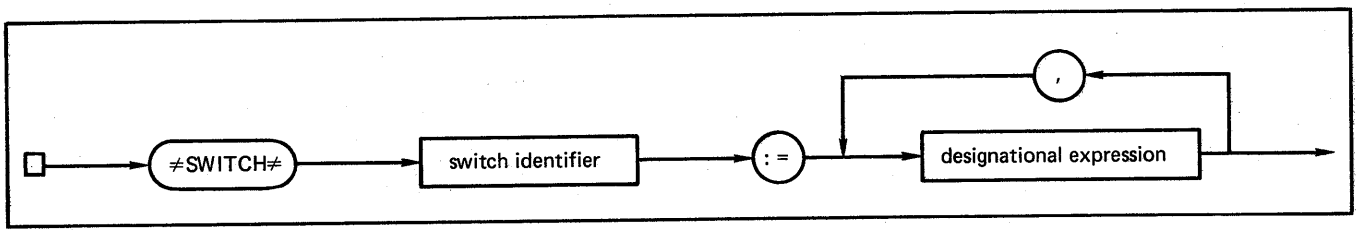


Figure 5-3. State Diagram for Switch Declarations

Example:

The declarations:

```
≠SWITCH≠ Q := S1, S2, S[M],
    ≠IF≠ V>-5 ≠THEN≠ S3 ≠ELSE≠ S4
≠SWITCH≠ S := PARIS, VIENNA, TRIESTE,
    VENICE, TORINO, AOSTA
```

specify that the switch named Q has four elements and the switch named S has six elements. Q[3] is the switch designator S[M]; when a reference is made to Q[3], the current value of M is used to select a designational expression from the switch list.

PROCEDURE DECLARATION

A procedure declaration is used to specify:

- The body of the procedure (the actions it performs)
- The procedure identifier
- The type of the procedure identifier if the name is used in a function designator
- Procedure parameters
- Parameter specifications

The declaration of procedures is described in section 6.

Procedures allow the user to execute the same sequence of code in different parts of the program without respecifying the code each time. A procedure is declared at the head of the block in which it is to be used, and brought into execution either by a procedure statement, or by the use of its name as a function designator in an expression. The body of the procedure, which contains the executable instructions, can be provided in one of two ways:

- As an ALGOL statement, in the procedure declaration. In this case, the procedure can only be used within the block in which it is declared.
- As a separately compiled procedure. A declaration must still appear in the block which is to use the procedure, but the body itself is compiled separately. In this way, more than one program can use the same procedure.

Separately compiled procedures are described in section 9; this section describes procedures declared completely within the ALGOL program. It also describes how procedures are called, which is the same for both kinds of procedures.

PROCEDURE DECLARATIONS

The syntax of a procedure declaration is shown in figure 6-1. A procedure declaration consists of an optional type identification (≠REAL≠, ≠INTEGER≠, or ≠BOOLEAN≠), the symbol ≠PROCEDURE≠, a procedure heading and a procedure body, in that order. The procedure body is either a statement (section 4) or a code part. A code part applies to separately compiled procedures, and is defined in section 9.

The procedure heading consists of either the procedure identifier followed by a semicolon, or the procedure identifier followed by a formal parameter part, a semicolon, an optional value part, and a specification part.

The procedure identifier is the name by which the procedure is brought into execution in the block in which it is declared.

The formal parameter part names the formal parameters in the same order as in the procedure call. The maximum number of formal parameters is 254. When the procedure is called, an actual parameter is substituted for each formal parameter. The formal parameters are identifiers; they are separated either by commas or by parameter delimiters of the form:

) letter string : (

where letter string is a series of letters (no blanks allowed). This form is logically equivalent to a comma; the other characters are documentary only and have no effect on the program.

No identifier can occur twice in the formal parameter list. The name of the procedure being declared cannot occur in the list.

The value part (if included) consists of the symbol ≠VALUE≠, a list of identifiers, and a semicolon. If present, the value part specifies that each of the parameters in the list is call-by-value, rather than call-by-name. Only formal parameters can appear in this list. Call-by-value and call-by-name are explained under Procedure Calling.

Virtual arrays cannot be called by value.

The specification part is similar to the declaration part of a block, in that it provides type and kind information. A specification part, however, only applies to the formal parameters of a procedure. The specifiers from ≠STRING≠ to ≠SWITCH≠ (as they are listed in figure 6-1) specify type and kind; no formal parameter can be specified by more than one of these specifiers (type ARRAY and type PROCEDURE each count as one specifier). All formal parameters must be specified.

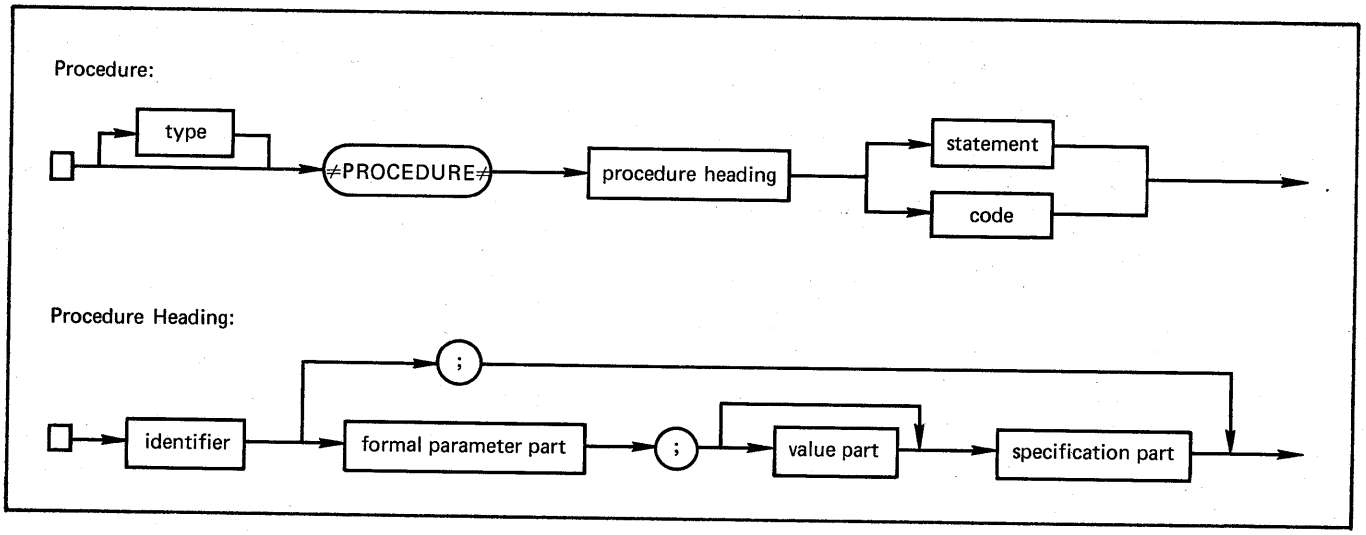
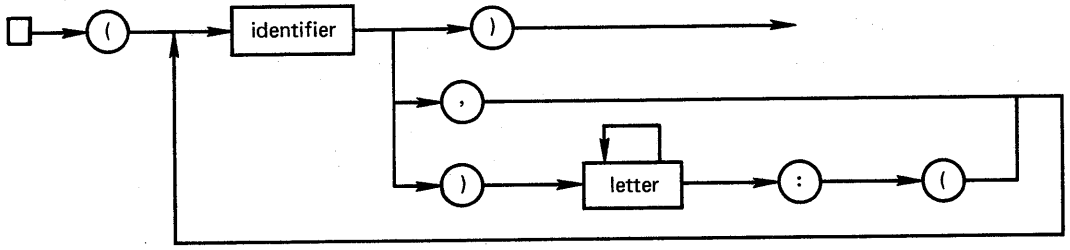
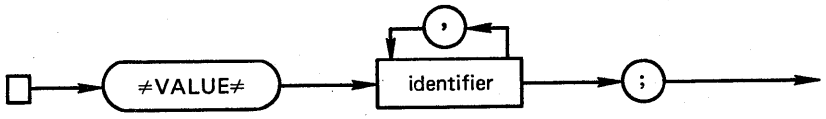


Figure 6-1. Syntax of Procedure Declaration (Sheet 1 of 2)

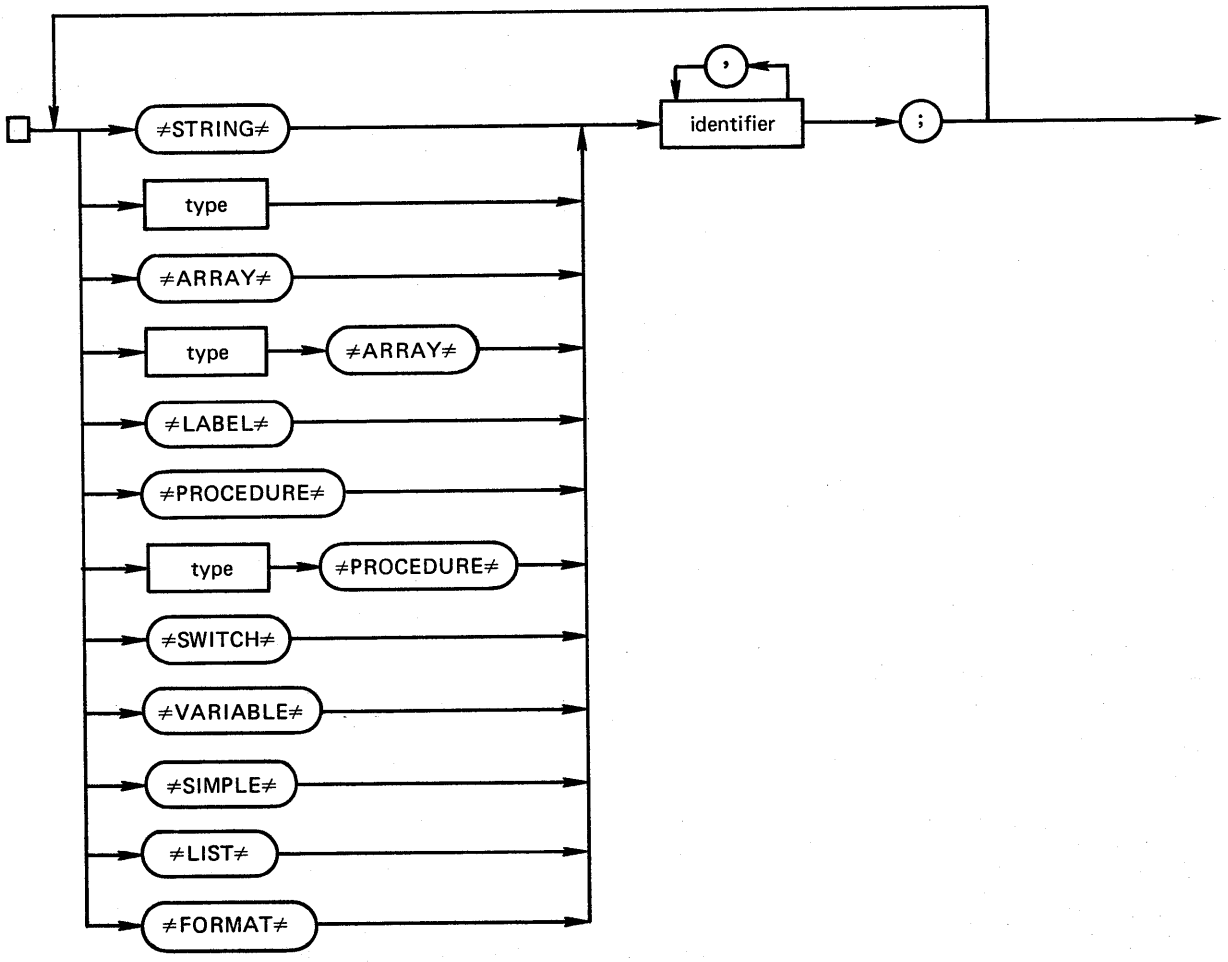
Formal Parameter Part:



Value Part:



Specification Part:



Code: See figure 9-1

Figure 6-1. Syntax of Procedure Declaration (Sheet 2 of 2)

The type and kind specifiers restrict the type and kind of the actual parameters that can correspond to the specified formal parameters. This correspondence is explained under Procedure Calling. The remaining specifiers (≠VARIABLE≠, ≠SIMPLE≠, ≠LIST≠, and ≠FORMAT≠) are additional specifiers that can only be used with call-by-name parameters to code procedures. They allow the compiler to generate more efficient code in certain special cases.

The specifier ≠VARIABLE≠ indicates that a value is assigned to the identifier somewhere in the procedure.

The specifier ≠SIMPLE≠ indicates that the identifier is used as the control variable in a FOR statement.

Formal parameters specified as ≠VARIABLE≠ or ≠SIMPLE≠ must also appear in a ≠REAL≠, ≠INTEGER≠, or ≠BOOLEAN≠ specification.

The specifier ≠LIST≠ indicates that instead of a single parameter, the procedure can be called with zero or more actual parameters, occupying the position of the identifier. The actual parameters can be of any type; mixing types is allowed. Only one parameter can be specified by ≠LIST≠, and it must be the last parameter. The procedure must be written in FORTRAN or COMPASS. If the body of the procedure is written in COMPASS, it can determine the number of actual parameters through the PARAMS macro (section 12). No other specification is allowed for the parameter.

Example:

```
≠PROCEDURE≠ A (B,C,D); . . .
≠LIST≠ D; ≠INTEGER≠ B, C;
```

allows both of the following procedure calls:

```
A (2,3);
A (2,3,4.2,6.3,7,8.4);
```

In the second case, the list items are of both real and integer type.

The specifier ≠FORMAT≠ indicates that the parameter is used only as a format string in the procedure (section 8). Use of this specifier allows compilation time checking of format string syntax. No other specification is allowed for the parameter.

If an array specification is immediately preceded by a comment of the following form:

```
≠COMMENT≠ ≠VIRTUAL≠;
```

then all the arrays specified are assumed to be virtual arrays. The comment directive only applies to one array specification; thus, either all the virtual arrays in a block must be specified in the same specification, or the same comment must precede each specification of a virtual array. Virtual arrays are discussed in section 2, under Arrays.

For a procedure whose body is a statement (that is, not a separately compiled procedure), the statement acts as a block even if it does not have the form of one. In particular, labels within the procedure have no scope outside of the procedure.

Identifiers in the procedure body that are not formal parameters are local if they are declared within the procedure, otherwise they are nonlocal. If they are nonlocal their scope must include at least the block in which the procedure is declared. If the identifier of a formal parameter is redeclared within a procedure body, it loses its significance as a formal parameter and is, in effect, a different identifier. The actual parameter corresponding to it ceases to be accessible as long as the new identifier is active.

If the procedure identifier is used as a function designator, a type is required before the symbol ≠PROCEDURE≠. Also, the identifier must appear in the procedure body (without declaration or specification) as the destination of an assignment statement at least once, and at least one such statement must be executed. The last value assigned to this identifier during the execution of the procedure is the value returned by the procedure in the expression in which it is called. If the procedure identifier appears in any other context within the body of the procedure, it indicates a recursive call to the procedure. If a GOTO statement in the procedure causes an exit from the procedure before a value has been assigned to the procedure identifier, the value of the function designator, and hence the value of the expression containing it, is undefined.

Figure 6-2 shows an example of a procedure declaration. The procedure A interchanges the first C elements of the arrays D and E. D is a real array, and E is an integer array; when an element of E is assigned to D, it is converted to real. The Boolean flag B is used to signal if

```
≠PROCEDURE≠ A(B) COUNTER: (C) ARRAYS: (D,E);
   ≠VALUE≠ C;
   ≠REAL≠ ≠ARRAY≠ D; ≠INTEGER≠ ≠ARRAY≠ E;
   ≠INTEGER≠ C; ≠BOOLEAN≠ B;
   ≠BEGIN≠
      ≠REAL≠ ≠ARRAY≠ F [1:C]; ≠INTEGER≠ I;
      B := ≠FALSE≠;
      ≠FOR≠ I := 1 ≠STEP≠ 1 ≠UNTIL≠ C ≠DO≠
      ≠BEGIN≠
         F[I] := D[I];
         D[I] := E[I];
         ≠IF≠ ABS (F[I]) > MAXINT ≠THEN≠ B := ≠TRUE≠;
         E[I] := F[I]
      ≠END≠
   ≠END≠;
```

Figure 6-2. Procedure Declaration Example

any of the elements of D are out of range, that is, larger than the system-defined function designator MAXINT (section 7). If so, B is set to true. The array F, which is local to the procedure, is used to hold the elements of D while the elements of E are assigned to D. In its declaration, the formal parameter C is used as the upper bound.

PROCEDURE CALLING

A procedure is called in one of two ways:

- In a procedure statement. The syntax of this statement is shown in figure 6-3.
- Through a function designator occurring in an expression. The syntax of this construction is the same as the procedure statement, figure 6-3.

In both cases, the steps followed in execution of the procedure are the same. These steps can be outlined as follows:

1. For each formal parameter in the procedure declaration defined as call-by-value, the corresponding actual parameter is evaluated and the value is assigned to a fictitious variable. The fictitious variable, which acts like a variable declared in a fictitious block that includes only the procedure body, is then substituted for the formal parameter in each occurrence. The net result is that assignment to call-by-value formal parameters is valid within the body of the procedure, but has no effect on the values of the actual parameters. When actual parameters corresponding to call-by-value formal parameters are evaluated, the non-arrays are evaluated first, from left to right, and then the arrays are evaluated from left to right.
2. For each remaining formal parameter, all of which are call-by-name, the entity provided as the actual parameter (that is, the actual characters making up the parameter as it appears in the call) is substituted for each occurrence of the formal parameter in the procedure. If necessary for correct evaluation, the actual parameter is treated as if it were surrounded by parentheses. If an identifier provided as an actual parameter is the same as an identifier already present in the procedure body, and they represent two different quantities, they are treated as two separate identifiers.
3. The procedure body, with the actual parameters substituted for the formal parameters, is executed.

A procedure can call itself, either directly or through another procedure.

In the example in figure 6-4, when procedure B is called, the first actual parameter is evaluated, since it corresponds to C, which is a call-by-value parameter. The value of A+3, which is 7, is assigned to C. For the second parameter, which is a call-by-name parameter, the expression E+2 is substituted for D in the OUTINTEGER call. Then the body of the procedure is executed: 17 is assigned to E, the first OUTINTEGER call writes the integer 7, and the second OUTINTEGER call writes 19. If D were call-by-value instead of call-by-name, the expression E+2 would be evaluated before entry into the procedure, and the value 14 would be output by the second OUTINTEGER call.

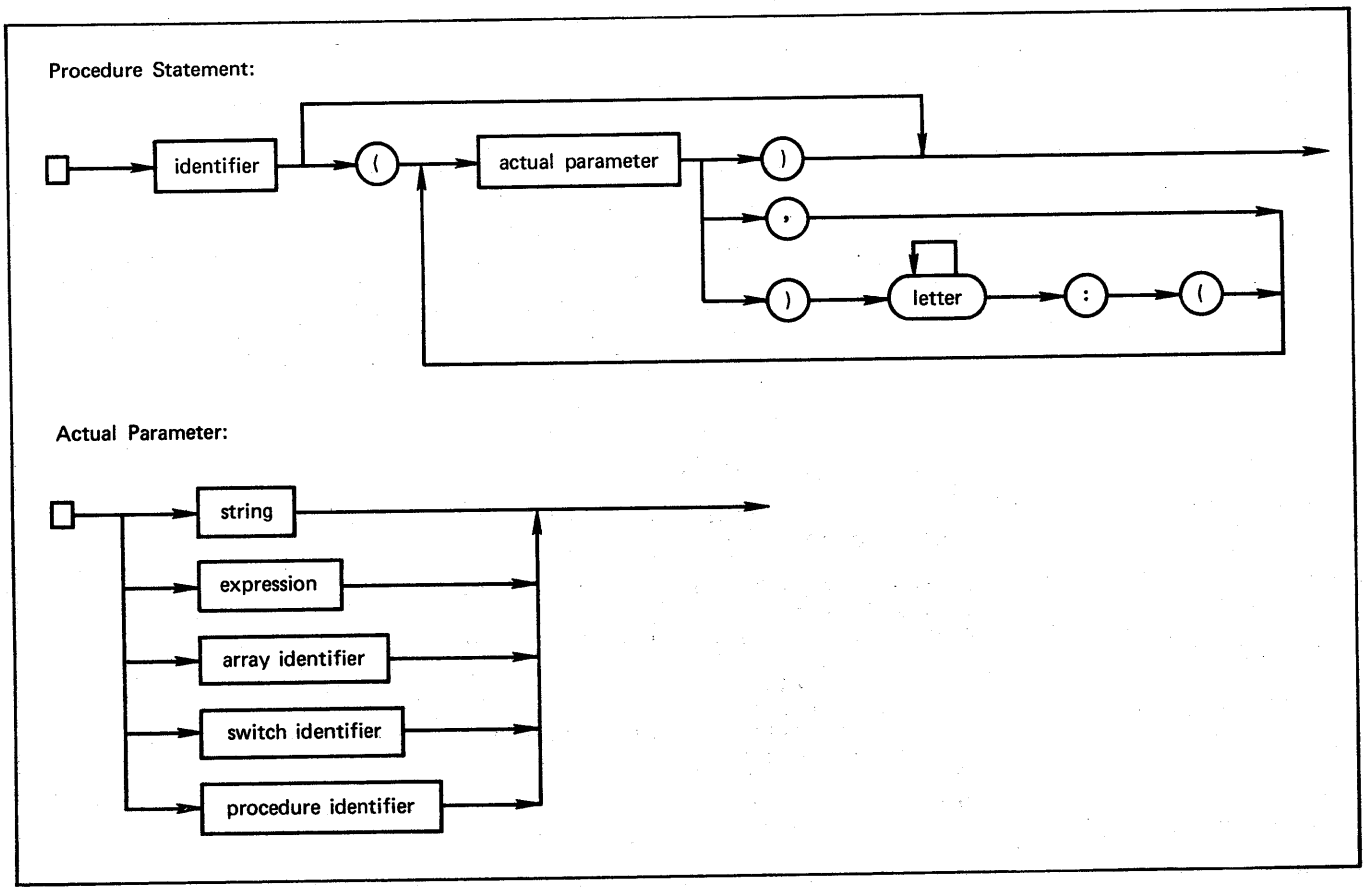


Figure 6-3. Procedure Statement Syntax

```
≠BEGIN≠ ≠INTEGER≠ A,E;
        ≠PROCEDURE≠ B(C,D);≠VALUE≠ C;
        ≠INTEGER≠ C,D;
        ≠BEGIN≠
                E:=17;
                OUTINTEGER (61,C);
                OUTINTEGER (61,D)
        ≠END≠;
        E := 12;
        A := 4;
        B (A+3,E+2)

≠END≠
```

Figure 6-4. Parameter Substitution Example

The type and kind of the formal parameter restrict the allowable types and kinds of the actual parameter. Table 6-1 lists the kinds of actual parameters allowed as well as the mode (call-by-value or call-by-name) allowed for each kind of formal parameter. In addition to these restrictions, the following rules must be observed:

- If a string is an actual parameter to a procedure, then the string can only be used within the procedure as a parameter to another procedure. This restriction does not apply to separately compiled procedures not written in ALGOL. Eventually, a string must be a parameter to a procedure not written in ALGOL, such as a standard procedure.

- If a value is assigned to a call-by-name formal parameter, the corresponding actual parameter must be a variable, rather than any other kind of expression. A value assigned to a call-by-value formal parameter changes the parameter value only within the procedure.

- A formal parameter used as the control variable of a FOR statement must correspond to an actual parameter that is a simple variable (not a subscripted variable or any other kind of expression).

- A formal parameter specified as an array must correspond to an actual parameter that is an array with the same number of dimensions.

  If the array is call-by-name, then the declaration for the actual parameter array currently in effect is used in the procedure for calculating subscripts based on the declared array bounds. If the array is call-by-value, a local copy of the array is made, and the array bounds used for calculating subscripts are the same as for the actual parameter array. Virtual arrays cannot be call-by-value.

- The parameters to a procedure that is itself an actual parameter must be compatible with the parameters to the corresponding formal parameter procedure in number, kind, and type.

- A call-by-value formal parameter cannot correspond to an actual parameter that is a label, switch identifier, or string. It also cannot correspond to an actual parameter that is a procedure identifier, with the exception of a procedure with no parameters that is used as a function designator. In this latter case, the procedure identifier is considered to be an expression. A function designator with parameters is also an expression and can be an actual parameter.

- If a formal parameter is specified to be a virtual array (as explained above, under Procedure Declarations), then the corresponding actual parameter must also be a virtual array.

A switch parameter in a procedure must be called by name. When the procedure is called, the actual expressions in the switch list in the calling block are substituted for switch designators of the formal parameter switch in the called procedure. When a switch designator is referred to, the current value of the expression in the switch list in the calling block is used.

TABLE 6-1. ACTUAL-FORMAL PARAMETER CORRESPONDENCE

Formal Parameter	Mode	Actual Parameter
Integer	Value	Arithmetic expression
	Name	Integer expression
Real	Value	Arithmetic expression
	Name	Real expression
Boolean	Value	Boolean expression
	Name	Boolean expression
Label	Value	Designational expression
	Name	Designational expression
Integer array	Value	Arithmetic array
	Name	Integer array
Real array	Value	Arithmetic array
	Name	Real array
Boolean array	Value	Boolean array
	Name	Boolean array
Typeless procedure	Name	Arithmetic procedure, or typeless procedure, or Boolean procedure
Integer procedure	Name	Integer procedure
Real procedure	Name	Real procedure
Boolean procedure	Name	Boolean procedure
Switch	Name	Switch
String	Name	Actual string or string identifier
Virtual array	Name	Virtual array

In the example in figure 6-5, the variable N is nonlocal to procedure B. When B is called, N has the value 5. However, when the GOTO statement is executed, N has the value 3. Since the designational expression X[N] has effectively been substituted for S[1], the branch is to the designational expression X[3], not X[5].

```
≠PROCEDURE≠ A(N); ≠INTEGER≠ N;
      .
      .
      .
≠BEGIN≠
≠SWITCH≠ Q := X[N], L;
≠PROCEDURE≠ B(S);
            ≠SWITCH≠ S;
                  .
                  .
                  .
            N := 3;
            ≠GOTO≠ S[1];
            ≠END≠;
                  .
                  .
                  .
N := 5;
B(Q);
      .
      .
      .
```

Figure 6-5. Parameter Switch Example

The procedure whose declaration is shown in figure 6-2 is called by the following program segment:

```
≠REAL≠≠ARRAY≠ M[1:32];
≠INTEGER≠ ≠ARRAY≠ N[1:32];
≠REAL  J; ≠BOOLEAN≠ BOO; . . .
A (BOO, J**2,M,N);
. . .
```

The following correspondence is established:

Actual	Formal
BOO	B
J**2	C
M	D
N	E

The formal parameter C is called by value; therefore, J**2 is evaluated when the procedure is called, and the result is assigned to a fictitious variable that is substituted for every occurrence of C in the procedure. The other parameters are call-by-name; the names BOO, M, and N are substituted for B, D, and E respectively. The procedure is then executed. Note that J is declared as real in the calling program segment, but is specified as an integer (C) in the procedure. This is valid, since C is called by value. The expression J**2 is converted to integer after it is evaluated.

STANDARD PROCEDURES 7

In addition to the procedures provided by the user, a set of standard procedures which require no definition by the user is available. These procedures are declared in the standard circumlude. See section 9 for a description of circumludes.

Standard procedures fall into the following categories:

- Simple functions. These are miscellaneous functions that perform simple operations on expressions, such as returning the absolute value, truncating a real number, returning the length of a string, and so forth.
- String manipulation procedures. These perform functions such as converting strings to integers, selecting characters from strings, and so forth. They are most commonly used with input/output procedures.
- Mathematical functions. These include the trigonometric, exponential, and logarithmic functions.
- Error handling and terminating procedures. These specify error conditions to be checked and actions to be taken, and provide for program termination in the normal and error cases. The procedure DUMP, which is related to this category, is discussed in section 14.
- Environmental inquiry and control procedures. These include procedures to return the value of variables relevant to the execution time environment of the program and set new values for some of these variables.

In addition, standard procedures are provided for input/output; these are discussed in section 8.

In the tables of this section, the following abbreviations are used to specify the types of actual arguments allowed for each procedure:

ae	Arithmetic expression.
re	Real expression. An integer expression is allowed, but is converted to real.
ie	Integer expression. A real expression is allowed, but is rounded to the nearest integer.
iv	Call-by-name integer variable (simple or subscripted). A value is assigned to this argument.
a	Array of any type. The corresponding formal parameter is call-by-name.
de	Designational expression.
s	String.

For each procedure, the table also shows the type of value returned by the procedure if it can be called as a function. If the type is listed as none, the procedure cannot be called as a function. Some procedures, such as MAXREAL, have no parameters; a reference to the procedure name in an expression results in a call to the procedure unless the name has been redeclared. Table 7-1 shows the simple functions. Table 7-2 shows the string manipulation functions. Table 7-3 shows the mathematical functions. Table 7-4 shows the environmental inquiry procedures. Table 7-5 shows the error handling and termination procedures. Table 7-6 shows the procedure MOVE.

The procedures from PROGRAMSIZE to MAXIMUMFIELDLENGTH (table 7-4) return the number of words of memory currently allocated for different components of the execution time field length. The arrangement of components of memory at execution time is explained in appendix E.

For the procedure MEMORY (table 7-4), the amount of field length obtained is constrained by the values of the variables MOPTION and IOPTION. These variables are defined in the standard circumlude. MOPTION is the maximum field length that can be requested by a job during execution, whether by user request through MEMORY or by the ALGOL execution time system. IOPTION is the minimum increment of field length that can be requested. Thus, if ae words are requested, fl is the current field length, and minfl is the minimum field length required for the current job step, the amount actually obtained is given by the formulas:

ae>fl:	minimum (maximum (ae, fl + IOPTION), MOPTION)
ae = fl:	fl
ae<fl:	maximum (ae, minfl)

TABLE 7-1. SIMPLE FUNCTIONS

Procedure Call	Type of Function	Action
ABS (re)	Real	Return absolute value of expression.
IABS (ie)	Integer	Return absolute value of expression.
SIGN (ae)	Integer	Return -1 if ae < 0.0 0 if ae = 0.0 +1 if ae > 0.0
ENTIER (re)	Integer	Return largest integer not greater than re; re -1 <ENTIER(re) ≤ re.

MOPTION and IOPTION can be set by the M and I parameters, respectively, on the execution control statement (section 13). The default values are maximum field length for MOPTION, and 1024 for IOPTION.

The error keys for ERROR (table 7-5) are as follows:

0	Any of the following errors
1	Arithmetic overflow (use of infinite or indefinite value)
2	Subscript out of range for array or switch
3	Parameter mismatch
4	Standard function parameter error; error in array specification or subscripting detected by MOVE procedure
5	Stack overflow
6	Error in the call to STRINGELEMENT

TABLE 7-2. STRING MANIPULATION PROCEDURES

Procedure Call	Type of Function	Action
EQUIV (s)	Integer	Return integer equivalent of string. (See A format, section 8.) Only the first eight characters are processed.
LENGTH (s)	Integer	Return number of characters in string.
CHLENGTH (s)	Integer	Same as LENGTH.
STRINGELEMENT(s1,ie,s2,iv)	None	Locate character by counting ie character positions from left in string s1. Search for this character in string s2, and set iv to character position number of leftmost occurrence of character in string. Set iv to 0 if character not found.

TABLE 7-3. MATHEMATICAL FUNCTIONS

Procedure Call	Type of Function	Action
SQRT (ae)	Real	Return square root of ae. Fatal error if ae is negative.
SIN (ae)	Real	Return sine of ae (ae is in radians).
COS (ae)	Real	Return cosine of ae (ae is in radians).
TAN (ae)	Real	Return tangent of ae (ae is in radians).
ARCSIN (ae)	Real	Return principal value, in radians, of arcsine of ae. $-\pi/2 \leq$ ARCSIN(ae) $\leq \pi/2$
ARCCOS (ae)	Real	Return principal value, in radians, of arccosine of ae. $0 \leq$ ARCCOS(ae) $\leq \pi$
ARCTAN (ae)	Real	Return principal value, in radians, of arccosine of ae. $-\pi/2 \leq$ ARCTAN(ae) $\leq \pi/2$
LN (ae)	Real	Return natural logarithm of ae. Fatal error if ae $\leq$ 0.
EXP (ae)	Real	Return exponential function of ae.

The procedure MOVE (table 7-6) transfers all or part of an array to another array. The arrays can be both virtual, both non-virtual, or mixed. The parameter a_1 is the name of the array from which the transfer is to take place; the parameter a_2 is the name of the array to which the transfer is to take place. The last parameter (ie_n), is the number of elements to be transferred. The parameters ie_1, ... , ie_i indicate the subscripts of the first element of the array a_1 to be transferred. The parameters ie_j, ... , ie_k indicate the subscripts of the first element of array a_2 to which the values are to be transferred.

Example:

```
MOVE (ARR1,2,3,ARR2,5,7,9,18);
```

transfers 18 elements of ARR1, beginning at ARR1(2,3) to ARR2, beginning at ARR2(5,7,9).

For successful execution of MOVE, the following conditions must be true:

- A and B must be of the same type.
- The number of subscripts provided in the call (1 through i and j through k) must be the same as the dimensionality of a_1 and a_2, respectively.
- The array elements specified as the starting locations must be within the bounds of their respective arrays.
- The number of elements to be transferred must be within the bounds of both arrays.

Diagnostics are issued for all of these conditions, as well as when an extended memory parity error occurs. The ERROR procedure, invoked with key 4, may be used to trap invalid move operations, recover from them, and continue program execution normally.

TABLE 7-4. ENVIRONMENTAL INQUIRY PROCEDURES

Procedure Call	Type of Function	Action
MAXINT	Integer	Return value of maximum allowable positive integer.
MAXREAL	Real	Return value of maximum allowable real number.
MINREAL	Real	Return value of smallest positive real number distinguishable from zero.
EPSILON	Real	Return the value of the smallest positive real number such that 1.0 + EPSILON > 1.0 and 1.0 - EPSILON < 1.0.
INRANGE (ae)	Boolean	Return false when ae is out of range or indefinite and true otherwise.
LOWERBOUND (a,ie)	Integer	Return the lower bound for the ieth subscript of the array a.
UPPERBOUND (a,ie)	Integer	Return the upper bound for the ieth subscript of the array a.
DATE	Integer	Return current date as eight characters in the form dd/mm/yy (or other form selected by installation option). Value can be output with 8A format.
TIME	Integer	Return the current time as eight characters in the form hh.mm.ss. Value can be output with 8A format.
CLOCK	Real	Return the elapsed central processor time for job in seconds, with a resolution of one millisecond.
PROGRAMSIZE	Integer	Return the current size in words of the loaded program exclusive of all the stacks and the heap.
SCALARSTACK	Integer	Return the current size in words of the scalar stack.
HEAPSIZE	Integer	Return the current size in words of the heap.
FIELDLENGTH	Integer	Return the field length currently allocated for the job.
MAXIMUMFIELDLENGTH	Integer	Return the maximum field length allowed for the job.
MEMORY (ie)	Integer	Request the field length to be set to ie; return the field length actually obtained as value of the function designator. MEMORY(0) sets the field length to the minimum currently needed for the job. Amount obtained is constrained by MOPTION and IOPTION; see text.

TABLE 7-5. TERMINATION AND ERROR HANDLING PROCEDURES

Procedure Call	Type of Function	Action	Procedure Call	Type of Function	Action
STOP	None	Stop execution of program.			label de if an error of type ie occurs. For error codes, see text. If label is not accessible, terminate program.
FAULT (s,ae)	None	Stop execution of program after writing message to channel 61 in the following format: FAULT s ae	ARTHOFLW (de)	None	Branch to label de if arithmetic overflow (use of infinite or indefinite value) occurs.
ERROR (ie,de)	None	Establish that control is to be transferred to	DUMP (ie,ie,s)	None	See section 14.

TABLE 7-6. MOVE PROCEDURE

Procedure Call	Type of Function	Action
MOVE (a_1, ie_1, . . . , ie_i, a_2, ie_j, . . . , ie_k, ie_n)	None	See Text.

ALGOL input/output is accomplished through calls to a set of standard procedures.

All input/output takes place between central memory and a file residing on an external device. The file is known to the operating system by its logical file name, which must have from one to seven letters and digits, the first being a letter. The CHANNEL procedure specifies some of the characteristics of the file, and links the logical file name with a channel number used in all input/output calls in the ALGOL program. The procedure calls that actually perform the input/output specify the channel number as the first actual parameter.

The input/output procedures are of three basic types:

- Coded sequential. Data in internal binary format is converted into coded format. Records are written sequentially; that is, the location of each record is defined only by the locations of the preceding and following records.
- Binary sequential. Data is moved between central memory and the external device without conversion. Records are written sequentially.
- Word addressable. Records are written randomly. Each record is identified by its word address, which is the offset of the record from the beginning of the file.

Other procedures described in this section perform control functions and return system information to the user.

CODED SEQUENTIAL INPUT/OUTPUT

Coded sequential output procedures transmit expression values from central memory and write them in the form of a sequence of characters to a file. The input procedures read sequences of characters into variables in a symmetrical way.

The method of conversion from expression values to character sequence depends on the procedure:

- The procedures INCHAR, OUTCHAR, and OUTSTRING transmit characters and strings with no conversion.
- The procedures ININTEGER, OUTINTEGER, ININTARRAY, OUTINTARRAY, INREAL, OUTREAL, INARRAY, and OUTARRAY transmit values of real and integer type with an implicit conversion that the user cannot control.
- The procedures INBOOLEAN, OUTBOOLEAN, INBARRAY, and OUTBARRAY transmit Boolean values with an implicit conversion that the user cannot control.
- The procedures INLIST, OUTLIST, INPUT, and OUTPUT transmit values of any type; a format string provided by the user determines the way in which conversion takes place.

The procedures INLIST and OUTLIST are the most powerful, enabling the user to specify not only the format but such features as page width and length. In addition, the values to be transmitted are ordered by a list procedure, providing greater execution time control over input/output. For these reasons, the description of these procedures is deferred until after some of the simpler procedures have been described; in this way, the complex capabilities of INLIST and OUTLIST will be clearer.

SIMPLE INPUT/OUTPUT

The most basic input/output procedures are INCHAR and OUTCHAR (figure 8-1).

```
INCHAR (channel, string, destination)
INCHARACTER (channel, string, destination)
INSYMBOL (channel, string, destination)
OUTCHAR (channel, string, source)
OUTCHARACTER (channel, string, source)
OUTSYMBOL (channel, string, source)
```

string	String from which character is read or written.
destination	Integer variable to be assigned the relative position in the string of the character read.
source	Integer expression indicating relative position in string of character to be output.

Figure 8-1. Formats of INCHAR and OUTCHAR

In both procedures, a mapping is established between the characters of the string (the second parameter), from left to right, and the integers 1, 2, 3, and so forth. Using this correspondence, INCHAR assigns to the integer variable (third parameter) the value corresponding to the next character appearing on the channel. If this character does not appear in the string, the value 0 is assigned. If there is no next character because the end of the input record has been reached, -1 is assigned. Similarly, OUTCHAR writes to the channel the character of the string corresponding to the integer value (third parameter). If the integer value is 0 or larger than the number of characters in the string, an error results. If the value is -1, the current record is terminated.

The standard procedures named INCHARACTER and INSYMBOL are synonymous with INCHAR. OUTCHARACTER and OUTSYMBOL are synonymous with OUTCHAR.

Example:

```
INCHAR (60,≠(≠0123456789≠)≠,N)
```

If the next character on channel 60 is the digit i, N is set to i+1. If the next character is not a digit, N is set to 0.

Example:

```
OUTCHAR (61,≠(≠ABCDEFGHIJKLM
NOPQRSTUVWXYZ≠)≠,J)
```

If J is 1, the character A is output to channel 61; if J is 2, B is output, and so forth.

The procedure OUTSTRING (figure 8-2) transmits a character string from central memory to the channel. The string is transmitted as is, without conversion. The outermost string quotes are not transmitted.

OUTSTRING (channel, string)

string	String to be output.

Figure 8-2. Format of OUTSTRING

INREAL and OUTREAL (figure 8-3) transmit real values according to the standard format for real values.

INREAL (channel, destination)
OUTREAL (channel, source)

destination	Real variable to be assigned the value read from the channel.
source	Real expression to be written to the channel.

Figure 8-3. Formats of INREAL and OUTREAL

A brief description of the standard formats for real numbers, integers, and Boolean values is shown in figure 8-4; a fuller description can be found under Format Strings, below. On input, standard format is actually free form input, as explained under Format Strings.

INREAL reads the next value on the channel and assigns it to the real variable; OUTREAL writes the value of the real variable to the channel. Because they use the same format, a value written by OUTREAL can be read by INREAL.

INARRAY and OUTARRAY (figure 8-5) transmit all the elements of a real array using standard format. The effect is the same as a series of calls to INREAL or OUTREAL. The elements are transmitted in the same order that they are stored (as defined in section 5). The dimensions of the array are determined by the applicable array declaration.

Because standard format is used, INARRAY and OUTARRAY are compatible with INREAL and OUTREAL; a number output by OUTARRAY can be read by INREAL.

Example:

```
OUTREAL (61, 452.9);
```

The number output to channel 61 is in the following form:

```
+4.52900000000000≠+002
```

Real values:

```
Δ+D.DDDDDDDDDDDDD≠+DDDΔ
```

Integer values:

```
ΔΔΔΔΔΔΔ+ZZZZZZZZZZZZZZZDΔ
```

Boolean values:

```
ΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔ≠TRUE≠Δ
```

or

```
ΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔ≠FALSE≠Δ
```

Δ	Blank.
D	Digit. Leading zeros are included.
Z	Digit. Leading zeros are replaced by blanks.
≠	For real values, indicates that the integer that follows is the exponent to the base 10. For Boolean values, appears as part of the special symbols ≠TRUE≠ and ≠FALSE≠.
+	Either a plus sign or a minus sign, as appropriate.

Figure 8-4. Standard Formats for Integer, Real, and Boolean Values

INARRAY (channel, destination)
OUTARRAY (channel, source)

destination	Real array to be assigned the values read from the channel.
source	Real array to be written to the channel.

Figure 8-5. Formats of INARRAY and OUTARRAY

Example:

```
≠REAL≠ ≠ARRAY≠ A[1:30];
OUTARRAY (52, A)
```

The values output by the call to OUTARRAY can be read by a FOR statement:

```
REWIND (52);
≠FOR≠ I := 1 ≠STEP≠ 1 ≠UNTIL≠ 30 ≠DO≠
INREAL (52, A[I])
```

Procedures OUTINTEGER and ININTEGER (figure 8-6) transmit integer values between a channel and a variable. The integer is read and written in standard format (figure 8-4), as a decimal number with a sign.

Example:

```
OUTINTEGER (61, 4*12 - 105)
```

The following is output:

```
-57
```

ININTEGER (channel, destination)
OUTINTEGER (channel, source)

destination	Integer variable to be assigned the value read from the channel.
source	Integer expression to be written to the channel.

Figure 8-6. Format of ININTEGER and OUTINTEGER

ININTARRAY and OUTINTARRAY (figure 8-7) transmit integer arrays in the same way the ININTEGER and OUTINTEGER transmit single values. The values are in standard format. Integers output by OUTINTARRAY can be read by either ININTEGER or ININTARRAY.

ININTARRAY (channel, destination)
OUTINTARRAY (channel, source)

destination	Integer array to be assigned the values read from the channel.
source	Integer array to be written to the channel.

Figure 8-7. Formats of ININTARRAY and OUTINTARRAY

Example:

```
≠INTEGER≠ ≠ARRAY≠ I[1:50];
OUTINTARRAY (33,I)
```

The values output by the call to OUTINTARRAY can be read by a FOR statement:

```
REWIND (33);
≠FOR≠ J := 1 ≠STEP≠ 1 ≠UNTIL≠ 50 ≠DO≠
ININTEGER (33,I[J])
```

Procedures INBOOLEAN and OUTBOOLEAN (figure 8-8) transmit Boolean values. The values are in the standard format for Boolean values (figure 8-4).

INBOOLEAN (channel, destination)
OUTBOOLEAN (channel, source)

destination	Boolean variable to be assigned the value read from the channel.
source	Boolean expression to be written to the channel.

Figure 8-8. Formats of INBOOLEAN and OUTBOOLEAN

INBARRAY and OUTBARRAY (figure 8-9) transmit arrays of Boolean type in the same way that INBOOLEAN and OUTBOOLEAN transmit single values. The values are in standard format.

INBARRAY (channel, destination)
OUTBARRAY (channel, source)

destination	Boolean array to be assigned the values read from the channel.
source	Boolean array to be written to the channel.

Figure 8-9. Formats of INBARRAY and OUTBARRAY

Example:

```
≠BOOLEAN≠ X;
X := ≠FALSE≠;
OUTBOOLEAN (31, X)
```

The following is output to channel 31:

```
≠FALSE≠
```

Example:

```
≠BOOLEAN≠ ≠ARRAY≠ A [1:3];
A[1] := ≠TRUE≠; A[2] := ≠FALSE≠;
A[3] := ≠TRUE≠;
OUTBARRAY (44, A)
```

The following is output to channel 44:

```
≠TRUE≠Δ ... Δ≠FALSE≠Δ ... Δ≠TRUE≠
```

Procedures INPUT and OUTPUT (figure 8-10) provide more flexibility by allowing the user to specify a format string to define the conversion to take place. With OUTPUT, the value of an expression in the list is converted to a string according to the format string and written to the channel. With INPUT, a string is read from the channel, converted according to the format string, and transmitted to a variable. Format strings are described fully below, under the heading Format Strings.

INPUT (channel, format string, $v_1, v_2, \ldots, v_n$)
OUTPUT (channel, format string, $e_1, e_2, \ldots, e_n$)

format string	String to indicate conversion between central memory and device; syntax is defined under Format Strings.
$v_1, \ldots, v_n$	Variables and arrays to which the values read from the channel are to be assigned; n must be from 0 to 252. The v parameters can be omitted.
$e_1, \ldots, e_n$	Expressions or arrays to be written to the channel; n must be from 0 to 252. The e parameters can be omitted.

Figure 8-10. Formats of INPUT and OUTPUT

When INPUT is executed, the list of items to be input can only include variables and arrays. If an array appears, each of its subscripted variables is treated as a separate item. Each list item is matched with a format item. If the number of format items is greater than the number of

list items, the remaining format items are ignored. If the number of list items is greater than the number of format items, standard format is used for the remaining list items. The items are then matched with values appearing on the channel. Each value on the channel is assigned to the next item in the list.

When OUTPUT is executed, a similar process takes place, except that the item list can include any expressions. The list items are matched with format items in the same way, and the value of each list item is converted and output.

HORIZONTAL AND VERTICAL CONTROL

For all coded sequential files, parameters defining horizontal and vertical positioning are defined. Data is grouped into lines, and the current character position of the file within a line is called its horizontal position. Lines can in turn be grouped into pages; the position of a line within a page is called the vertical position. Horizontal and vertical positioning can apply logically to any file, even when physically the file is not divided into lines and pages (for example, mass storage). The most general case of horizontal and vertical control applies to files processed through INLIST and OUTLIST; a subset of the capabilities is available for files processed through other input/output procedures.

For each file, three horizontal and three vertical parameters are established. These parameters take as values numbers indicating character positions (columns) in a horizontal direction, or lines in a vertical direction. The leftmost column in a line is numbered 1, as is the topmost line of a page. The horizontal parameters are as follows:

P The maximum number of characters per line.

R The right margin of a line (default: infinite; parameter applies to INLIST/OUTLIST only).

L The left margin of a line (default: 1; parameter applies to INLIST/OUTLIST only).

The vertical parameters are:

PP The maximum number of lines to a page (default: 0).

RR The bottommost line of a page (default: infinite; parameter applies to INLIST/OUTLIST only).

LL The topmost line of a page (default: 1; parameter applies to INLIST/OUTLIST only).

Values for P and PP are established through calls to the procedures CHANNEL and SYSPARAM or by default. Nondefault values for R, L, RR, and LL can only be set through the procedures INLIST and OUTLIST. For other coded sequential input/output procedures, R, L, RR, and LL are not used.

Coded sequential files are divided into paged and unpaged files, based on their setting of the PP parameter. Paged files are those that are formatted for printing, and have a nonzero value for PP. Carriage control characters are added by ALGOL. Unpaged files are not formatted for printing, and have a value of zero for PP. No carriage control characters are added. The value zero indicates that the file is not to be regarded as being divided into pages; operations like skipping to the top of a new page are ignored. The default value for P is 136 for paged files (carriage control characters added by the system are not counted) and 80 for unpaged files.

The effect of these parameters depends on device type. Although any file can be paged, a page advance does not necessarily result in physical repositioning of the device. It is the user's responsibility to ensure that the values selected for P and PP are appropriate for the device the file resides on, and the disposition of the file at job termination.

L, R, and P are interpreted according to the device type as follows:

- Punch card files

  P indicates the maximum number of card columns to be used; it should be set to the number of columns on the card. L and R are the margins within the card between which characters are to be read or punched. Columns less than L or more than R are bypassed on input and not punched on output.

- Line printer files

  P indicates the maximum number of character positions to be printed; L and R are the margins between which printing takes place. Character positions less than L or greater than R are printed as blanks.

- Mass storage

  P is the maximum record length; L and R are the character positions within the record that delimit data to be interpreted. Character positions less than L or greater than R are bypassed.

- Magnetic tape

  Identical to mass storage. Block size is fixed and has no relation to record size. (See Record Manager Interface, section 12.)

For the simple input/output procedures (all those except INLIST and OUTLIST), the user has limited control over horizonal and vertical arrangement of data. For INPUT and OUTPUT, which allow the user to specify a format string, the format code / can be used to start a new line, and the code ↑ or * can be used to start a new page. B and J allow horizontal spacing. Otherwise, characters are input or output in a continuous stream unless overflow occurs.

For simple input/output procedures, overflow is handled symmetrically for input and output. On output, an item that is too long to fit on the remainder of a line is output at the beginning of the next line instead; that is, items are not broken between lines. The only exception is an item that is too large to fit on one line; it is output beginning in column 1 of the next line and then continuously for as many lines as required. On input, the end of a line is treated as a delimiter; values cannot be broken between lines. The exception is a value too large for one line. Such a value cannot begin on the same line as any other value; it must begin on a line by itself and extend continuously for as many lines as required.

For INLIST and OUTLIST, the handling of overflow is more complicated, because the user can specify left and right margins as well as procedures to be executed when specified conditions occur. (See Layout Procedures, below.) The algorithm used for INLIST and OUTLIST is designed to ensure reasonable results under any circumstances.

INLIST AND OUTLIST

The greatest flexibility in coded sequential input/output is achieved through the procedures INLIST and OUTLIST (figure 8-11). These procedures allow dynamic specification of the major components of the input/output process: the list of items to be input or output, the format string, and the physical layout of the items on the file. These specifications take place through a list procedure and a layout procedure, provided as parameters to each call to INLIST and OUTLIST.

INLIST (channel, layout, list)
OUTLIST (channel, layout, list)

layout	Name of layout procedure to be called each time INLIST or OUTLIST is executed.
list	Name of a list procedure to be used in conjunction with INLIST or OUTLIST.

Figure 8-11. Formats of INLIST and OUTLIST

When a call to INLIST or OUTLIST is executed, the following steps take place:

1. The layout procedure is called to establish the format, page layout, and other characteristics of the physical form of the data.
2. The list procedure is called. Each item presented by the list procedure is input or output as specified by the format string and other layout parameters. When the items presented by the list procedure have been exhausted, the input/output process terminates.

List Procedures

The primary application of list procedures is in conjunction with INLIST and OUTLIST; however, they can also be useful in any context in which the same operation is performed on every member of a set.

A list procedure has exactly one parameter, which is itself a procedure. The list procedure executes a series of calls to this formal parameter procedure, one call for each item in the list to be processed. When the list procedure itself is called, an actual procedure name is substituted for the formal parameter procedure. This procedure is then executed once for each item in the list.

For INLIST and OUTLIST, the user writes a list procedure and specifies its name as the third parameter in a call to INLIST or OUTLIST. When INLIST or OUTLIST is executed, one of the standard procedures INITEM or OUTITEM, respectively, is substituted for the formal parameter procedure, and the list of items presented by the procedure is input or output. (INITEM and OUTITEM cannot be called directly by a user program.)

Figure 8-12 is a program illustrating list procedures in a context other than input/output. In this case, ALIST is the list procedure. The result of the procedure is to replace each element of the array A with its absolute value. The order of execution within the program is as follows:

1. The procedure call ABSLIST(ALIST) is executed (line 17).
2. The actual parameter ALIST is substituted for the formal parameter LIST and execution of ABSLIST begins (line 4).

```
 1.  ≠BEGIN≠
 2.     ≠INTEGER≠ ≠ARRAY≠ A (1:10); ≠INTEGER≠ K;
 3.     ≠PROCEDURE≠ ABSLIST(LIST); ≠PROCEDURE≠ LIST;
 4.     ≠BEGIN≠
 5.        ≠PROCEDURE≠ Z (X); ≠INTEGER≠ X;
 6.           ≠IF≠ X < 0 ≠THEN≠ X := -X;
 7.        LIST(Z);
 8.     ≠END≠;
 9.        ≠PROCEDURE≠ ALIST(ITEM); ≠PROCEDURE≠ ITEM;
10.        ≠BEGIN≠
11.           ≠INTEGER≠ I;
12.           ≠FOR≠ I := 1 ≠STEP≠ 1 ≠UNTIL≠ 10 ≠DO≠
13.              ITEM(A[I]);
14.        ≠END≠;
15.     ≠FOR≠ K := 1 ≠STEP≠ 1 ≠UNTIL≠ 10 ≠DO≠
16.        A[K] := (-2)**K;
17.     ABSLIST(ALIST);
18.     ≠FOR≠ K := 1 ≠STEP≠ 1 ≠UNTIL≠ 10 ≠DO≠
19.        OUTPUT(61,≠(≠≠)≠,A[K]);
20.  ≠END≠
```

Figure 8-12. List Procedure Example

3. The procedure statement LIST(Z) is executed; since ALIST has been substituted for LIST, the statement is actually ALIST(Z).

4. The FOR statement (line 12) is executed, causing the statement part of the FOR statement:

   ITEM(A[I])

   to be executed 10 times. Before execution, the actual parameter Z is substituted for the formal parameter ITEM, so that the call actually executed is Z(A[I]).

5. Each execution of Z involves execution of the conditional statement (line 6) in which an element of A is changed to its absolute value (if necessary).

Figure 8-13 shows a section of a program that uses INLIST and OUTLIST. Notice that the list procedure ALIST is the same as the list procedure in the previous example. In this example, the order of execution is as follows:

1. The call to INLIST is executed:

   a. The layout procedure FORM is executed.

   b. The list procedure ALIST is executed. The procedure named INITEM is substituted for the formal parameter procedure ITEM; the FOR statement results in 10 calls to ITEM (which become calls to INITEM); 10 items are input from channel 60 into the elements of the array A (because the actual arguments to INITEM are the array elements A[I]).

2. The remainder of the program is executed.

3. The call to OUTLIST proceeds exactly the same as the call to INLIST except that the elements of A are transmitted to channel 61.

```
≠BEGIN≠ ≠ARRAY≠ A[1:100];
  ≠PROCEDURE≠ ALIST (ITEM); ≠PROCEDURE≠ ITEM;
  ≠BEGIN≠ ≠INTEGER≠ I;
    ≠FOR≠ I := 1 ≠STEP≠ 1 ≠UNTIL≠ 10  ≠DO≠
      ITEM (A[I]);
  ≠END≠;
≠PROCEDURE≠ FORM;
FORMAT(≠(≠N≠)≠);
INLIST (60,FORM,ALIST);
  .
  .     (remainder of program)
  .
OUTLIST (61,FORM,ALIST)
≠END≠
```

Figure 8-13. INLIST, OUTLIST Example

Layout Procedures

Layout procedures are user-written procedures specified as the second parameter to calls to INLIST and OUTLIST. They cannot be written to control input/output performed through any other input/output procedures.

A layout procedure has no parameters. Its function is to control aspects of the input/output process through calls to a series of standard descriptive procedures. Each of these descriptive procedures establishes nondefault values for parameters the system uses internally during input/output. These parameters cannot be set by the user except through these procedures. If a descriptive procedure is called when INLIST/OUTLIST is not in effect, it acts as a dummy procedure; the call is permitted but no operation takes place. Descriptive procedures can be called from any procedure that is itself called as a result of an INLIST/OUTLIST activation.

Table 8-1 shows the names of the descriptive procedures, their functions, and parameters (with the default value for each parameter).

Example:

Figure 8-14 shows a layout procedure. The format string consists only of an indefinite number of D's (indicating decimal digits); the actual number at execution time is the current value of N. N is defined outside of the layout procedure. Margins are established at columns 4 and 84 if TEST is true, or at columns 8 and 80 otherwise. EXIT is specified as the label to be branched to in the event an end-of-partition is encountered on the file; EXIT also occurs outside the layout procedure.

```
≠PROCEDURE≠ LAYT;
≠BEGIN≠
FORMAT(≠(≠XD≠)≠,N);
≠IF≠ TEST ≠THEN≠ HLIM(4,84)
    ≠ELSE≠ HLIM(8,80);
NODATA(EXIT);
≠END≠
```

Figure 8-14. Example Layout Procedure

Execution of INLIST and OUTLIST

The execution of INLIST and OUTLIST is somewhat complicated, for the following reasons:

The user can change the layout parameters, even after input/output has already begun, through the layout procedure and the overflow procedures.

The system makes every attempt to continue execution in a reasonable way when an unusual condition occurs.

In the discussion that follows, the steps taken when OUTLIST is executed are outlined. The algorithm for INLIST is symmetrical, except as noted.

During execution of INLIST/OUTLIST, the current position of the file is defined by the values of two parameters: CP (horizontal position) and CPP (vertical position). CPP equals the number of lines that have already been output; CP equals the number of characters in the current line that have been output. Thus, the file is positioned so that the next character to be written will be at vertical position CPP + 1 and horizontal position CP + 1. At the beginning of the program, CPP = CP = 0.

1. On each call to OUTLIST, the layout parameters (described in table 8-1) are reset to their default values. Then, the user layout procedure, which might change some of these values, is executed. Because the parameters are reset on each call to OUTLIST, nondefault values are not cumulative, but must be reset each time.

TABLE 8-1. DESCRIPTIVE PROCEDURES

Call		Parameters	Default	Function
FORMAT(string,x_1,...,x_n)	string	Format string.	Standard format	Establish format string for input/output calls.
	xi	Nonnegative integer providing value for the i^{th} X replicator.		
HLIM(l,r)	l	Left margin for input/output lines; character positions < l are not used.	1	Establish left and right margins.
	r	Right margin for input/output lines; character positions > r are not used.	unlimited	
VLIM(ll,rr)	ll	Top margin of a page of input/output; lines numbered < ll are not used.	1	Establish top and bottom margins; ignored for unpaged channels.
	rr	Bottom margin of a page of input/output; lines numbered > rr are not used.	unlimited	
HEND(rnorm,rover,pover)	rnorm	Name of a procedure to be executed for normal line alignment (processing of / format code).	Dummy procedure (no operation)	Establish procedures to be executed at the end of a line.
	rover	Name of a procedure to be executed for overflow of the right margin (R parameter).	Dummy procedure (no operation)	
	pover	Name of a procedure to be executed for overflow of the maximum number of characters in a line (P parameter).	Dummy procedure (no operation)	
VEND(bnorm,bover,ppover)	bnorm	Name of a procedure to be executed for normal page alignment (processing of 1 format code).	Dummy procedure (no operation)	Establish procedures to be executed at the end of a page (ignored for unpaged channels).
	bover	Name of a procedure to be executed for overflow of the bottom margin (RR parameter).	Dummy procedure (no operation)	
	ppover	Name of a procedure to be executed for overflow of the maximum number of lines to a page (PP parameter).	Dummy procedure (no operation)	
TABULATION(n)	n	Tabulation increment; establishes tabulation settings of L, L+n, ...	1	Establish tabulation settings; a J in the format string indicates a skip to the next tabulation setting, with blank fill on output.
NODATA(lab)	lab	Label to be branched to if no data remains on input. Detects end-of-partition (end-of-section on file INPUT).	Program is terminated	Establish label to branch to when end-of-data occurs.

2. The next format item is examined. (The first time through, the first format item in the string is examined.) If the format string has been exhausted, standard format is used until the end of the call to OUTLIST. If the next format item is a title format, it does not require a list item, and it is output without activation of the list procedure. On input, the number of characters required by the title format is skipped. After output of the title format, the next format item is examined, and so on, until a nontitle format is encountered.

3. When a nontitle format has been encountered, the list procedure is activated. The first time, it is activated by calling it with the actual parameter OUTITEM (or INITEM for INLIST). Subsequently, the list procedure is activated by returning from OUTITEM. The list procedure then continues execution from the last call to OUTITEM. Eventually, it either terminates, in which case the execution of OUTLIST is complete, or it calls OUTITEM. The conversion and transfer of a data item, as well as detection and processing of overflow conditions, are all under control of OUTITEM. The following lettered steps give the algorithm for OUTITEM.

   a. The current format item is examined. If the list procedure has called the descriptive procedure FORMAT, thereby changing the format string, the first item in the new string is defined as the current item. Otherwise, the current item is the item that was examined in step 2, which is unchanged. At this point, the format item is removed from the format string and copied elsewhere, so that any further calls to FORMAT have no effect on the current item.

   b. Any alignment marks at the left of the format item are processed by executing line advance for each / and page advance for each ↑. (See description under Page and Line Advance, below.) If the format item consists entirely of alignment marks, then after the page and line advances have been executed, return to step 2. The list procedure is not reactivated until another list item is required.

   c. If necessary, page and line alignment are performed (as described below), to ensure that data is only written between the specified margins.

   d. The size of the data item (the number of characters it occupies on the file) is calculated from the format item. This can be calculated in every case except input under standard format, which is essentially free-form. In this case, data is input until a delimiter is encountered. (See Standard Format, below.) In all other cases, a determination is made whether transfer of the current item would result in overflow (that is, printing beyond the R or P margin). If so, steps f and g are executed; if not, step e is executed.

   e. In the nonoverflow case, the item is converted and written according to the current format item. Then any alignment marks to the right of the format item are processed by executing line advance for each /, page advance for each ↑, and tabulation skipping for each J. Return to step 2.

   f. In the overflow case, if writing the item would require writing characters past the right margin (R parameter), the file is advanced to the beginning of the next line (CP=0), and the user's overflow procedure (specified by the rover parameter of HEND) is executed. Otherwise, if the item would overflow the maximum number of characters per line (the P parameter), the file is advanced to the beginning of the next line, and the overflow procedure indicated by the pover parameter of HEND is executed. In other words, R-overflow takes precedence over P-overflow.

   g. Line and page alignment are then performed, if the overflow procedure has changed the current position or margins of the file. Then, as many characters of the item as possible are written, without exceeding character position R or P. If more characters remain to be transferred, the file is advanced to the beginning of the next line, and either the rover procedure (when $R \leq P$) or the pover procedure (when $R > P$) is called. Step g is repeated as many times as necessary until the whole item is output. After the item is transferred, any alignment marks to the right of the format item are processed. Return to step 2.

Page and Line Advance

Line advance consists of checking page alignment, then advancing the file to the left of the next line (CP=0) and executing the procedure specified by the rnorm parameter of HEND.

Page advance consists of performing line advance (unless the file is positioned at the extreme left margin, that is, CP=0). Then the file is positioned at the top of the next page (CP=0, CPP=0), and the procedure indicated by the bnorm parameter of HEND is executed.

Page and Line Alignment

Line alignment is required when the current position (CP) of the file is to the left of the left margin (L parameter). In this case, blanks are inserted (on output) or characters skipped (on input) until CP = L -1. This is so that the next input/output operation begins at the margin.

Page alignment is necessary when the current line position (CPP) of the file is not within the top and bottom margins. If the file is positioned above the top margin (LL), then a check is made to see if it is positioned at the left of the line (CP=0). If so, lines are skipped until the current line is LL-1, and the file is positioned at the left of that line. If not, the file advances to the left of the next line, procedure rnorm is executed, and page alignment is repeated.

If the file is positioned below the bottom margin (RR) or below the maximum number of lines per page (PP), the file is advanced to the beginning of the next page, and procedure bover or ppover, respectively, is executed. Page alignment is then repeated in case the overflow procedure has changed the margins or file position.

FORMAT STRINGS

A format string is a string with a particular syntax used to control the conversion of data on input or output by the procedures INPUT, OUTPUT, INLIST, and OUTLIST. For INPUT and OUTPUT, the string is specified as a parameter to the procedure call; for INLIST and OUTLIST, it is defined through the descriptive procedure FORMAT. In either case, the string can be either specified as a string in the program, or be input to an array using the H format specification. The array must be an integer array.

A format string consists of a list of format items, separated by commas. The list is bracketed by string quotes. The format items are interpreted from left to right; each controls the input or output of one value.

Format items can be parenthesized and optionally preceded by replicators. This is explained under Replicators. An empty format string (string quotes with no nonblank characters between them) implies standard format.

Example:

```
≠(≠ ≠)≠
```

This is an empty format string.

The format items in turn are composed of individual characters called format codes. In general, each format code specifies the conversion to be performed for one character position on the external device. The exceptions include replicators, the string quotes used in insertion sequences, alignment marks, and certain special codes that specify the conversion of a sequence of character positions. The character codes are summarized in table 8-2.

Replicators

A replicator is a format code that specifies the repetition of the object following it. There are two kinds of replicators:

- An unsigned integer indicates that the object it precedes is to be repeated the specified number of times. For example, 5D is equivalent to DDDDD.
- The format code X specifies that the object it precedes is to be repeated an unspecified number of times. The exact number of repetitions is specified at execution time through the procedure FORMAT. Since this can only be called when INLIST or OUTLIST is active, X replicators can only be used with these procedures. The method for providing values for X replicators is explained in table 8-1, under the FORMAT procedure.

TABLE 8-2. FORMAT CODE SUMMARY

Code	Meaning	Code	Meaning
A	Character, translated into display code	T	Truncation
B	Blank	V	Implied decimal point
C	Comma	X	Variable replicator
D	Digit	Z	Digit with zero suppression
E	Exponent part indicator	+	Write sign unconditionally
F	Boolean value	-	Write sign if negative
H	8-character string	≠	Exponent part indicator
I	Integer	()	Delimiters for replicated sequence of format items
J	Tabulation advance	,	Separates format items
L	Boolean	/	Advance to new line
M	Unconverted, any type	↑	Advance to new page
N	Standard format	*	Advance to new page
P	Boolean value (1 or 0)	≠(≠ ≠)≠	Delimiters for insertion
R	Real	.	Explicit decimal point
S	Single character, output only		

Two types of objects can be repeated through replicators:

- Format codes. The replicator immediately precedes the format code to indicate the number of times the code is to be repeated. The following format codes can be preceded by a replicator:

  B
  Z
  D
  S
  A
  ↑
  /
  *
  J

- Parenthesized format items. The format item, or a list of format items separated by commas, is bracketed by parentheses and preceded by a replicator. At execution time, the item or list of items is repeated the specified number of times. Parentheses can be nested to 32 levels.

A parenthesized item or list of items that is not preceded by a replicator is repeated a maximum of 131 071 times. In practice, this feature would be used when all the remaining items to be transferred are to be converted according to the same code or series of codes. When the list of items is exhausted, transmission stops. Thus, infinite repetition is only meaningful when it is specified at the end of the format string.

Example:

```
≠INTEGER≠ I, J;
≠INTEGER≠ ≠ARRAY≠ K[1:14];
OUTPUT (61, ≠(≠5D,ZZDD,(4D)≠)≠, I, J, K);
```

I is output according to the specification 5D; J is output according to the specification ZZDD; and each subscripted variable in the array K is output according to the repeated specification 4D.

Insertion Sequences and Title Formats

An insertion is a sequence of characters that interrupts the transfer of characters in a number format or other format. On output, when an insertion is encountered in a format item, the characters specified by the insertion are written to the channel. On input, the number of characters in the insertion is skipped in reading from the channel. In this case, the characters composing the insertion are irrelevant; only the number of characters matters.

Insertions can be specified in two ways:

- By the B format code. Each B in a format item specifies that a blank is to be inserted on output, or a character to be skipped on input. The B code can be preceded by a replicator.

- By a string, bracketed by string quotes. On output, the characters in the string are written; on input, the same number of characters is skipped.

Example:

```
OUTPUT (61, ≠(≠2D3B2DB
≠(≠UNITS:≠ )≠BD≠)≠, NUM)
```

If the value of NUM is 13579, it is output as follows:

```
13ΔΔΔ57ΔUNITS:Δ9
```

Anywhere that a single insertion (string or B format code) is permitted, a series of insertions can appear. A series of insertions is known as an insertion sequence. An insertion sequence can appear either before or after any of the following format codes:

Z
C
D
T
+
-
V
≠
S
A
P, 5F, FFFFF, or F (Boolean formats)
↑, /, *, J (alignment marks)
. (period)
E

Insertion sequences cannot appear in the following contexts:

- Between a replicator and the object to be repeated.

  Invalid: 5≠ (≠REPEAT THIS ONE 5 TIMES≠) ≠D

- Before or after any of the nonformat codes I, R, L, M, or H, or the standard format N.

  Invalid: 3BI

  However, the same result can be achieved by using a title format in the same position.

  Valid: 3B, I

- Between any two of the characters making up the Boolean formats 5F and FFFFF.

If a format item contains nothing but an insertion sequence (with or without alignment marks), it is known as a title format.

Figure 8-15 shows an OUTPUT procedure statement that includes a title format, and the line that is written.

ALGOL Statement:

```
OUTPUT (61, ≠(≠ 5B, ≠(≠NAME≠)≠ 6B
    ≠(≠RANK≠)≠ 6B ≠(≠SERIAL
    NUMBER≠)≠≠)≠)
```

Line Written:

```
ΔΔΔΔΔNAMEΔΔΔΔΔΔRANKΔΔΔΔΔΔ
    SERIALΔNUMBER
```

Figure 8-15. Title Format Example

Number Formats

Number formats are used to input or output values of real or integer type. Any number format can be used to transfer values of either type. Figure 8-16 shows the basic layout of a number format. In format 1, the number format must have either an unsigned integer part, a decimal fraction part, or both. A sign part and an exponent part are optional.

In format 2 the number format must have an unsigned integer part, a decimal fraction part, or both. The required sign part follows the number, instead of preceding it as in format 1. The number format components appearing in figure 8-16 are broken down further in figure 8-17.

A sign part consists of a plus sign or a minus sign. If a plus sign is used, the sign of the number (either + or -) is always written on output; if a minus sign is used, the sign is only written if the number is negative. When a sign is written, it appears in a sign position or in the position of a suppressed Z or C, right justified as far as possible.

Examples:

-ZD

Under this format, the number -1 is output as: Δ-1

-ZBD

Under this format, the number -1 is output as: Δ-Δ1

When the sign is not written, a blank is written instead. If no sign part appears, and the number is negative, standard format is substituted.

On input under format 1, the sign can appear in any Z, D, or C position to the left of the first digit of the number, or in the position occupied by the sign part. Under format 2, the sign must appear in the position occupied by the sign part at the right of the number.

Unsigned integer format consists of a Z part, a D part, or a Z part followed by a D part. The Z part consists of one or more occurrences of the letter Z, by itself or preceded by a replicator, optionally interspersed with occurrences of the letter C. The D part is identical in syntax to the Z part, except that the letter D replaces the letter Z. Z indicates zero suppression, and D indicates digit printing without zero suppression. Each Z or D corresponds to a single digit position. Their effect is identical except in the case of leading zeros. A digit in a character position corresponding to a D in the format string is always written. A digit in a character position corresponding to a Z in the format string is written, unless the digit and all the digits to its left are zeros. When a zero is suppressed, a blank is written instead. If no D's appear in the string, a zero value is written as all blanks. Therefore, it is advisable to ensure that at least one D is used.

On input, no distinction is made between Z and D. Leading zeros can appear even if Z's are used, and leading blanks can appear even if the format begins with D.

Each C corresponds to the location of a comma. The comma is written unless the character position to the left is occupied by a zero that was suppressed according to Z format. In this case, a blank is written instead of a comma. A comma following a D is always written.

Decimal fraction format consists of a period or V, followed by a D part, followed by an optional T. The D part is as described for unsigned integer format; no Z's can appear after a period. Either a period or a V is required. If a period is used, it is always written in the corresponding character position. If a V is used, the period is not written, but its location is defined by the location of the V. In this case, no blank is written either. On input, a period is required if a period is used in the format string, but the period must be omitted if V is used.

The format code T indicates truncation. When no T appears, the last digit of the integer or fraction part of a number is rounded to fit the format specified. T specifies that the remainder of the number is to be truncated instead. If N is the number of decimal places specified for a number in a format, the formula for rounding is represented by the ALGOL expression:

$$10\uparrow(-N) * \text{ENTIER}(10\uparrow N * X + .5)$$

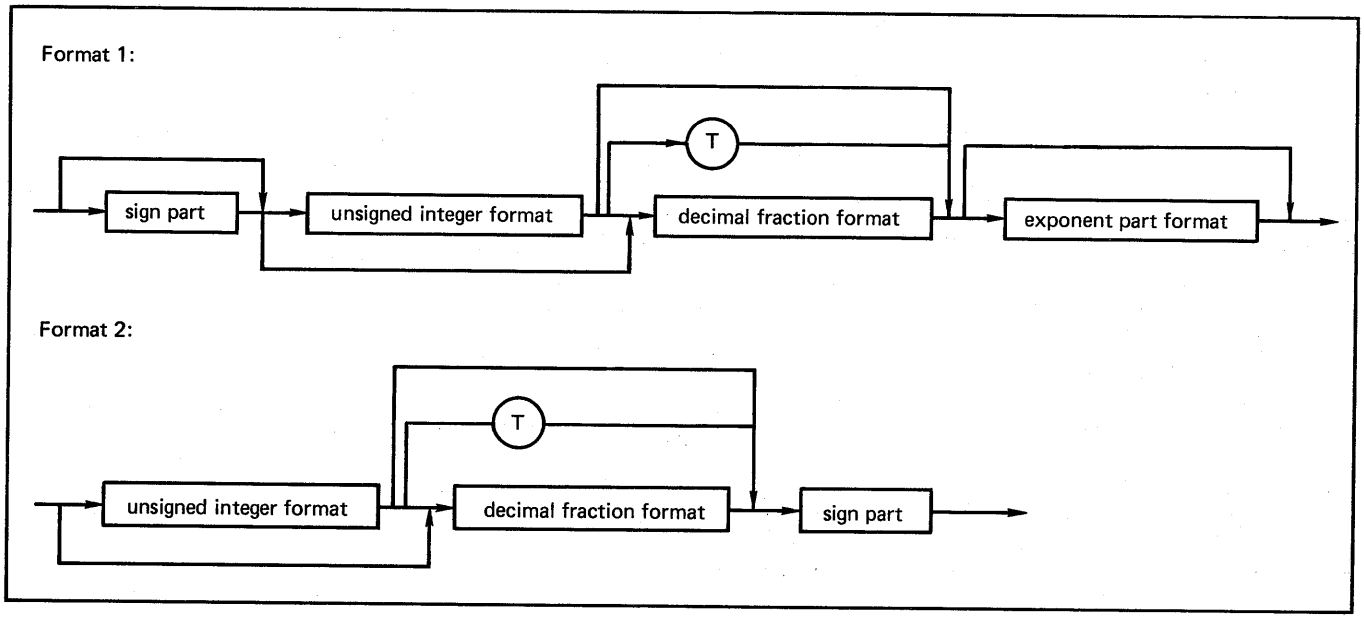


Figure 8-16. Number Format

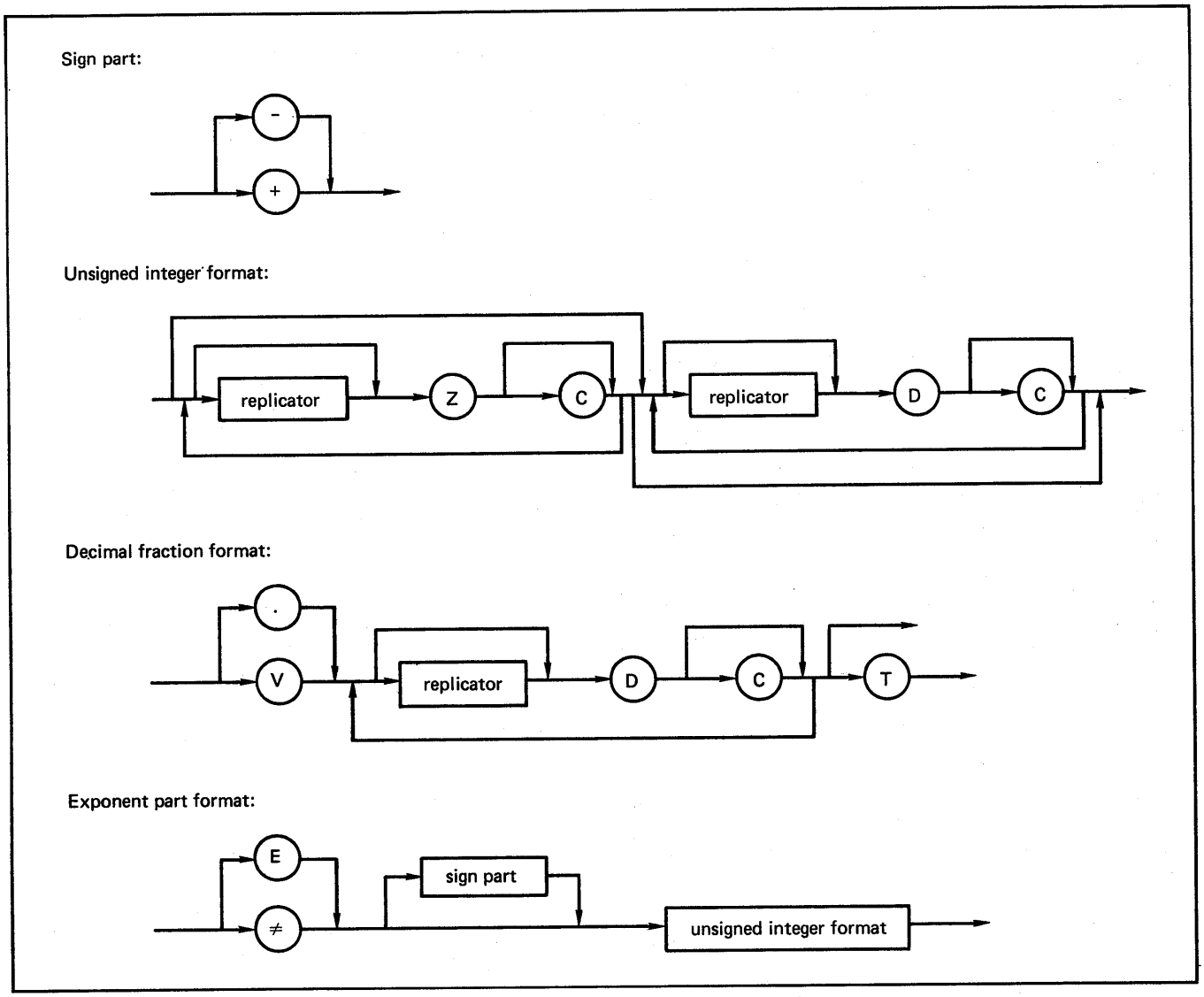


Figure 8-17. Components of Number Format

The formula for truncation is:

10↑(-N) * SIGN(X) *ENTIER(10↑N * ABS(X))

The exponent part format consists of the character ≠ or E, followed by an optional sign part, followed by an unsigned integer format. The character ≠ or E is interpreted to mean that the number following it is the exponent of the preceding number to the base 10. The mathematical interpretation of this notation corresponds to that for a real constant with an exponent (section 2). On output, if no D's are present and the value of the exponent is zero, then the sign and the ≠ or E are omitted. Otherwise, the interpretation of the sign part is the same as when it appears at the left of the number format, except that it applies to the sign of the exponent.

In exponent part format, E is synonymous with ≠, except that on output, an E instead of a ≠ is written. The primary purpose of this code is to allow numbers with exponents to be written in the same format as FORTRAN. On input, either E or ≠ is accepted.

If the exponent part format is omitted, as many digits of the number are output as the format allows. The division between the unsigned integer and the decimal fraction is made where indicated by the period or V. If not enough digit positions are provided for the unsigned integer portion, standard format is substituted. If an unsigned integer format, but no decimal fraction format, is specified, the integer portion of the number is written, and the fractional portion is ignored. If a decimal fraction format is specified, but no unsigned integer format is specified to the left of the decimal point, the fractional part of the number is written unless the absolute value of the number is greater than or equal to 1. In this case, standard format is substituted.

If an exponent part format is included, the decimal point is aligned so that the most significant digit (the leftmost nonzero digit) occupies the character position indicated by the first D or Z in the format. The exponent is adjusted accordingly. If the absolute value of the exponent is too big for the number of digit positions provided in the

exponent part, standard format is substituted. This substitution also takes place when the exponent is negative and no sign part precedes the exponent part format.

Table 8-3 shows how each of several values would be output, using various formats.

Character Formats

The three character format codes are S (string format), A (alpha format), and H (Hollerith format). String format can only be used to output strings. A string format item consists of a series of S's, any of which can be preceded by replicators. Insertion sequences can appear before, after, or within the format item. Each S indicates a character position corresponding to a character in the string. If the number of S's is less than the number of characters in the string, the leftmost characters are output. If the number of S's is greater than the number of characters in the string, the remaining character positions are filled with blanks.

Alpha format specifies transfer (input or output) of a string of characters on the channel to or from an integer variable or array. An alpha format item has the same syntax as a string format item, except that A's are used instead of S's. On input, characters are read from the channel into the integer array or variable specified in the input call. Eight characters are stored in each word, left-justified with zero fill, in the lower 48 bits of the word. The symmetrical process occurs for output. The translation of the characters is according to display code. (See appendix A.) The value of the integer variable or array can be used in all arithmetic operations. On output, if the number of A's in the format item is less than the number of characters originally read, only the leftmost number specified by the format item are output. If the number of A's is greater than the number of characters originally input, the characters are output, followed by characters whose display code value is 0 (colons on most printers). The largest replicator allowed for A is 8.

Hollerith format is identical to 8A format. Exactly eight characters are transferred by each H format item.

The function EQUIV translates a string into display code values in a manner identical to input under A format. The result of the function is an integer corresponding to the actual parameter string.

Boolean Format

When Boolean values are input or output, one of the format items P, F, 5F, or FFFFF, or standard format, must be used. The format item can be preceded and followed by insertion sequences. The correspondence between the Boolean value and the string on the channel is shown in table 8-4.

TABLE 8-4. REPRESENTATIONS OF BOOLEAN VALUES

Format Item	Value	String on Channel
P	≠TRUE≠	1
	≠FALSE≠	0
F	≠TRUE≠	T
	≠FALSE≠	F
5F or FFFFF	≠TRUE≠	TRUE△
	≠FALSE≠	FALSE

Nonformats

Nonformats are single-character format items that specify the transfer of the internal value of a single

TABLE 8-3. NUMBER FORMAT EXAMPLES

Value	Format	Characters Written
1756.1791	+3ZC3ZCZZD.6D	+1,756.179100
	8DV2D-	0000175618
	8DV2DT+	0000175617+
	DD3BDD3BDD ≠-ZZD	17△△△ 56△△△ 18≠△△ -2
	6D ≠3D	Standard format (no sign part is provided for exponent, which is negative)
	3ZV5D	Standard format (integer portion of number is too large for unsigned integer format provided)
-.00019375	-4D.3DT ≠+3Z	-1937.500≠ - △△7
	+DDDD.DDDT≠+ZZZ	-1937.500≠ - △△7
	-3DE-3Z	-194E-△△6
	BB ≠(≠NUMBER:≠)≠ +D.4D ≠(≠EXPONENT:≠)≠≠ -2D	△△NUMBER:-1.9375EXPONENT: ≠-04

variable of a given type. The nonformat codes, and their corresponding types, are as follows:

I Integer

R Real

L Boolean

M Any type

When a value is output according to a nonformat code, the machine representation of the value is written as a string of 20 octal digits. An integer value can be output using the R code, and a real value can be output using the I code; in each case the number is converted before it is output. If the M code is used, no conversion takes place. On input, the 20 octal digits read are transferred to the variable. Either real or integer numbers can be read under either I or R format, and are converted appropriately. Only Boolean values can be read under L format. Insertion sequences and replicators cannot be used with nonformat codes.

Standard Format

Standard format is used whenever any of the following occur:

- The format item N is used.
- The format string is exhausted before transmission of the input/output list is complete.
- The format string is empty (≠(≠ ≠)≠).
- The format provided by the user is not compatible with the value being output. In this case, the blanks at each end of the standard format are replaced by asterisks.

Standard format differs depending on whether input or output is taking place, and also differs among real, integer, and Boolean values, and strings.

On output, a different format is used for each type of value. Table 8-5 shows the format used for each type.

Example:

The following procedure call:

```
OUTPUT (I,≠(≠4(N)≠)≠,1428571,7421.35≠-6,
≠TRUE≠,≠(≠POPOVERS≠)≠)
```

produces the output:

```
ΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔ+142857
Δ+0.07421350000000≠-001Δ
ΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔ ≠TRUE≠
ΔPOPOVERS
```

On input, standard format implies free-format input. For a real or integer variable, any value that obeys the syntax rules for numbers (section 2) is valid. If necessary, the number is converted from real to integer or vice versa. For a Boolean variable, the value must be a valid Boolean constant. A string can be read into an integer array; it must be surrounded by string quotes.

TABLE 8-5. STANDARD FORMAT FOR OUTPUT

Type	Equivalent Format Item
Real	B+D.14D≠ +3DB
Integer	7B+14ZDB
Boolean	17B ≠TRUE≠ B or 16B ≠FALSE≠ B
String	nS, where n is the length of the string

A number read in standard format is delimited at the right by any of the following:

1. A nonblank character other than a period, + or -, digit, E, or ≠ when occurring to the right of a period, + or -, digit, E, or ≠. In other words, insertions can occur to the left of the number, but once meaningful characters of the number have been read, further insertions indicate the end of the number.
2. A sequence of K or more blanks, if occurring to the right of a period, + or -, digit, E, or ≠. K can be set by a call to CHANNEL or SYSPARAM; the default is 2. If fewer than K blanks occur, they are ignored.
3. The end of the first line, if the line contains any meaningful characters of the number, such as period, + or -, digit, E, or ≠. If the end of a line is encountered before any meaningful characters are read, the number is read until a delimiter of type 1 or 2 is encountered; further end-of-lines are ignored. That is, a number can be split across more than one line, but only if the first line contains no meaningful characters.

Alignment Marks

Alignment marks are format items that do not control the input/output of any input/output list items, but rather specify horizontal or vertical repositioning before the next list item is processed. Alignment marks can only appear to the left or right of a format item.

The alignment marks are:

/ New line (explained under Horizontal and Vertical Control, previously in this section)

↑ New page (explained under Horizontal and Vertical Control)

* Same as ↑

J Tabulation

Alignment marks can be preceded by replicators.

J indicates an advance to the next tabulation setting. Tabulation settings are established by a call to the procedure TABULATION. The tabulation settings are L, L+N, L+2N, ... where L is the left margin and N is set by TABULATION. J can only be used under control of INLIST or OUTLIST.

Figure 8-18 shows an example of a format string including alignment marks and the output produced from the call to OUTPUT.

BINARY SEQUENTIAL FILES

A binary sequential file is a sequential file in which the data is not converted. Each record in a binary sequential file is the image of a single array in exactly the same form as it appeared in memory.

PUTARRAY writes an array; GETARRAY reads an array. These procedures are shown in figure 8-19.

For both procedures, the number of words transferred is the same as the length of the array unless file structure parameters specified by the CHANNEL procedure or by a Record Manager FILE control statement do not allow this length to be transferred. For PUTARRAY, the number of words actually written is the smaller of the array size and MRL/10, where MRL is the maximum record length in characters. For GETARRAY, the number of words actually read is the smaller of the array size and RL/10, where RL is the length in characters of the current record. The number of words actually read or written can be obtained from a call to IOLTH (described under Miscellaneous Input/Output Procedures).

On binary sequential files, the procedure EOF detects an end-of-partition. End-of-partition can be written by the procedure ENDFILE.

WORD ADDRESSABLE FILES

A word addressable file is a random access file in which each record is located by the word address of its first word. A word address is an integer equal to the offset of each word in the file from the beginning of the file. Word addresses are consecutive, beginning with 1. If the word address of a record is k, and it is n words long, the record occupies words k through k+n-1 of the file.

The user supplies the word address as a parameter of the input/output call, and the record is written to or read from contiguous words beginning with the word specified. After execution of the input/output call, the channel is positioned at the next word address following the last word of the record. The word address parameter (which is call-by-name) is set by the system to the next available word address; if records are written sequentially, it is unnecessary to reset this parameter.

GETARRAY (channel, destination)
PUTARRAY (channel, source)

destination	Real, integer, or Boolean array into which data is to be read.
source	Real, integer, or Boolean array from which data is to be written.

Figure 8-19. GETARRAY and PUTARRAY Format

Three pairs of standard procedures execute input/output for word addressable files. FETCHLIST and STORELIST (figure 8-20) use a list procedure (the third parameter) to transfer items between central memory and the word addressable file beginning at the word address specified by the second parameter. The operation of the list procedure is exactly the same as for INLIST and OUTLIST. (See under List Procedures, above.)

FETCHLIST (channel, address, list)
STORELIST (channel, address, list)

address	Word address on file where data is to be read or written (call-by-name integer).
list	Name of user's list procedure, executed to provide items to be read or written.

Figure 8-20. FETCHLIST and STORELIST Format

FETCHLIST begins by calling the list procedure, which in turn calls a system procedure that inputs one item at a time. The items requested by the list procedure are read from successive words of the channel and transferred, without conversion, to the list items. Because no conversion takes place, the user should ensure that the types of the list items specified by the list procedure are the same as the types of the data items on the file.

PROGRAM:

```
1.      ≠BEGIN≠
2.        OUTPUT(61,≠(≠≠(≠LINE 1 OF PAGE 1≠)≠↑,
3.          ≠(≠LINE 1 OF PAGE 2≠)≠/,
4.          ≠(≠LINE 2 OF PAGE 2≠)≠≠)≠)
5.      ≠END≠
```

FIRST PAGE OF OUTPUT:

```
LINE 1 OF PAGE 1
```

SECOND PAGE OF OUTPUT:

```
LINE 1 OF PAGE 2
LINE 2 OF PAGE 2
```

Figure 8-18. Alignment Marks Example

STORELIST writes data items, as provided by the list procedure, to the channel, beginning at the specified word address. The items are not converted. Since any word address is permitted, regardless of whether data has already been written at that address, the user should ensure that records do not unintentionally overlap. If a word address is specified that is beyond the current end of the file, the file is extended by allocating space sufficient to accommodate all the intervening words. Thus, if a record is written to a very large word address, a large amount of unused space might be allocated.

FETCHITEM and STOREITEM (figure 8-21) operate the same as FETCHLIST and STORELIST, except that the items to be transferred are specified in the call instead of by a list procedure.

FETCHITEM (channel, address, v1, . . . , vn)
STOREITEM (channel, address, e1, . . . , en)

address	Word address on file where data is to be read or written (call-by-name integer).
v1, . . . , vn	Call-by-name variables and arrays to which the values read from the channel are to be assigned; n must be less than or equal to 252.
e1, . . . , en	Expressions or arrays to be written to the channel; n must be less than or equal to 252.

Figure 8-21. FETCHITEM and STOREITEM Format

The amount of data actually read or written by FETCHITEM, FETCHLIST, STOREITEM, and STORELIST depends on the Record Manager record type. (Nondefault record types are set by the CHANNEL procedure or the FILE control statement.) If the record type is F (fixed length), the combined length of the items specified in the input or output list must be the same as the fixed record length. If the record type is U (unknown) or W (control word), as many records as necessary to fill the specified list items are read or written. (U is the default record type for word addressable files.)

FETCHARRAY and STOREARRAY (figure 8-22) have the same effect as FETCHLIST and STORELIST, except that each call transfers a single array. No list procedure is used. FETCHARRAY and STOREARRAY are faster than FETCHLIST and STORELIST. For both procedures, the number of words transferred is the same as the length of the array unless file structure parameters specified by the CHANNEL procedure or by a Record Manager FILE control statement do not allow this length to be transferred. For STOREARRAY, the number of words actually written is the smaller of the array size and MRL/10, where MRL is the maximum record length in characters. For FETCHARRAY, the number of words actually read is the smaller of the array size and RL/10, where RL is the length in characters of the current record. For record type U, RL is either MRL or the number of characters before the next end-of-section, whichever is less.

For any word addressable procedure, the number of words in the most recently read or written record can be obtained from a call to IOLTH (described under Miscellaneous Input/Output Procedures). This number might not be the same as the number of words of ALGOL data actually read or written.

FETCHARRAY (channel, address, array)
STOREARRAY (channel, address, array)

address	Word address on file where data is to be read or written (call-by-name integer).
array	Name of real, integer, or Boolean array to be read or written.

Figure 8-22. FETCHARRAY and STOREARRAY Format

OTHER INPUT/OUTPUT PROCEDURES

The remainder of the procedures described in this section are auxiliary input/output procedures that perform various functions in conjunction with the procedures already described. With the exception of READECS and WRITEECS, they do not transfer data but perform other functions such as channel specification, file positioning, and error processing. The procedures READECS and WRITEECS are not technically input/output procedures, since extended memory is not a device, but they are included here because they are similar to input/output procedures.

Unless otherwise noted, these procedures apply to all channels, coded sequential, binary sequential, and word addressable. The procedure CHANNEL must be executed for all channels except those for which system defaults are available.

CHANNEL Procedure

The CHANNEL procedure (figure 8-23) links the channel number used in ALGOL input/output calls with a logical file name known to the operating system, and specifies some of the characteristics of the file. Except for channels 60 and 61, which are defined by default, all channels used in a program must be defined by a call to the CHANNEL procedure. The program can check the existence of a particular channel with the CHEXIST procedure. The call to CHANNEL must be executed either before any other reference to the channel has been executed, or after the channel has been explicitly unloaded by the UNLOAD, RETURN, or DETACH procedure; CHEXIST can refer to a nondeclared channel. The channel definition takes effect when the file is opened, either explicitly, by a call to OPEN, or implicitly, when the first input/output operation is executed. The CHANNEL procedure specifies the channel number, the channel type, and other optional parameters. The channel type is coded sequential, binary sequential, or word addressable. The operations allowed on the channel depend on the type.

Once a channel has been defined, it remains available to the program until either the program ends execution or the channel is referenced by the procedure UNLOAD, RETURN, or DETACH (see below).

CHANNEL (channel, type, paramlist)

channel	Channel number; integer expression.
type	One of the following:
	≠(≠ C ≠)≠ Coded sequential channel
	≠(≠ B ≠)≠ Binary sequential channel
	≠(≠ W ≠)≠ Word addressable channel
paramlist	List of parameter/value pairs, separated by commas. Parameters are strings, and values are either strings or integers. See table 8-6 for complete list.

Figure 8-23. CHANNEL Format

A channel can be equated to a previously defined channel by using the format shown in figure 8-24. In this case, all references to either channel number are to the same file.

CHANNEL (ch1, ≠(≠ E ≠)≠ , ch2)

ch1	Number of new channel to be equated to ch2; integer.
ch2	Number of previously defined channel; integer.

Figure 8-24. Channel Equate Format

At execution time, each channel must be linked with a logical file name. This can be accomplished in one of four ways:

- By using the ≠LFN≠ parameter in the CHANNEL procedure to define the file name directly.
- By using the ≠ILF≠ parameter in the CHANNEL procedure. The parameter is provided as an integer representing the display code value of the logical file name, in the same format as 7A format or the EQUIV function.
- By using the LFN parameter in the Record Manager FILE control statement (section 12). The channel must still be declared through the CHANNEL procedure (or by default) but the logical file name becomes that specified in the FILE control statement.
- By default. If no logical file name is specified, the system constructs a name of the form TAPEnn, where nn is the channel number. Only channel numbers 0 to 99 are valid in this case. Channel 1 generates TAPE1, not TAPE01.

The CHEXIST procedure determines whether or not the specified channel currently exists within the executing program. It is a function procedure which returns a boolean result to the point of call.

Example:

```
CHAN:=3;

IF CHEXIST (CHAN)
THEN OUTSTRING (61,≠(≠CHANNEL PRESENT≠)≠)
ELSE OUTSTRING (61,≠(≠CHANNEL MISSING≠)≠);
```

Parameters of the CHANNEL procedure are of two types: ALGOL parameters and Record Manager parameters. Parameters are not cumulative; each call to CHANNEL resets unspecified parameters to the default setting. Therefore, any nondefault values desired must be reset by each call to CHANNEL.

File structure values supplied by the FILE control statement override those supplied by CHANNEL.

Parameters that can be specified in the CHANNEL procedure are shown in table 8-6.

Examples:

```
CHANNEL(92, ≠(≠C≠)≠, ≠(≠P≠)≠, 136,
    ≠(≠PP≠)≠, 60)
```

specifies the following attributes for channel 92 explicitly or by default:

Type	Coded sequential
Logical file name	TAPE92
Horizontal length	136 characters
Vertical length	60 lines
File organization	SQ (sequential)
Record type	Z (zero-byte terminated)
Block type	C (character count)
Fixed record length	137 characters

```
CHANNEL(77,≠(≠W≠)≠,≠(≠PD≠)≠,≠(≠OUTPUT≠)≠,
    ≠(≠ILF≠)≠,EQUIV(≠(≠BASTA≠)≠))
```

specifies the following attributes for channel 77 explicitly or by default:

Logical file name	BASTA
Type	Word addressable
Record type	U (undefined)
Maximum record length	5120 characters
Processing direction	Output

Channels 60 and 61 are defined by default; the definitions are included in the standard prelude. These channels can be redefined provided they are first unloaded by the UNLOAD, RETURN, or DETACH procedures; however, action taken varies with the operating system being used. Consult the operating system reference manual for further detail. The default channel definitions are shown in table 8-7.

FILE OPEN AND CLOSE

Files are implicitly opened when the first input/output statement is executed, and implicitly closed when the program terminates. If desired, however, a file can be explicitly opened by the OPEN procedure and closed by the CLOSE procedure. In addition, the UNLOAD, RETURN, and DETACH procedures can be executed to close the file and dispose of it in various ways. These procedures are applicable to any file type.

The OPEN procedure (figure 8-25) performs the same function as the Record Manager OPENM macro. Any FILE control statement values already specified take effect when the file is opened. The second parameter of OPEN specifies what kinds of operations are allowed for the file: input, output, or both. To change the operations allowed, the file can be closed and reopened. If the file is opened implicitly, the operations allowed depend on the first operation performed. An implicit open takes place when an input/output operation is performed on the file, but the OPEN procedure has not been called.

TABLE 8-6. CHANNEL PROCEDURE PARAMETERS

Parameter†	Type	Value Type	Values	Default	Meaning
PP	A	I	0-inf	0	Maximum number of lines per page (coded sequential). PP=0 means channel unpaged.
P	A	I	0-inf	80 if PP=0 136 if PP > 0	Maximum number of characters per line (coded sequential).
K	A	I	1-inf	2	Number of blanks that serve as a delimiter in standard format (coded sequential).
RT††	C	S		Z for coded sequential under NOS or NOS/BE W for coded sequential under SCOPE 2 S (binary sequential) U (word addressable)	Record type.
			Z		Zero-byte terminated (coded sequential only).
			S		System-logical-record (coded sequential or binary sequential).
			W		Control word.
			F		Fixed length.
			U		Undefined.
BT†††	C	S		C (When RT≠W)††††	Block type (coded or binary sequential only).
				I (When RT=W)††††	
			C		Character count.
			I		Internal control word (only allowed for RT=W).
FL	C	I	1-10(217-1)	Paged file: 137 if P not provided, otherwise P+1 Unpaged file: 80	Maximum record length for Z type records; fixed record length for F type records.
OF	C	S		N	File position on open (sequential only).
			N		No rewind.
			R		Rewind.
			E		Extend.
CF	C	S		N	File position after close (sequential only).
			N		No rewind.
			R		Rewind.
			U		Unload.
			RET		Return.
			DET		Detach.

TABLE 8-6. CHANNEL PROCEDURE PARAMETERS (Contd)

Parameter	Type	Value Type	Values	Default	Meaning
PD	C	S		IO	Processing direction.
			IO		Allow input and output.
			INPUT		Allow input only.
			OUTPUT		Allow output only.
LT	C	S		UL	Label type.
			UL		Unlabeled.
			S		ANSI standard.
			NS		Nonstandard.
			ANY		Any.
MRL	C	I	1-10(2^{17}-1)	5120	Maximum record length for RT=W,S,U.
BFS	C	I	1-(2^{17}-1)	Set by Record Manager	Buffer size in words.
LFN	C	S	Logical file name	see text	Logical file name.
ILF	A	S	Integer	see text	Integer equivalent of logical file name (A format)

Type: A=ALGOL, C=Record Manager Value Type: I=integer, S=string

†Consult the Record Manager reference manual for restrictions on parameter combinations for CHANNEL.

††Record type U is not permitted for a coded sequential channel. The RT parameter must not be specified for a word-addressable file.

†††The BT parameter must not be specified for a word-addressable file.

††††For SCOPE 2 mass storage files, the default is unblocked. For SCOPE 2 tape files, the default is I for RT=W, C for RT=S, and K for all other record types.

TABLE 8-7. DEFAULT CHANNEL DEFINITIONS

Channel Number	Logical File Name	Type	Parameters
60	INPUT	C	P = 80
			FL = 138
61	OUTPUT	C	P = 136
			PP = 60

OPEN (channel, direction)

direction — String specifying whether input, output, or both can take place on the channel. One of the following:

≠(≠ INPUT ≠)≠	Input only
≠(≠ OUTPUT ≠)≠	Output only
≠(≠ IO ≠)≠	Input or output

Figure 8-25. Format of OPEN

OPEN can be called when a file has been closed by a call to CLOSE, or when the file has already been defined by CHANNEL and no input/output operations have taken place yet.

The CLOSE procedure (figure 8-26) performs the same function as the Record Manager CLOSEM macro. When a file is closed, no input/output operations can be performed until another OPEN is executed (either an explicit or an implicit open). If the last operation on the file was a write, any information remaining in the buffer is written to the file; an end-of-partition is written, and the file is backspaced past the end-of-partition.

```
CLOSE (channel)
```

Figure 8-26. Format of CLOSE

The UNLOAD procedure (figure 8-27) closes the file and performs an operating system unload function. The file becomes undefined to the ALGOL program; its channel number can be reused. Any equipment allocated for the file is freed.

```
UNLOAD (channel)
```

Figure 8-27. Format of UNLOAD

The RETURN procedure (figure 8-28) performs an operating system return function. It closes the file, frees the device the file is on, and removes its definition. The file becomes undefined to the ALGOL program. RETURN differs from UNLOAD in that, if the file is on tape, RETURN decreases by one the maximum number of tape units allowed for the job, while UNLOAD does not.

```
RETURN (channel)
```

Figure 8-28. Format of RETURN

The DETACH procedure (figure 8-29) closes a file and removes its definition, but does not free the device on which the file resides.

```
DETACH (channel)
```

Figure 8-29. Format of DETACH

The CONNECT procedure (figure 8-30) associates a channel with the terminal. Data can then be entered or displayed as the program executes. If a channel exists as a local file but is not connected when CONNECT is called, the buffer of the file is flushed before the channel is connected to the terminal. Under NOS, the file is returned. Calls to CONNECT are ignored in batch mode.

```
CONNECT (channel)
```

Figure 8-30. Format of CONNECT

The DISCONT procedure (figure 8-31) disassociates a channel from the terminal so that no more information can be exchanged between the program and the terminal. On NOS/BE, after being disconnected, the channel remains a local file. Also, if the channel existed before being connected, the file name is reassociated with the information contained on the device where the channel resided before connection. Data written to a connected channel is not contained in the file after disconnection.

```
DISCONT (channel)
```

Figure 8-31. Format of DISCONT

On NOS, the DISCONT procedure causes the connected channel to be returned. The disconnected file name is not reassociated with the preexisting information.

FILE POSITIONING PROCEDURES

File positioning procedures include those that write end-of-partition boundaries and those that skip them forwards or backwards, as well as procedures that rewind files. REWIND is applicable to all files; all other file positioning procedures are only applicable to sequential files.

ENDFILE (figure 8-32) writes an end-of-partition boundary on a file. On an output file, any information in the buffer is written to the file before the end-of-partition is written. On an input file, ENDFILE is treated as a dummy procedure.

```
ENDFILE (channel)
```

Figure 8-32. Format of ENDFILE

SKIPF and SKIPB (figure 8-33) skip end-of-partition boundaries forwards and backwards, respectively. Each call skips one boundary. After the skip, the file is positioned on the other side of the boundary.

```
SKIPF (channel)
SKIPB (channel)
```

Figure 8-33. Formats of SKIPF, SKIPB

BACKSPACE (figure 8-34) positions the file backwards one record. On a coded sequential channel, a record is a single card image or print line or the corresponding amount of information. On a binary sequential file, a record is the amount of information written by a single call to PUTARRAY.

```
BACKSPACE (channel)
```

Figure 8-34. Format of BACKSPACE

An end-of-partition counts as a record on any sequential file. An end-of-section counts as a record on any sequential file with record type other than S (the default record type for binary sequential files). If one of these boundaries is encountered, the exit established by a call to EOF, if any, is taken. If that label is no longer accessible, or if none was established, the program terminates abnormally.

REWIND (figure 8-35) positions a file at beginning-of-information. It is applicable to all file types.

```
REWIND (channel)
```

Figure 8-35. Format of REWIND

EXTENDED MEMORY PROCEDURES

The procedures READECS and WRITEECS (figure 8-36) transfer data between central memory and extended memory. On the CDC 7600, CYBER 70 Model 76, and CYBER 170 Model 176 computers, the ECS procedures transfer data between small central memory (SCM or SSM) and large central memory (LCM or LCME). Each call reads or writes a single array of any type.

```
READECS (address, array, label)
WRITEECS (address, array, label)

address    Address in ECS where reading or writing
           is to begin.

array      Name of array to be read or written.

label      Label to be branched to in case of
           irrecoverable parity error.
```

Figure 8-36. Formats of READECS, WRITECS

If the extended memory field length currently allocated would be exceeded by a call to WRITEECS, a request is made to increase the field length, up to the maximum allowed for the job. If insufficient extended memory field length is available, or if the allocated field length is exceeded on a call to READECS, the program terminates abnormally.

The procedure MOVE (section 7) should not be used to move virtual arrays allocated in extended memory by the same program as the procedures READECS and WRITEECS.

MISCELLANEOUS INPUT/OUTPUT PROCEDURES

The remaining input/output procedures are those that check or alter the status of ALGOL system control variables (SYSPARAM), establish labels to branch to in case of specified conditions (EOF, BADDATA, PARITY, CHANERROR), or return the amount of data read or written by a previous call (IOLTH).

SYSPARAM (figure 8-37) either returns or changes the value of a system input/output parameter, depending on the value of the function parameter. If function is an odd number, the value of the system parameter is returned in the variable specified by quantity. If function is an even number, the parameter is set to the value given by quantity. The parameters that can be set or checked are shown in table 8-8.

```
SYSPARAM (channel, function, quantity)

function   Integer value from 1 to 10. Specifies parameter
           and whether it is to be set or checked. See
           table 8-8.

quantity   Integer value. If function is odd, an integer
           variable that receives value of parameter. If
           function is even, an integer expression that
           provides value for parameter.
```

Figure 8-37. SYSPARAM Format

TABLE 8-8. SYSPARAM FUNCTIONS

Function	Action	Parameter	Meaning
1	R	CP	Current position within line.
2	S	CP	
3	R	CPP	Current position of line within page (returns 0 for unpaged channels).
4	S	CPP	Dummy procedure for unpaged channels.
5	R	P	Maximum number of characters per line.
6	S	P	
7	R	PP	Maximum number of lines per page (returns 0 for unpaged channels).
8	S	PP	Dummy procedure for unpaged channels.
9	R	K	Number of spaces that serve as a delimiter in standard format input/output.
10	S	K	

R = Value is returned.

S = Value is set.

If the value of function is 2 or 4, the parameters that are changed are CP (current character position within line) or CPP (current line position on page). Changing these positions usually involves manipulation of the file contents. If the file is an output file (last operation was a write), blanks are written until the file is properly positioned; if the file is an input file (last operation was a read), characters are skipped. If no operations have taken place, or the last operation was a positioning operation such as REWIND, no action takes place.

For function=2, if the new value is greater than the old value, the file is positioned further to the right on the same line. If the new value is less than the old value, the file is skipped to a new line, and then spaced over the appropriate number of character positions.

For function=4, if the new value is greater than the old value, the file is positioned to the beginning of a new line, and then spaced down the appropriate number of lines. If the new value is less than the old value, the file is positioned to the beginning of the line at the top of a new page, and then spaced down the appropriate number of lines. In no case is a file ever moved backwards as a result of a call to SYSPARAM.

When function=6 or 8, the quantities P and PP cannot be changed unless a new value is allowed for the device on which the file resides. Otherwise, the procedure acts as a dummy procedure.

A paged file can be made into an unpaged file at any time by resetting the quantity PP to 0 (through function = 8). When this happens, the value for P is automatically increased by 1, to allow for the carriage control character. Similarly, an unpaged file can be made into a paged file by changing PP to a nonzero value. In this case, the value for P is automatically decreased by 1.

Example:

```
CHANNEL (92,≠(≠C≠)≠,≠(≠P≠)≠,136,≠(≠PP≠)≠,60)

SYSPARAM (92,8,0)
```

Changes channel 92 to an unpaged file, and resets P to 137.

```
SYSPARAM (92,8,40)
```

Changes channel 92 back to a paged file, and resets P to 136.

EOF (figure 8-38) establishes a label to be branched to when a boundary is encountered during a read operation on the specified channel. The boundary causing the branch is end-of-section on a coded sequential file and end-of-partition on a binary sequential file. During execution of BACKSPACE, the label is also branched to if an end-of-partition or beginning-of-information is encountered, or if end-of-section is encountered on any file except those with record type S (the default record type for binary sequential files).

EOF (channel, label)

label	Label to branch to if a boundary condition is encountered during a read operation on the specified channel.

Figure 8-38. EOF Format

During execution of INLIST, the label established by a call to NODATA takes precedence over the label established by EOF. If the label is no longer accessible when EOF is executed, or if no label is established, and a boundary is encountered, the program terminates abnormally.

BADDATA (figure 8-39) establishes a label to branch to when, during a read operation on the specified channel, the data read does not match the corresponding format (string or Boolean data with number format, and so forth). If the label is no longer accessible, or no label has been established, the program terminates abnormally when the error occurs.

BADDATA (channel, label)

label	Label to be branched to in event of a conflict between input data and format string on the specified channel.

Figure 8-39. BADDATA Format

PARITY (figure 8-40) establishes a label to branch to when an irrecoverable parity error occurs on the specified channel. If the label is no longer accessible, or no label has been established, the program terminates abnormally.

PARITY (channel, label)

label	Label to be branched to in event of an irrecoverable parity error on the specified channel.

Figure 8-40. PARITY Format

CHANERROR (figure 8-41) establishes a label to branch to when a specified type of error occurs on the specified channel. The errors, and their codes, are shown in table 8-9.

CHANERROR (channel, key, label)

key	Integer value from 0 to 6, indicating type of error to cause branch. See table 8-9.
label	Label to be branched to if error of specified type occurs on the specified channel.

Figure 8-41. CHANERROR Format

TABLE 8-9. CHANERROR KEYS

Key	Type of Error
0	Any of the errors 1 through 6
1	Irrecoverable parity error
2	End-of-partition
3	Bad data (see BADDATA procedure)
4	Formatting errors
5	Illegal operation for file type or input/output direction
6	Errors in input/output procedures other than reads and writes

IOLTH (figure 8-42) returns the length in words of the last record read or written on the specified channel. It only applies to binary sequential and word addressable channels; when called for a coded sequential channel, it returns a value of 0.

IOLTH (channel)

IOLTH	Function designator returning length in characters of last record read or written on specified channel.

Figure 8-42. IOLTH Format

Three types of program groupings are compiled by ALGOL:

- Programs. A program is a block or compound statement; see section 4.
- Separately compiled procedures. These are procedures whose text is compiled separately from the program from which they are called. Thus, several programs can use the same procedure without recompilation.
- Circumludes. These are separately compiled program segments containing declarations and statements, which logically surround a program.

A single call to the ALGOL compiler compiles either programs and separately compiled procedures only, or circumludes only. Programs and separately compiled procedures can be mixed, but a circumlude must be compiled by itself. Any number of units can be compiled by one call; the units are separated by the symbol ≠EOP≠. Any characters following ≠EOP≠ on the same line are ignored. Thus, the section of the job deck to be processed by one call to the compiler follows one of two patterns:

One program or any number of separately compiled procedures

≠EOP≠

One program or any number of separately compiled procedures
.
.
.

or

One circumlude

≠EOP≠

One circumlude
.
.
.

Separately compiled procedures can follow each other without an intervening ≠EOP≠; they are separated by semicolons. Procedures compiled together (not separated by ≠EOP≠ symbols) are all loaded whenever one of them is. The ≠EOP≠ symbol is required between a program or circumlude and any other unit.

Any of these three kinds of units can be preceded by an identification sequence. An identification sequence is a sequence of characters preceding the first ≠BEGIN≠ symbol of a program or circumlude, or the ≠CODE≠ or ≠ALGOL≠ symbol of a separately compiled procedure. The sequence cannot contain any of the symbols ≠BEGIN≠,≠CODE≠, ≠ALGOL≠, or ≠EOP≠. The first identifier encountered in the sequence is truncated to seven characters and used as identification for the unit. A circumlude name is truncated to six characters.

Identification is required for a circumlude. For a program, if no identification appears, the name used is A60PROG. For a separately compiled procedure, if no identification appears, the name used is the external identifier associated with the first code procedure in the source text.

SEPARATELY COMPILED PROCEDURES

Separately compiled procedures allow the user to declare and use a procedure in one or more programs without repeating the text in each program. The text of the procedure is linked with its declaration in the calling program unit by a code identifier which is present in this procedure declaration and also in the separate compilation. The procedure declaration is found in the calling program unit; however, the text of the procedure is compiled separately.

PROCEDURE DECLARATION

The procedure declaration for a separately compiled procedure can be contained in a program, a circumlude, or another separately compiled procedure. The declaration has the same syntax as the procedure declaration described in section 6, except that the statement forming the procedure body is replaced with a code part. The syntax of the code part is shown in figure 9-1.

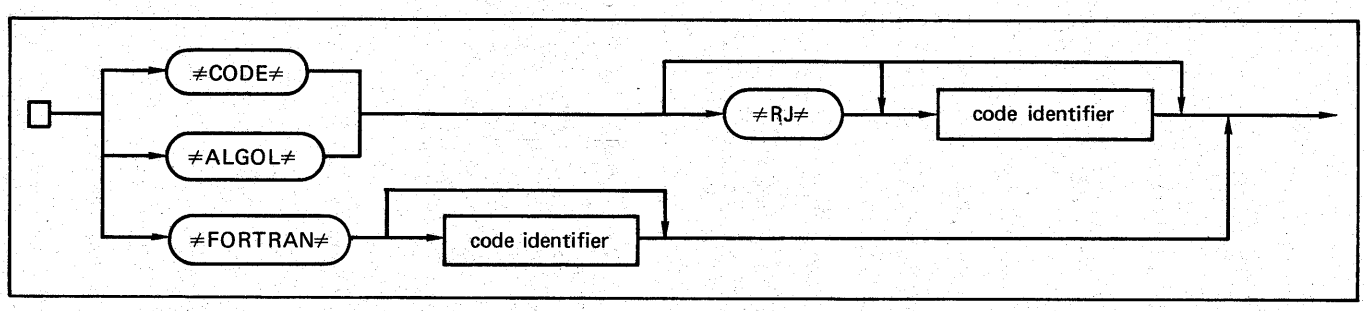


Figure 9-1. Syntax of Code Part

If the code part begins with the symbol ≠ALGOL≠ or ≠CODE≠, the compiler assumes that the separate procedure is written in either ALGOL or COMPASS, and that if written in COMPASS, it uses the ALGOL calling sequence. To facilitate the writing of COMPASS procedures for ALGOL programs, and ensure that the linkage is correct, a set of interface macros is provided; they are described in section 12.

If the code part begins with the symbol ≠FORTRAN≠, the compiler assumes that the procedure is written in FORTRAN Extended version 4, SYMPL, or COMPASS. In this case, it generates a calling sequence compatible with FORTRAN (SYMPL has the same calling sequence). If the procedure is written in COMPASS, the interface macros cannot be used. The following restrictions apply to FORTRAN procedures and to any other FORTRAN procedures they call:

- If a formal parameter is defined in the FORTRAN subprogram (that is, it is given a value), the corresponding actual parameter must be a simple or subscripted variable.
- A formal parameter that is a procedure can only correspond to an actual parameter that is a separately compiled procedure declared with the ≠FORTRAN≠ symbol.
- An actual parameter cannot be a switch or designational expression.
- The FORTRAN procedure must not perform any input/output.
- The FORTRAN procedure must not use blank common.
- An array should not be specified in the value part of the procedure declaration in the ALGOL program, since FORTRAN might change the values of the array. However, a call-by-value array in an ALGOL procedure that calls a FORTRAN subprogram can be passed as an actual parameter to the FORTRAN subprogram.
- If a FORTRAN subprogram is called from an ALGOL procedure, a formal string parameter in the ALGOL procedure cannot be an actual parameter to the FORTRAN subprogram.
- In FORTRAN, array elements are stored with the leftmost subscript varying most rapidly. In ALGOL, the rightmost subscript varies the most rapidly. Array elements are not reordered by ALGOL, so that a subscript of a multidimensional array element in FORTRAN does not in general reference the same array element as the same subscript in ALGOL.

The code identifier that follows the ≠ALGOL≠, ≠CODE≠, or ≠FORTRAN≠ symbol is either an external identifier or a number of one to five digits. If it is a number, ALGOL makes it into an external identifier by prefixing it with the letters CD and enough leading zeros to make five digits. Thus, 123 becomes CD00123. In either case, the external identifier links the procedure declaration in the calling program with the identification preceding the separately compiled procedure. The external identifier must match the identification found in the separately compiled procedure.

The code identifier is optionally preceded by the symbol ≠RJ≠ (unless the procedure is declared with the ≠FORTRAN≠ symbol). This feature enables the separately compiled procedure to use a faster calling sequence. The procedure must be written in COMPASS to take advantage of this calling sequence. The calling sequence is defined in section 12.

For a procedure specified with the ≠RJ≠ symbol, the linkage between the calling program unit and the procedure depends on how it is called. If the procedure is called directly (that is, its name is used in a procedure statement or as a function designator) the fast calling sequence is used. If the procedure is called indirectly, (that is, a formal parameter procedure is called and the name of the ≠RJ≠ procedure is the corresponding actual parameter), then the compiler cannot determine at compile time which calling sequence the procedure uses. Therefore, when the indirect call is executed, control is transferred to a special piece of code that translates the normal ALGOL calling sequence into the fast calling sequence.

The ≠RJ≠ feature is only applicable to procedures with at most one parameter. The parameter must be called by value, and cannot be an array.

PROCEDURE TEXT

Separately compiled procedures are written in FORTRAN Extended, COMPASS, SYMPL, or ALGOL. If they are not written in ALGOL they must be compiled or assembled by the appropriate compiler or assembler with a name that matches that derived from the procedure declaration in the ALGOL program. They must be available to the loader; when the ALGOL program is loaded, the procedures it calls are loaded as well.

If the procedure is written in ALGOL, it can be compiled by the ALGOL compiler. Separately compiled procedures can be grouped together for compilation; a separately compiled procedure must be followed by a semicolon. A grouping of procedures must be separated from any programs in the same compilation by the symbol ≠EOP≠.

The syntax of a separately compiled procedure written in ALGOL is the same as that for a procedure declaration (section 6) except that the symbol ≠PROCEDURE≠ (or the type, if any) must be preceded by the symbol ≠ALGOL≠ or ≠CODE≠ and an optional external identifier or number. The external identifier or number is linked with the procedure declaration in the calling program as explained under Procedure Declaration, above. The syntax of a separately compiled procedure is shown in figure 9-2. Logically, a separately compiled procedure is considered to be part of the circumlude under which it is compiled, or under the standard circumlude if no user-defined circumludes are used. That is, it can only make use of identifiers declared in the procedure itself and in the circumlude.

A main program written in another language cannot call an ALGOL procedure directly. Interlanguage communication is accomplished using a COMPASS module. See section 12. ALGOL procedures can be called only if the main program is written in ALGOL.

CIRCUMLUDES

A circumlude is a set of declarations and statements that serves as an outer block to one or more programs, but is compiled separately from the programs. It consists of two parts:

- The prelude, which contains declarations that are available to the entire program, and code that is to be executed before the program begins.
- The postlude, which contains code that is to be executed after the program ends.

In order to use a circumlude, the following steps must take place:

1. The circumlude is compiled. The circumlude is then written to a file.
2. The file containing the circumlude must be made into a library by the user.
3. When the actual program is compiled, the library must be available. The declarations in the prelude are then available to the program.
4. When the program is executed, the library must also be available. Execution begins with the first executable code in the prelude, continues with the program itself, and ends with the executable code in the postlude.

To compile a circumlude, the N option must be present on the ALGOL5 control statement (section 11). When this option is present, only circumludes, not programs or separately compiled procedures, can be compiled. Two or more circumludes can be compiled by the same control statement, but they must be separated by ≠EOP≠ symbols in the source input. Each circumlude can be either simple or nested.

The syntax of a circumlude is shown in figure 9-3. A simple circumlude consists of any number of labels, each followed by a colon, followed by a prelude, followed by the symbol ≠PROG≠ and a semicolon, followed by a postlude. A prelude consists of the symbol ≠BEGIN≠, followed by any number of declarations, each followed by a semicolon, followed by any number of statements, each followed by a semicolon. A postlude consists of any number of statements, separated by semicolons, followed by the ≠END≠ symbol.

A circumlude is logically equivalent to an outer block, surrounding one or more inner blocks. The symbol ≠PROG≠ occupies the place of the innermost block. At execution time, ≠PROG≠ is logically replaced by the actual program.

Circumludes can be nested either statically or dynamically. Statically, more than one circumlude is compiled at the same time. The circumludes are nested as follows:

```
Prelude of outermost circumlude
.
.
.
Prelude of innermost circumlude
≠PROG≠
Postlude of innermost circumlude
.
.
.
Postlude of outermost circumlude
≠EOP≠
```

Dynamically, circumludes are nested by compiling the outermost circumlude first, and then using that circumlude to compile the next outermost circumlude, and so forth. This is explained below.

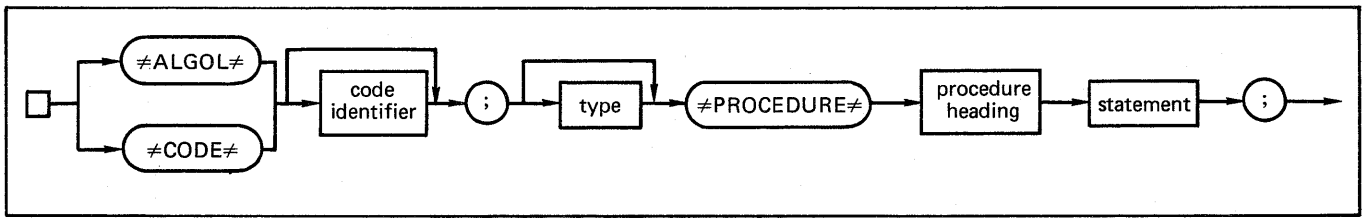


Figure 9-2. Syntax of Separately Compiled Procedure

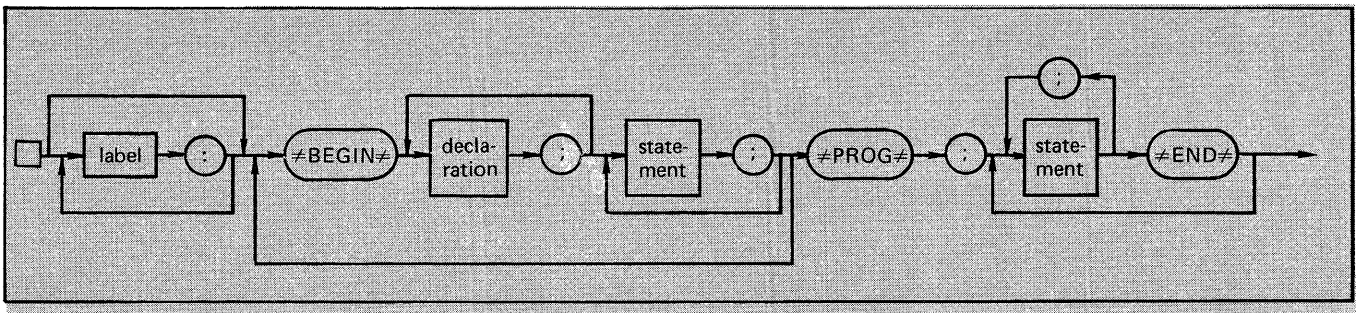


Figure 9-3. Syntax of Circumlude

The output from a circumlude compilation to the file denoted by the B option on the ALGOL5 control statement is two modules, separated by an end-of-section. The first module is an absolute overlay, containing tables derived from the declarations in the prelude. The second module, which is relocatable, contains the executable code from the prelude and the postlude. To compile a program using the circumlude, the user must use the library-creating utility for his operating system.

Under NOS/BE, EDITLIB is used. (See the NOS/BE reference manual.) Both modules of the binary file are placed on the same library.

Under SCOPE 2, the LIBEDT utility is used. (See the SCOPE 2 reference manual.) Both modules of the binary file are placed in the same library.

Under NOS, the absolute module (the first module on the binary file) must be written to a nonlibrary file that has the same name as the prelude. The relocatable module (the second module on the binary file) must be placed in a library using the LIBGEN utility. (See the NOS reference manual.)

To compile a program using the circumlude, the library (or, for NOS, the library and the nonlibrary file) containing the binaries of the circumlude compilation must be local files. No LIBRARY control statement is necessary. The N parameter on the ALGOL5 control statement is omitted, and the S parameter is required in the form:

S = lib-prel

where lib is the library name, and prel is the prelude name. At execution time, the library containing the relocatable module must be a local file. No LIBRARY control statement or control statement option is required.

A circumlude can be used to compile another circumlude, instead of a program. In this case, the circumludes are logically nested, and the effect is the same as if the nested circumludes were compiled together. To do this, the first circumlude (which is the outer circumlude) is compiled first, using the N option, and then written to a library as described above. Then the second, or inner, circumlude is compiled, again using the N option, and naming the library containing the first circumlude in the S option. When the program using the nested circumludes is compiled, the N option is omitted, and the S option names the library containing the innermost circumlude. The process can be repeated, to nest circumludes several levels deep; in each case the S option for a circumlude compilation names the library of the next outer circumlude, and the program compilation uses the S option to name the library containing the innermost circumlude.

A default circumlude, called the standard circumlude, is provided by the compiler and requires no action by the user. It is always available during compilation, and contains declarations of the standard procedures (section 7).

An example of a circumlude is given in section 15.

The user can add directives to a program to enable compilation time options. These directives are included as comment sequences in the following format:

≠COMMENT≠ directive ;

By including the appropriate option with the CD parameter on the ALGOL5 control statement (section 11), the user instructs the compiler to take the action indicated by the directive. If the CD parameter is omitted, or if it does not specify the appropriate option for the directive in question, the comment containing the directive is listed, but the directive is ignored. If the CD parameter selects the appropriate option, the specified action is taken.

Comment directives are especially useful in the debugging stages of a program; the program can be compiled with the directives selected in the early stages of debugging, and then compiled with the directives disabled for the compilation that actually produces executable code.

The comment directives, as well as the options of the DB parameter that select them, are shown in table 10-1.

The paired directives: ≠LIST≠/≠NOLIST≠, ≠OBJLIST≠/≠OBJNOLIST≠, and ≠CHECKON≠/≠CHECKOFF≠, have a scope that is determined statically at compile time. The default settings are ≠LIST≠, ≠OBJNOLIST≠, and ≠CHECKOFF≠. When a nondefault directive is encountered, it remains in effect as the text of the program is read until the default directive is encountered. Thus the period during which a nondefault directive remains in effect is not related to the dynamic flow of control of a program.

≠LIST≠, ≠NOLIST≠, ≠OBJLIST≠, and ≠OBJNOLIST≠ are in effect at compile time; ≠CHECKON≠ and ≠CHECKOFF≠ are in effect at execution time.

An additional comment directive, ≠VIRTUAL≠, specifies that the arrays named in the immediately following array declaration or specification are to be virtual arrays. It is always in effect, regardless of any control statement options. The V option on the ALGOL5 control statement determines the residence of virtual arrays, not whether the comment directive is to be honored. Virtual arrays are discussed in section 2, under Arrays.

TABLE 10-1. COMMENT DIRECTIVES

Directive	Selected By	Effect
≠INCLUDE≠	CD=I	All the characters following ≠INCLUDE≠ and preceding the first semicolon are included in the source program. The effect is the same as if the characters ≠COMMENT≠ ≠INCLUDE≠ were omitted.
≠LIST≠	CD=L	Begin listing source code with this comment and continue listing until a ≠NOLIST≠ directive is encountered.
≠NOLIST≠	CD=L	Stop listing source code, and do not list until a ≠LIST≠ directive is encountered.
≠EJECT≠	CD=L	Eject a page in the source listing.
≠OBJLIST≠	CD=O	Begin listing generated object code with this comment, and continue listing until an ≠OBJNOLIST≠ directive is encountered.
≠OBJNOLIST≠	CD=O	Stop listing object code, and do not list until an ≠OBJLIST≠ directive is encountered.
≠CHECKON≠	CD=S	Include code to begin checking subscripts in array references to see if the array bounds are exceeded. When bounds are exceeded at execution time, issue a fatal error message. Continue checking until a ≠CHECKOFF≠ directive is encountered.
≠CHECKOFF≠	CD=S	Stop generating code to check array reference subscripts. At execution time, requests to store or fetch values out of bounds are honored unless field length is exceeded.

The ALGOL compiler is called with a control statement whose keyword is ALGOL5 in any of the following forms:

ALGOL5(p_1,p_2, . . . ,p_n) comments

ALGOL5,p_1,p_2, . . . ,p_n. comments

ALGOL5. comments

The parameters (p_1,p_2, . . . ,p_n) can be written in any order and are separated by commas. Comments are optional; they are written on the dayfile but are ignored by the system.

The control statement must conform to operating system syntax. In particular, it can be continued on a second line under NOS/BE or SCOPE 2 but not under NOS.

All parameters are optional. The effect of omitting a parameter is described under the definition of that parameter. Parameters are presented in this section in alphabetical order.

B BINARY OUTPUT FILE

omitted	Same as B=LGO.
B	Same as B=BIN.
B=0	No binary output is generated.
B=lfn	Relocatable compiled code is written to file named lfn. The file is not rewound before or after compilation.

CD COMMENT DIRECTIVES OPTION

More than one comment directives option can be selected by separating the options with slashes; for example, CD=I/S. See section 10 for comment directives.

Omitted	No comment directive options are selected.
CD=I	Honor ≠INCLUDE≠ command directive.
CD=L	Honor ≠LIST≠, ≠NOLIST≠, and ≠EJECT≠ comment directives.
CD=O	Honor ≠OBJLIST≠ and ≠OBJNOLIST≠ comment directives.
CD=S	Honor ≠CHECKON≠ and ≠CHECKOFF≠ comment directives.

DB DEBUGGING OPTION

More than one debugging option can be selected by separating the options with slashes; for example, DB=D/P.

Omitted	No debugging options are selected.
DB=D	Information required for execution time symbolic dump is included in object code. Array elements are not dumped. See section 14.
DB=DA	Same as DB=D, with addition of array elements.
DB=P	Preset all non-own variables at block entry to negative indefinite for real and integer, and true for Boolean. Overridden by Z option on execution control statement (section 13).
DB=SB	Perform subscript bounds checking for arrays, regardless of ≠CHECKON≠ and ≠CHECKOFF≠ directives.

EL ERROR SEVERITY LEVEL

See appendix B for a discussion of error messages and their severity level.

Omitted	Same as EL=W.
EL	Same as EL=F.
EL=T	Trivial errors, plus errors of levels W, F, and C are listed. Trivial errors indicate suspicious usages. The syntax is correct, and the output is executable, but the compiler has reason to believe that what the program specifies is not what is intended.
EL=W	Warning errors, plus errors of levels F and C, are listed. Warning errors are those in which the syntax is incorrect, but the compiler has made an assumption about what was intended and has continued compilation. The binary is output, but the program might not run as intended.
EL=F	Fatal errors, plus errors of level C are listed. Fatal errors prevent the compiler from compiling the statement in which the error is detected. The binary is not executable. Unresolvable semantic errors fall in this category; the actual error might have occurred in a statement other than the one flagged.
EL=C	Only catastrophic errors are listed. Catastrophic errors are those from which the compiler cannot recover. Compilation of the current program is discontinued and the compiler advances to the next program.

ET ERROR TERMINATION

If an ET option is selected, and errors of the specified type occur, the job step aborts to an EXIT(S) control statement (under NOS/BE or SCOPE 2) or an EXIT control statement (under NOS) when compilation finishes. For an explanation of the error levels, see the EL parameter.

Omitted	Same as ET=C.
ET	Same as ET=F.
ET=0	Do not abort job step even if errors occur.
ET=T	Abort job step if errors of level T or higher have occurred.
ET=W	Abort job step if errors of level W or higher have occurred.
ET=F	Abort job step if errors of level F or higher have occurred.
ET=C	Abort job step if errors of level C have occurred.

I SOURCE INPUT FILE

Omitted	Source input is on file INPUT.
I	Source input is on file COMPILE.
I=lfn	Source input is on file lfn.

L OUTPUT LISTING FILE

Omitted	Output is listed on file OUTPUT.
L	Output is listed on file LIST.
L=0	Diagnostics only are listed on file OUTPUT.
L=lfn	Output is listed on file lfn.

LO OUTPUT LISTING OPTIONS

More than one listing option can be selected by separating the options with slashes; for example, LO=O/S. Any option can be negated by prefixing it with a minus sign.

Omitted	Same as LO=S.
LO	Same as LO=R/S.
LO=O	Object and source listing but no reference map.
LO=-O	Source listing but no object listing or reference map.
LO=R	Source listing and reference map.
LO=R/-S	Reference map only.
LO=S	Source listing only.
LO=-S	No source listing, object listing, or reference map.
LO=-S/O	Object listing but no source listing or reference map.

N CIRCUMLUDE COMPILATION

Omitted	Source input contains programs and separately compiled procedures only.
N	Source input contains circumludes only. See section 9.

OPT OPTIMIZATION LEVEL

Omitted	No extra optimizations performed.
OPT=IS	Instruction scheduling performed.

PD PRINT DENSITY

Omitted	Same as PD=6.
PD	Same as PD=8.
PD=6	Compilation listings produced at a density of six lines per inch.
PD=8	Compilation listings produced at a density of eight lines per inch.

PS PAGE SIZE

Omitted	Same as PS=60 if PD=6; same as PS=80 if PD=8.
PS=n	Maximum number of lines per page for compilation listings is n. If n is less than 4, the default is substituted.

PW PAGE WIDTH

Omitted	Same as PW=72 if output file is a terminal file; same as PW=136 if output file is a printer file.
PW=n	The number of characters on a line of the compilation time output listing is n. Values less than 50 or greater than 136 are ignored.

RES RESERVED WORDS

Omitted	ALGOL keywords are delimited by the ≠ character. See section 2.
RES	ALGOL keywords are recognized as reserved words, and are delimited by blanks or ≠.

S SYSTEM TEXT FOR CIRCUMLUDE

Omitted	Only standard circumlude is available for compilation.
S=circ	Circumlude named circ from library ALG5LIB is available during compilation.
S=lib-circ	Circumlude named circ from library named lib is available during compilation.

SEQ SEQUENCED INPUT

Omitted	Same as SEQ=0.
SEQ	Input is in sequenced line number format. Sequenced format is explained in section 1.
SEQ=0	Input is in unsequenced format.

SGM SEGMENT DIRECTIVES

Omitted	Provide no special code to verify the proper segmentation of program.
SGM	Provide special code to allow segmentation of procedures. See section 13 for details about the effect of this parameter.

SW SOURCE LINE WIDTH

Omitted	Same as SW=72.
SW	Same as SW=80.
SW=n	Columns 1 through n of each source line are compiled. The remaining columns are listed, but are otherwise ignored.

V VIRTUAL ARRAYS

Omitted	Virtual arrays are to be allocated in central memory. See section 2, under Arrays.
V	Virtual arrays are to be allocated in extended memory (ECS, LCM, or LCME).

COMPILATION LISTINGS

The format and contents of the listings output by the compiler depend on several control statement options:

- The LO option specifies whether the listing is to include a source listing, an object listing, a reference map, or any combination of the three.
- The EL option determines the levels of diagnostics to be listed.
- The L option names the file on which the listing is to be written. If L=0 is specified, then diagnostics are written to the file OUTPUT.
- The PD option specifies the number of lines per inch.
- The PS option specifies the number of lines per page.
- The PW option specifies the width of each line.

The format of the diagnostics is shown in appendix B.

An example of a source listing for program COMPLEX (described in section 1) is shown in figure 11-1. The components of the listing are identified.

An example of an object listing of program COMPLEX is shown in figure 11-2.

An example of a cross-reference map for program COMPLEX is shown in figure 11-3. For each identifier the following information is listed:

- Name. Identifiers are listed in alphabetical order. (Heading: NAME)
- Line number on which the identifier is declared. (Heading: DECL)
- Name of the procedure or program in which the identifier is declared, truncated to seven characters. (Heading: SCOPE)
- Line number of the start of the block in which the procedure is declared. (Heading: START)
- Kind: variable, array, procedure, label, switch, or string. Blank indicates a variable. (Heading: KIND)
- Type: integer, real, or Boolean. (Heading: TYPE)
- Attributes, such as formal parameter, value parameter or external. (Heading: ATTR)
- Line numbers of all occurrences of the identifier, in ascending order. (Heading: USAGE/FLAG) Each line is accompanied by a one-character flag indicating the following:

Letter	Significance
S	Identifier is assigned a value.
P	Identifier is used as an actual parameter.
C	Identifier is used as the controlled variable of a FOR statement.
blank	Other, such as use of value of the identifier in an expression.

Address of the identifier relative to the beginning of the section of code it falls within. (Heading: LOC)

Relocation, or type of block in which the identifier is located (such as // for blank common, INLINE for a standard procedure, or CODE for a code block). (Heading: REL)

Redundant declaration indicator. If an identifier is either declared twice, or is declared once but never referenced, the character < appears on the left of the identifier's line in the map. If the value of the identifier is used before a value has been assigned, the character * appears on the left. If the variable is accessible for future compilations (which can only be true during a circumlude compilation), the letter X appears on the left.

```
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM

      COMPLEX      * SOURCE LISTING *        ALGOL  5.1 79122     06/14/79  16.30.51.            PAGE      1

   1.          COMPLEX:                                                                        0  COMMENT
   2.          ≠BEGIN≠                                                                         1
   3.          ≠COMMENT≠     THIS PROGRAM COMPUTES THE SQUARE ROOT OF A COMPLEX NUMBER.        1  COMMENT
   4.                        THE COMPLEX NUMBER IS REPRESENTED AS (X,Y).                       1  COMMENT
   5.                        COPYRIGHT 1967, ASSOCIATION FOR COMPUTING MACHINERY.              1  COMMENT
   6.                        USED BY PERMISSION:                                               1  COMMENT
   7.                                                                                          1  COMMENT
   8.             ≠REAL≠ ≠PROCEDURE≠ CABS (X,Y);                                               1
   9.                ≠COMMENT≠ CALCULATE THE ABSOLUTE VALUE OF COMPLEX NUMBER (X,Y):           1  COMMENT
  10.                ≠VALUE≠ X,Y: ≠REAL≠ X,Y:                                                  1
  11.                ≠BEGIN≠                                                                   2
  12.                   X := ABS(X): Y := ABS(Y):                                              2
  13.                   CABS := ≠IF≠ X = 0 ≠THEN≠ Y                                            2
  14.                      ≠ELSE≠ ≠IF≠ Y = 0 ≠THEN≠ X                                          2
  15.                      ≠ELSE≠ ≠IF≠ X > Y                                                   2
  16.                         ≠THEN≠ X * SQRT(1+(Y/X)**2)                                      2
  17.                      ≠ELSE≠ Y * SQRT(1+(X/Y)**2 ):                                       2
  18.                ≠END≠ CABS:                                                               1  COMMENT
  19.                                                                                          1  COMMENT
  20.             ≠PROCEDURE≠ CSQRT (X,Y,A,B);                                                 1
  21.                ≠COMMENT≠ CALCULATE THE SQUARE ROOT OF THE COMPLEX NUMBER:                1  COMMENT
  22.                ≠VALUE≠ X,Y: ≠REAL≠ X,Y,A,B:                                              1
  23.                ≠BEGIN≠                                                                   2
  24.                   ≠IF≠ X = 0 ≠AND≠ Y = 0 ≠THEN≠ A := B := 0                              2
  25.                   ≠ELSE≠                                                                 2
  26.                   ≠BEGIN≠                                                                3
  27.                      A := SQRT((ABS(X)+CABS(X,Y))*0.5):                                  3
  28.                      ≠IF≠ X >= 0 ≠THEN≠ B := Y/(A+A)                                     3
  29.                      ≠ELSE≠                                                              3
  30.                      ≠BEGIN≠                                                             4
  31.                         B := ≠IF≠ Y < 0 ≠THEN≠ -A ≠ELSE≠ A:                              4
  32.                         A := Y/(B+B):                                                    4
  33.                      ≠END≠                                                               3
  34.                   ≠END≠                                                                  2
  35.                ≠END≠ CSQRT:                                                              1  COMMENT
  36.                                                                                          1  COMMENT
  37.          ≠BEGIN≠                                                                         2
  38.             ≠COMMENT≠ MAIN PROGRAM.................                                      2  COMMENT
  39.                       CHOOSE A VALUE FOR (X,Y) AND FIND ITS SQUARE ROOT (A,B):           2  COMMENT
  40.             ≠REAL≠ X,Y,A,B: ≠INTEGER≠ OUTP:                                              2
  41.                OUTP := 61:                                                               2
  42.                X := 1.0: Y := 0.0:                                                       2
  43.                CSQRT(X,Y,A,B):                                                           2
  44.                OUTPUT (OUTP,≠(≠2B,+7D.DD,2B,+7D.DD≠)≠, A, B):                            2
  45.          ≠END≠ MAIN:                                                                     1  COMMENT
  46.          ≠END≠                                                                           0
```


Figure 11-1. Source Listing

```
COMPLEX     * OBJECT CODE *                          ALGOL  5.1 79122     06/14/79  16.30.51.     PAGE      2


      000000  031715201405300000012+          VFD   42/031715201405308,18/G000012
      000001  000000020000000000006+          VFD   24/2,18/0,18/G000006
      000002  330000000000000003C1            VFD   60/A+3300000000000000000008
      000003  330000000000000004C1            VFD   60/B+3300000000000000000008
      000004  172040000000000000000           DATA  172040000000000000008
      000005  171740000000000000000           DATA  171740000000000000008
      000006  060005060003030202018           DATA  060005060003030202018
      000007  061000020600030403008           DATA  061000020600030403008
      000010  011307000206050010018           DATA  011307000206050010018
      000011  020720020000000000008           DATA  020720020000000000008
                                               USE   //          C1
      000000                          BLK.COM  BSS   1
      000001                          X        BSS   1
      000002                          Y        BSS   1
      000003                          A        BSS   1
      000004                          B        BSS   1
      000005                          OUTP     BSS   1
                                               USE   /A60$ST$/   C2
                                               USE   /A60$ST2/   C3
Line Numbers
from Source
Listing 000000                                 BSS   1
      000001                          MOPTION  BSS   1
      000002  Relative Address        IOPTION  BSS   1
              of Code                          USE   /COMPLEX/   C4
                                               USE
                       Machine Code
2.    000012            Generated    G000012
2.    000012  66200                  COMPLEX  SB2   B0
                  7110000016+                  SX1   PRGENT
      000013  71600000000C1                    SX6   BLK.COM
                        0200000000X            JP    =XA60$ST=
      000014  031715201405300000000+           VFD   42/031715201405308,18/G000000
      000015  100000000001600000000            VFD   6/8,18/0,18/G000160,18/0
      000016  6143777763               PRGENT   SB4   B3+APPETITE
                        0641000020+            GE    B4,B1,*+2
      000017  77114                            SX1   -B4+B1
                    0100000000X                RJ    =XG+STRQ
      000020  51600000000C1                    SA6   BLK.COM
R.                      0400000126+    G000020  EQ    G000126
      000021  000022222622226                  VFD   15/2,15/9366,15/9366
                              00000            VFD   15/0
B.    000022  030102230000000000C0+             VFD   42/030102230000008,18/G000000
      000023  030000010000520000001            VFD   6/3,18/G000001,18/G000052,18/1
      000024  6143777774               CABS     SB4   B3+APPETITE
                        0641000026+            GE    B4,B1,*+2
      000025  77114                            SX1   -B4+B1
                    0100000000X                RJ    =XG+STRQ
      000026  55610                            SA6   B1
```

Figure 11-2. Object Code (First Page Only) (Sheet 1 of 2)

```
12.                  5111777776                  SA1   B1+X
                                 10011           BX0   X1
                                                 AX0   60
      000027  21074                              BX6   X3-X1
                    13601                        SA6   A1
                          54610
12.                               46000          NO
      000030  5121777775                         SA2   B1+Y
                          22002                  LX0   B0,X2
                                 21074           AX0   60
      000031  13702                              BX7   X0-X2
```

Figure 11-2. Object Code (First Page Only) (Sheet 2 of 2)

```
        COMPLEX    * CROSS REFERENCE *                          ALGOL  5.1 79122     06/14/79  16.30.51.     PAGE      7

   NAME     TYPE  KIND   ATTR   DECL    SCOPE    START    LOC     REL       USAGE/FLAG  (S:SET, P:ACT.PAR., C:CONTR.VAR.)

   A          R           F.P.    22  CSQRT       20       3                24S   27S   28    28    31    31    32S
   A          R           F.P.    40  COMPLEX     37  000003  //            43P   44P
   ABS        R  PROC()   XREF        A60$ST=    189          INLINE        12    12    27
   B          R           F.P.    22  CSQRT       20       4                24S   28S   31S   32    32
   B          R           F.P.    40  COMPLEX     37  000004  //            43P   44P
   CABS       R  PROC()              8  COMPLEX     2  000024  CODE          13S   27
   CSQRT         PROC()             20  COMPLEX     2  000060  CODE          43
   OUTP       I                  40  COMPLEX     37  000005  //            41S   44P
   OUTPUT        PROC()   XREF        A60$ST=    189                        44
   SQRT       R  PROC()   XREF        A60$ST=    189                        16    17    27
   X          R           V.P.    10  CABS         8       1                12S   12P   13    14    15    16    16    17
   X          R           V.P.    22  CSQRT       20       1                24    27P   27P   28
   X          R                   40  COMPLEX     37  000001  //            42S   43P
   Y          R           V.P.    10  CABS         8       2                12S   12P   13    14    15    16    17    17
   Y          R           V.P.    22  CSQRT       20       2                24    27P   28    31    32
   Y          R                   40  COMPLEX     37  000002  //            42S   43P

REQUIRED  CM 036000. CP    .907 SEC.
```

Figure 11-3. Reference Map

INTERFACES 12

This section describes the interfaces between an ALGOL program and Record Manager (which processes all the input/output for ALGOL) as well as guidelines for writing a procedure in COMPASS intended to be called from an ALGOL program. A COMPASS procedure called from ALGOL must be a separately compiled procedure; how to declare and call these procedures from ALGOL is explained in section 9. This section explains modifications to the COMPASS program which enable it to interface properly with ALGOL.

Procedures written in FORTRAN and SYMPL can also be compiled separately and called from ALGOL. This facility is described in section 9.

RECORD MANAGER INTERFACE

Calls to input/output procedures (section 8) in an ALGOL program are translated by the compiler into calls to Record Manager, which performs the input/output at execution time.

For each file (which is equivalent to a channel), Record Manager requires, and the compiler provides, a file information table, which guides the Record Manager processing. This table contains fields that specify the structural characteristics of the file. The fields in this table are set by the compiler at execution time when the CHANNEL procedure is executed, or by default if the logical file name is INPUT or OUTPUT. If the user provides a FILE control statement, the values provided override those established by the CHANNEL procedure. The FILE control statement is processed when the file is opened. A file is opened either explicitly, by the OPEN procedure (section 8), or implicitly, when the first input/output procedure referencing the file is executed. Table 12-1 lists the file information table settings for each of the three file types (coded sequential, binary sequential, and word addressable). For the meanings of these parameters, and more information in general, see the CYBER Record Manager/Basic Access Methods reference manual, or the SCOPE 2 Record Manager reference manual.

COMPASS INTERFACE

A COMPASS subroutine written to be called as a separately compiled procedure from an ALGOL program can be declared in the program by any of the symbols ≠CODE≠, ≠ALGOL≠, or ≠FORTRAN≠. ≠CODE≠ and ≠ALGOL≠ are synonymous, and indicate that the procedure is to be called with the ALGOL calling sequence. If the procedure is declared with the symbol ≠FORTRAN≠, the FORTRAN Extended calling sequence is used, as described in the FORTRAN Extended reference manual. The procedure is entered through the RJ instruction, and return must be through a branch to the entry point. ALGOL saves and restores all relevant registers.

If the procedure is declared with the symbol ≠RJ≠, an especially fast calling sequence is used. The procedure must have no more than one parameter, which must be called by value and cannot be an array. The procedure is entered by an RJ instruction; the value of the parameter is in register X1. The procedure should not change the contents of registers B1, B2, and B3 without restoring them before exit. The result of the procedure (if it is a function) must be in register X6 when control returns to the calling program unit.

TABLE 12-1. FILE INFORMATION TABLE DEFAULTS

FIT Field		Coded Sequential	Binary Sequential	Word Addressable
Mnemonic	Meaning			
FO	File organization	SQ	SQ	WA
RT	Record type	Z (NOS, NOS/BE) W (SCOPE 2)	S	U
BT	Block type	C (NOS, NOS/BE) unblocked (SCOPE 2) I (SCOPE 2 tape files)	C	--
FL	Fixed record length (RT=Z), in characters	137 for paged, 80 for unpaged	--	--
MRL	Maximum record length, in characters	--	5120	5120
OF	Open flag	N	N	N
CF	Close flag	N	N	N
PD	Processing direction	IO	IO	IO
LT	Label type	U	U	--

Location	Operation	Variable
tname	PROC	name,type,parlist,fast,list

tname — Optional. If present, this name is used in the traceback.

name — Entry point name. Either a number of one to five digits, or a name of up to seven letters and digits, the first a letter. If a number, it is prefixed with leading zeros and the letters CD to create an acceptable 7-character entry point name for the loader. The name provided in the macro should appear after the ≠CODE≠ symbol in the procedure declaration.

type — Optional. If absent, procedure is assumed to be a typeless procedure. If present, must be one of the following, to indicate the type of the procedure:

I Integer
R Real
B Boolean

parlist — List of formal parameter descriptions. The whole list must be parenthesized, and each description must be parenthesized. The list is of the form:

((p,k,t,v,d), . . . , (p,k,t,v,d))

Parameters of descriptions are as follows:

p Name by which parameter is referred to in macros.

k Letter indicating kind of parameter; one of the following:

G String
A Array
C Constant
V Simple variable
L Label
P Procedure with parameters
W Switch
X Expression
N Procedure without parameters
S Subscripted variable
R Virtual array

t Type of parameter:

I Integer
B Boolean
R Real
omitted Type not relevant

v If nonblank, value parameter; if omitted, name parameter.

d For arrays only. If present, number indicating required number of dimensions of array. If omitted, any number is permitted.

fast — Optional. If absent, generate stack request. If present, suppress stack request (only allowed when the procedure does not use the scalar stack).

list — Optional. If present, additional parameters are permitted, and are treated as if call-by-name. If omitted, no additional parameters are permitted.

Figure 12-1. PROC Macro Format

COMPASS INTERFACE MACROS

The COMPASS interface macros simplify the process of writing a COMPASS subroutine meant to be called as a procedure from an ALGOL program. Calls to the macros are included in a COMPASS procedure and must follow all the conventions for macro calls as described in the COMPASS reference manual. To use the macros, S=AL5TEXT must be specified on the COMPASS control statement.

The steps to be followed for separate compilation of procedures written in COMPASS are outlined in section 9. When the interface macros are used, the procedure must be declared in the calling program unit using the symbols: ≠ALGOL≠ or ≠CODE≠ (not ≠FORTRAN≠ or ≠RJ≠). This is so that the usual ALGOL calling sequence is used. COMPASS procedures not using the interface macros can be declared with the symbols ≠FORTRAN≠ or≠RJ≠.

Macros are of six types:

- Procedure definition. These macros define the beginning and end of a procedure, and return control to the calling procedure.
- Parameter checking. These macros check the number, type, and kind of the actual parameters the procedure was called with.
- Parameter accessing. These macros make available to the procedure the values and locations of the actual parameters.
- Procedure-calling. These macros enable the procedure to call other procedures.
- Stack-handling. These macros obtain space on the scalar stack and store values on, or fetch them from, the stack.
- Miscellaneous. These macros convert values, post error messages, and save and restore B-registers.

In procedures using the macros, the character ↓ (ASCII ?) should be avoided, since it is heavily used in a variety of contexts by the macros.

There are several registers that should not be used by a procedure, unless they are saved after each macro call and restored before the next macro call. These include B1, B2, and B3, which are used by the run-time system, and X0, X1, X2, X6, A1, A2, A6, and B4, which are used by the macros themselves. If an X-register is used as a parameter to a macro call, the corresponding A-register might be used by the macro. When it is necessary for a macro to evaluate a call-by-name parameter, however, all registers might be used by the macro; this applies only to the VALUE, ASSIGN, and ADDRESS macros. This is also true whenever a stack request is necessary; this applies only to the PROC macro. Registers B1, B2, and B3 can be saved by the macro SBREGS and restored by the macro RBREGS.

The macros attempt to optimize register usage; the user can consult a COMPASS listing in which the macros are expanded to see which registers are actually used in any given case.

PROCEDURE DEFINING MACROS

The procedure defining macros establish and disconnect the linkage between the COMPASS procedure and the ALGOL run-time system.

The PROC macro (figure 12-1) establishes the entry point of the procedure and must be called in each procedure using the interface macros. It should be the first code generating instruction. Procedures cannot be nested; that is, an ENDPROC macro call must occur before the next PROC macro call is allowed.

The ENDPROC macro (figure 12-2) indicates the end of the procedure. It should be called as the last code generating instruction of the procedure. ENDPROC assigns values to local variables needed by other macros. It does not transfer control; this can only be accomplished through the RETURN macro.

Location	Operation	Variable
	ENDPROC	

Figure 12-2. ENDPROC Macro Format

The RETURN macro (figure 12-3) returns control to the calling procedure. It can appear more than once in a procedure. If the procedure has a result, it is put in an X-register, and the xreg parameter indicates which register.

Location	Operation	Variable
	RETURN	xreg,type

xreg	For typed procedures, name of X-register containing result. Omitted for typeless procedures.	
type	Optional; indicates type of number in X-register and conversion to be performed:	
	R	Floating point number, to be normalized.
	I	Floating point number, to be rounded to nearest integer.
	B	Boolean value, no conversion performed.
	F	Integer, to be floated and normalized.
	omitted	No conversion performed.

Figure 12-3. RETURN Macro Format

PARAMETER CHECKING MACROS

The parameter checking macros check the number, type, and kind of actual parameters the procedure has been called with. They allow the user to take alternative courses of action depending on the actual parameters.

The PARAMS macro (figure 12-4) returns the number of actual parameters. No check is made by the system as to whether this number is the same as the number of formal parameters specified in the PROC macro.

Location	Operation	Variable
	PARAMS	xreg

xreg — Name of X-register to receive number of actual parameters; number is returned as an integer.

Figure 12-4. PARAMS Macro Format

The KIND macro (figure 12-5) must be called once for each parameter for which checking is desired. In the macro call, the user specifies labels to be branched to when the actual parameter is one of the specified kinds. If the actual parameter is not of any of the specified kinds, control falls through to the next statement after the macro call.

Location	Operation	Variable
	KIND	parname, (k:lab, . . . , k:lab)

parname — Parameter name, as established via the PROC macro; an integer parameter number; or the B register containing the parameter number.

k — Any concatenation of characters indicating the kinds to be checked. The characters are those shown in figure 12-1 for the PROC macro.

lab — COMPASS label to be branched to if the actual parameter is one of the specified kinds.

Figure 12-5. KIND Macro Format

The TYPE macro (figure 12-6) is the same as the KIND macro, except that it checks types instead of kinds.

Location	Operation	Variable
	TYPE	parname, (t:lab, . . . , t:lab)

parname — Same as for KIND.

t — Concatenation of characters indicating type to be checked for. Allowable codes are:

- I Integer
- R Real
- B Boolean
- N No type; allowed only for a label, switch, procedure, or string.

lab — Label to be branched to if actual parameter is one of the specified types.

Figure 12-6. TYPE Macro Format

PARAMETER ACCESSING MACROS

The parameter accessing macros make available to the COMPASS procedure the values and addresses of actual parameters, the lengths of strings and arrays, and the number of dimensions in an array. They also assign values to call-by-name parameters and transfer control to label parameters.

The VALUE macro (figure 12-7) computes the value of a specified actual parameter and places it in a specified X-register.

Location	Operation	Variable
	VALUE	parname,xreg,kind,check

parname — Same as for KIND.

xreg — Name of X-register to receive value of parameter.

kind — Optional; one of the allowable kinds for this parameter. The only kinds allowed are C, V, X, N, and S (as listed in figure 12-1). The codes are the same as for the PROC macro (figure 12-1). For call-by-name parameters, more efficient code is produced if this parameter is included.

check — Optional; only meaningful for call-by-name parameters. If omitted, no check is performed. If specified as I, B, or R, kind is checked according to the kind parameter and type is checked for integer, Boolean, or real, respectively. If kind or type does not match, a fatal error results. If specified as a nonblank value other than I, B, or R, kind is checked but not type.

Figure 12-7. VALUE Macro Format

The ADDRESS macro (figure 12-8) computes the first word address of an array or string, or the address of a constant or variable. The parameter must be call-by-name.

Location	Operation	Variable
	ADDRESS	parname,xreg,kind,check

parname — Same as for KIND.

xreg — Name of X-register to receive address of parameter. For kind = R, the sign bit is 1 if the array is allocated in extended memory, and 0 if the array is allocated in central memory.

kind — One of the allowable kinds for this parameter. Only G, C, A, V, R, and S are allowed, as listed in figure 12-1.

check — Optional; if specified, the kind of the parameter is checked; if it does not match the kind parameter, the program aborts. If omitted, no checking takes place.

Figure 12-8. ADDRESS Macro Format

The FWA macro (figure 12-9) performs the same function as ADDRESS, except that it only applies to arrays and strings, and the kind of the parameter is not specified.

Location	Operation	Variable
	FWA	parname,xreg,check

parname, xreg, check	See VALUE macro.

Figure 12-9. FWA Macro Format

The LENGTH macro (figure 12-10) computes the length of an array or string; the parameter must be call-by-name.

Location	Operation	Variable
	LENGTH	parname,xreg,kind,check

parname, kind	See VALUE macro.
check	See ADDRESS macro (figure 12-8).
xreg	Name of X-register to receive length of array or string. The length is returned in characters for a string and in words for an array. String length does not include string quotes.

Figure 12-10. LENGTH Macro Format

The ORDER macro (figure 12-11) returns the number of dimensions of an actual parameter array.

Location	Operation	Variable
	ORDER	parname,xreg,check

parname	See VALUE macro.
xreg	X-register to receive number of dimensions in array.
check	See ADDRESS macro (figure 12-8).

Figure 12-11. ORDER Macro Format

The ASSIGN macro (figure 12-12) assigns a value to a call-by-name parameter that is a simple or subscripted variable, or to any call-by-value parameter.

The GOTO macro (figure 12-13) transfers control to a designational expression (label or switch) provided as the value of an actual parameter. The environment of the procedure to which control is transferred is restored.

Location	Operation	Variable
	ASSIGN	parname,xreg,kind,check

parname, kind, check	See VALUE macro.
xreg	X-register containing value in correct format to be assigned to parameter.

Figure 12-12. ASSIGN Macro Format

Location	Operation	Variable
	GOTO	parname,index,check

parname	See VALUE macro.
index	Required for switch; forbidden for label. Indicates which label in switch is to be transferred to.
check	See ADDRESS macro (figure 12-8).

Figure 12-13. GOTO Macro Format

PROCEDURE-CALLING MACROS

The procedure-calling macros set up the parameters for a call to another procedure and then execute the call. The macros should be executed in this order:

1. The macro CALLING (figure 12-14) is called to establish that a calling sequence is being established.

Location	Operation	Variable
	CALLING	npar,name

npar	Absolute expression equal to the number of actual parameters in the call.
name	If absolute expression or B-register: number of parameter to this procedure (the calling procedure) which is the procedure to be called. If relocatable or external expression: name of entry point of procedure to be called.

Figure 12-14. CALLING Macro Format

2. For each actual parameter, either the macro PARAM (figure 12-15) or one of the macros SIMPLE (figure 12-16), STRING (figure 12-17), or CONSTANT (figure 12-18) is called to place the actual parameter on the stack. If the parameter is a call-by-value parameter, its value is placed on the stack; if it is a call-by-name parameter, a specially formatted descriptor is placed on the stack. PARAM can be called for any kind of parameter, call-by-name or

call-by-value; SIMPLE, STRING, and CONSTANT can be called for call-by-name parameters that are simple variables, strings, and constants, respectively. For call-by-name parameters, PARAM requires that the user provide the descriptor; for the others, the descriptor is provided automatically. The formats of descriptors are presented in appendix E.

Location	Operation	Variable
	PARAM	par

par	If X-register: the contents of the register are stored on the stack. If the parameter is call-by-value, the contents are interpreted as the value of the parameter; if call-by-name, as the descriptor of the parameter. Descriptor formats are presented in appendix E. If the procedure being called is a parameter of the current procedure, all actual parameters must be call-by-name. If B-register or absolute expression: the value is interpreted as the index of the parameter to this procedure (the calling procedure) which is to be transmitted as a parameter to the called procedure.

Figure 12-15. PARAM Macro Format

Location	Operation	Variable
	SIMPLE	type,name

type	Type of parameter: I Integer R Real B Boolean
name	If absolute expression: index of simple variable in the stack. If relocatable expression: address of variable.

Figure 12-16. SIMPLE Macro Format

Location	Operation	Variable
	STRING	name,str

name	Optional; required when str parameter is not present. Address in code of descriptor for string.
str	Optional; required when name parameter is absent. String contained in parentheses, not containing ≠ or $.

Figure 12-17. STRING Macro Format

Location	Operation	Variable
	CONSTANT	type,value

type	Type of constant: I Integer R Real B Boolean
value	Value of constant. If Boolean, value is T or F.

Figure 12-18. CONSTANT Macro Format

3. The CALL macro (figure 12-19) is called, bringing the procedure into execution. The next statement after the call to CALL is the first statement to be executed after return from the procedure. If the procedure is typed, the value is in register X5. If the procedure being called is a parameter of the current procedure, all actual parameters must be call-by-name. In case the corresponding formal parameter in the called procedure is declared ≠VALUE≠, the system automatically evaluates the parameter and the parameter is afterwards treated as a call-by-value parameter.

Location	Operation	Variable
	CALL	

Figure 12-19. CALL Macro Format

STACK-HANDLING MACROS

The stack-handling macros request additional space on the scalar stack, and fetch and store variables. The scalar stack is described in appendix E. The DECL macro (figure 12-20) requests a specified number of words of additional scalar stack space. This macro uses no registers. The PUTVAR macro (figure 12-21) places a variable on the stack. The GETVAR macro (figure 12-22) retrieves the value of a variable from the stack. In both cases, the index of the variable provided is not checked for validity at execution time. If the value of the index is an absolute expression, it is checked at compilation time.

Location	Operation	Variable
	DECL	name,length

name	Name of location or register to receive index of first word of available stack space after macro call is executed. Name is the stack address, relative to the base address in register B1.
length	Optional; absolute expression equal to the number of words requested. If omitted, value assumed is 1.

Figure 12-20. DECL Macro Format

Location	Operation	Variable
	PUTVAR	xreg,index

xreg	X-register containing value to be stored.
index	Absolute expression, relocatable expression, or register containing index of variable on stack. The index is the stack address relative to the base address in register B1.

Figure 12-21. PUTVAR Macro Format

Location	Operation	Variable
	GETVAR	xreg,index

xreg	X-register to receive value of variable.
index	Same as for PUTVAR (figure 12-21).

Figure 12-22. GETVAR Macro Format

MISCELLANEOUS MACROS

The miscellaneous macros take care of various housekeeping functions. The XFORM macro (figure 12-23) converts a value in an X-register into a different format, and places it in the same or a different register.

The ERROR macro (figure 12-24) starts the error traceback and optionally writes a message to the post-mortem dump output file.

The SBREGS macro (figure 12-25) saves the dedicated B-registers (B1, B2, and B3) and the RBREGS macro (figure 12-26) restores them.

Location	Operation	Variable
	XFORM	operand,result,action

operand	X-register containing value to be reformatted.
result	X-register to receive result. Can be the same register as operand.
action	Type of conversion: RI Round to nearest integer. RF Convert operand from real to integer by unpacking and shifting. FR Convert operand from integer to real by packing and normalizing.

Figure 12-23. XFORM Macro Format

Location	Operation	Variable
	ERROR	string

string	Optional; error message enclosed in parentheses and not containing $ or, in 64-character set, two consecutive colons.

Figure 12-24. ERROR Macro Format

Location	Operation	Variable
	SBREGS	

Figure 12-25. SBREGS Macro Format

Location	Operation	Variable
	RBREGS	

Figure 12-26. RBREGS Macro Format

Compilation of an ALGOL program results in a relocatable binary. The program can be loaded and executed in the same way as any other relocatable binary. (See the loader reference manual.)

SEGMENT LOADING RESTRICTIONS

If the segment loader is used, the following restrictions should be observed:

- The main program and all circumludes should reside in the root segment. This takes place automatically when the main program is assigned to the root segment and the circumludes are not assigned to any segment.
- Code procedures can be assigned to nonroot segments. However, the results are not valid if a return is made to a calling segment when the segment is no longer loaded. For example, the following sequence produces invalid results:

  Segment A, containing procedure A1, is the ancestor of both segment B, which contains procedure B1, and segment C, which is on the same level as B. Procedure B1 calls procedure A1, which then causes segment C to be loaded on top of segment B. When A1 terminates and attempts to return to B1, segment B is no longer in memory and a branch is made to some irrelevant address in segment C. This situation cannot be diagnosed; the user must guard against it.

- Other situations that can cause invalid results are calling a parameter procedure, branching to a parameter label, and evaluating a name parameter when the procedure, label, or name parameter is in a different segment from that containing the called procedure.

The SGM parameter on the ALGOL5 control statement eliminates these restrictions in most cases. For main programs and circumludes, the SGM option is not necessary (but harmless). A separately compiled procedure to be executed under segmentation should be compiled with the SGM option if either of the following conditions is true:

- The procedure is called from a procedure in a different segment. This applies whether the call is direct (through a call to the procedure) or indirect (through a formal parameter).
- The procedure itself calls a procedure in a different segment, either directly or indirectly.

When a procedure is compiled under the SGM option, it must either have a name that does not exceed six characters, or a number. The module name must be different from the procedure name.

The main program of a segmented program should be loaded in the root segment. When one separately compiled procedure is compiled with the SGM option, all procedures compiled at the same time should also use the option.

EXECUTION CONTROL STATEMENT

Parameters can be included on the control statement that executes an ALGOL program (frequently the LGO control statement). The parameters are all optional and can be specified in any order, except on an EXECUTE statement, where the first parameter is the entry point name.

D POSTMORTEM DUMP FORMAT

The dump format is described in section 14. DB=D or DB=DA must have been specified on the ALGOL 5 control statement; inclusion of arrays depends on the compile time specification.

Omitted Same as D=DT.

D Same as D=DV.

D=ab This option is a two-character string. The first character, denoted by a, indicates the scope of code to be dumped. The second character, denoted by b, indicates the type of dump.

Choices for a:

- B Current block only
- P Current procedure
- S All accessible blocks (static)
- D All active blocks (dynamic)

Choices for b:

- T Traceback only
- V Traceback and dump of simple variables
- F Traceback and dump of array elements and simple variables

D=abA When the two characters are followed by A, the addresses of the entities are listed as well.

D=0 No dump or traceback information.

E POSTMORTEM DUMP FILE

Omitted Same as E=OUTPUT.

E Same as E=OUTPUT.

E=lfn Postmortem dump is written to file lfn.

I INCREMENT FOR MEMORY REQUEST

Omitted Same as I=2000.

I Same as I=2000.

I=n Minimum number of words by which field length can be increased is n (n is interpreted as an octal number). See MEMORY procedure, section 7.

L LINE LIMIT FOR DUMP

The line limit applies to both the postmortem dump (section 14) and to dumps induced by the DUMP procedure (section 7).

Omitted Same as L=200.

L Same as L=1000.

L=n Dump output limited to n lines.

M MAXIMUM FIELD LENGTH

Omitted Maximum field length determined by loader. (See the Loader reference manual.)

M Field length when execution begins is to be maximum field length.

M=n Maximum field length is n (n is an octal number). See MEMORY procedure, section 7.

R RECOVERY TYPE

R indicates under which conditions a recovery subroutine is to be executed (see the explanation of the RECOVR macro in the NOS or NOS/BE reference manual or the REPRIEVE macro in the SCOPE 2 reference manual). The n option specifies a mask that indicates the types of conditions that cause recovery.

Omitted Same as R=77. (R=77770014B for SCOPE 2)

R Same as R=77. (R=77770014B for SCOPE 2)

R=n Execute recovery subroutine under conditions indicated by mask n (n is an octal number, less than or equal to 77).

Z PRESET VALUE

Z is effective when DB=P has been selected on the ALGOL 5 control statement.

Omitted Numeric values are preset to negative indefinite; Boolean values to true.

Z Numeric values are preset to zero; Boolean values to false.

EXECUTION TIME ERROR PROCESSING 14

When a fatal error occurs during execution of an ALGOL program, execution terminates and diagnostic information is output. All the information is output to the file named in the E option on the execution control statement (section 13), or to the file OUTPUT by default. The following types of errors can occur at execution time:

- Errors detected by the ALGOL execution time system. These include array subscripts out of bounds (only when DB=SB is specified on the ALGOL5 control statement, or when CD=S is specified and the ≠CHECKON≠ directive is in effect for the statement causing the error); invalid correspondence between actual and formal parameters; invalid arguments to standard procedures; and input/output errors.

- Errors detected by the hardware. These are also known as arithmetic mode errors and include the use of indefinite and infinite operands, and references to addresses outside of the user's field length. More detail about these errors can be found in the appropriate operating system reference manual.

The information output when an error occurs consists of a diagnostic, traceback information, and possibly a symbolic dump. The amount of information included in the dump depends on the options selected for the DB parameter on the ALGOL5 control statement and the D parameter on the execution control statement. If neither DB=D nor DB=DA is selected on the ALGOL5 control statement, no dump is produced. If DB=D is selected, the dump includes the following:

- For each block in the traceback, a line indicating the module name and line number of the block, the values of all simple variables declared in the block, and the names, types, and bounds of each array in the block.

- For each procedure in the traceback, the name of the module in which the procedure is declared, as well as the line number of the declaration, and the values of the formal parameters of the procedures.

If DB=DA is selected, the values of array elements for each array declared in each block in the traceback are added to the information output for DB=D.

For the D parameter on the execution control statement, the options selected consist of two or three letters, as follows:

1. The first letter indicates the blocks to be included in the traceback:

   B Current block only

   P All active blocks in the current procedure

   S All accessible blocks in the static chain

   D All active blocks in the dynamic chain

2. The second letter indicates the entities to be dumped:

   T Traceback only

   V Traceback plus simple variables

   F Traceback, simple variables, and array elements

   If a more restrictive set of entities is specified on the ALGOL5 control statement, the execution control statement is overridden. For example, if DB=D is specified on the ALGOL5 control statement, and D=xF is specified on the execution control statement, array elements are not dumped because the required information is not added to the object code at compile time.

3. The third letter is either A or omitted. If A, stack addresses of entities are included in the dump, and COMPASS procedures are added to the traceback. If A is omitted, no addresses are included and COMPASS procedures cannot be traced.

For example, D=SVA means that all blocks in the static chain are dumped, and a traceback, simple variables, their values, and their stack addresses are output for each block. D=BF means that only the current block is traced, and all its variables and arrays are output, but not their addresses. If D=0 is specified, no traceback or dump is produced.

A dump is also produced when the standard procedure DUMP is called; this is called a dynamic dump as opposed to the postmortem dump produced for a fatal error. It is in the same format as the postmortem dump except for the wording of some messages. The format of the procedure call is shown in figure 14-1. The formats of the messages output for a dump are shown in table 14-1. Figure 14-2 shows a sample program and the dump it produces.

DUMP (ie1,ie2,s)

ie1 Channel number of file on which dump is to be output.

ie2 Identification number to be output in heading of dump.

s String consisting of two letters preceded by an optional plus or minus sign. If a plus sign is present, a page eject takes place before the dump is output. If a minus sign, two lines are skipped.

Values for the first letter:

B Current block only
P Current procedure only
S All accessible blocks in static chain
D All active blocks in dynamic chain

Values for the second letter:

T Traceback only
V Traceback plus simple variables
F Traceback plus simple variables and arrays

Figure 14-1. DUMP Procedure Format

TABLE 14-1. DUMP MESSAGE FORMATS

Message	Output For	Required Control Statement Parameters	
		ALGOL (DB=)	LGO (D=)
proc CALLED AT LINE line IN PROCEDURE proc, DECLARED AT LINE line, {AT STACK ADDRESS add} IN MODULE mod	Header line	any	any {A}
IN BLOCK STARTING AT LINE line OF proc	Block	any	any
type ident = value {ADDRESS add}	Variable	D,DA	V,F {A}
type ARRAY ident [bound pair list] {STARTING AT ADDRESS add}	Array	D,DA	V,F {A}
ident [row subscript list] =	Array row	DA	F
sub: value value value value value (num *)	Array elements	DA	F
FORMAL PARAMETERS OF PROCEDURE proc	Formal parameters header line	D,DA	P,S,D,V,F
type ident CALLED BY VALUE ACTUAL = value	Formal parameter variable called by value	D,DA	P,S,D,V,F
type ident CALLED BY NAME ACTUAL {VALUE=value / EXPRESSION}	Formal parameter variable called by name	D,DA	P,S,D,V,F
type ARRAY ident [bound pair list] CALLED BY VALUE	Formal parameter array called by value	DA	P,S,D,F
type ARRAY ident [bound pair list] CALLED BY NAME	Formal parameter array called by name	DA	P,S,D,F
PROCEDURE proc ACTUAL ident DECLARED AT LINE line OF MODULE mod	Formal parameter procedure	D,DA	P,S,D,V,F
STRING string ACTUAL CHARACTERS = ccccccccc	Formal parameter string	D,DA	P,S,D,V,F
{LABEL / SWITCH} label ACTUAL AT LINE line OF proc DECLARED AT LINE line OF MODULE mod	Label or switch parameter	D,DA	P,S,D,V,F

Key:

proc	Procedure name.
line	Line number, as listed on output listing.
add	Stack address.
mod	Module name.
type	REAL, INTEGER, or BOOLEAN.
ident	Variable or array identifier, truncated to 7 characters.
value	Value of variable or array element. Numbers are left justified, and only significant digits are shown. Boolean values appear as TRUE or FALSE.
bound pair list	Pairs of array subscript bounds for each dimension, separated by commas.
row subscript list	Subscripts, separated by commas, of all but the rightmost dimension, indicating which row of the array is listed next.
sub	Last subscript of first array element in line listing array element values. Five numeric or ten Boolean values are listed per line.
num	Number of consecutive array elements equal to the last one listed.
string	Name of string parameter.
C...C	Characters (maximum of 10).
label	Name of label or switch parameter.
{ }	Item is included only if A is selected on the execution control statement.

```
PROGRAM:
          WUNDERD              * SOURCE LISTING *          ALGOL  5.0 78318     12/06/78  13.12.43.     PAGE       1

       1.            WUNDERDUMP DEMONSTRATIEPROGRAMMA;                                                    0  COMMENT
       2.            ≠BEGIN≠                                                                              1
       3.            ≠COMMENT≠ THIS PROGRAM PRODUCES AN ARRAY BOUNDS ERROR AND THEN A DUMP;               1
       4.                                                                                                 1
       5.               ≠PROCEDURE≠ KNAL(P1,P2,P3); ≠VALUE≠ P1,P2;                                      1
       6.               ≠REAL≠ P1; ≠INTEGER≠ P2; ≠INTEGER≠ ≠ARRAY≠ P3;                                  1
       7.               ≠BEGIN≠                                                                         1
       8.                  ≠INTEGER≠ LOCAL;                                                             2
       9.                  ≠REAL≠ ≠ARRAY≠ LANG[-P2:P2];                                                 2
      10.                  ≠FOR≠ LOCAL := -P2 ≠STEP≠ 1 ≠UNTIL≠ P2 ≠DO≠                                  2
      11.                     LANG[LOCAL] := 0.0;                                                       2
      12.                  LANG[3201] := -5;                                                            2
      13.                  LOCAL := P1 + P2;                                                            2
      14.               ≠END≠    KNAL;                                                                1
      15.                                                                                              1
      16.               ≠INTEGER≠ I,J,NUL,AKTUEEL,MET EEN LANGE NAAM;                                   1
      17.               ≠REAL≠ ONEINDIG,ONGEDEFINIEERD,GROOT,KLEIN;                                     1
      18.               ≠BOOLEAN≠ ≠ARRAY≠ ONDERGRENS[-5:5];                                             1
      19.               ≠INTEGER≠ ≠ARRAY≠ ALPHATEXT[10:17];                                             1
      20.               ≠REAL≠ ≠ARRAY≠ DRIEDIM[1:3,1:3,1:20];                                           1
      21.                                                                                              1
      22.               ≠FOR≠ I := 1 ≠STEP≠  1 ≠UNTIL≠ 20 ≠DO≠                                          1
      23.                  ≠FOR≠ J := 1 ≠STEP≠ 1 ≠UNTIL≠ 3 ≠DO≠                                         1
      24.                     DRIEDIM[J,J,I] := I + J;                                                  1
      25.                                                                                              1
      26.               ≠BEGIN≠                                                                         1
      27.                  ≠INTEGER≠ I;                                                                 2
      28.                  ≠FOR≠ I := 10 ≠STEP≠ 1 ≠UNTIL≠ 17 ≠DO≠                                       2
      29.                     INPUT (60,≠(≠8A≠)≠,ALPHATEXT[I]);                                         2
      30.               ≠END≠     BLOCK;                                                              1
      31.                                                                                              1
      32.               GROOT := ≠+250;   KLEIN := 1/GROOT;                                            1
      33.               ONEINDIG := GROOT/KLEIN;                                                       1
      34.               AKTUEEL := 1111;                                                               1
      35.               ≠FOR≠ NUL := 0 ≠STEP≠ 1 ≠UNTIL≠ 10 ≠DO≠                                        1
      36.                  KNAL (ONEINDIG,AKTUEEL/2,ALPHATEXT);                                        1
      37.            ≠END≠    WUNDERDUMP                                                              0
 PROGRAM LENGTH     000306B WORDS
 REQUIRED  CM 037500. CP   .766 SEC.
```

Figure 14-2. Sample Dump (Sheet 1 of 3)

```
DUMP:

  ARRAY BOUNDS ERROR  FOUND BY ALGOL5
  THE ERROR WAS DETECTED DURING EVALUATION OF SUBSCRIPT  1 OF ARRAY *******


  POST-MORTEM DUMP STARTED

  PM-DUMP CALLED AT LINE   12 IN PROCEDURE KNAL    DECLARED AT LINE    8 IN MODULE WUNDERD

  ------------------------------------------------------------------------------------------------------------------------
  IN BLOCK STARTING AT LINE    8 OF KNAL

      INTEGER LOCAL   =  557
      REAL    ARRAY LANG    [ -556 :  556 ]
  LANG    [ * ] =
         -556 :  0.0 (  1113 * )
  ------------------------------------------------------------------------------------------------------------------------
  FORMAL PARAMETERS OF PROCEDURE KNAL

  REAL    P1      CALLED BY VALUE ACTUAL = + INFINITE
  INTEGER P2      CALLED BY VALUE ACTUAL =  556
  INTEGER ARRAY P3      [  10 :  17 ] CALLED BY NAME
  ------------------------------------------------------------------------------------------------------------------------

  KNAL    CALLED AT LINE   36 IN PROGRAM   WUNDERD DECLARED AT LINE    0 IN MODULE WUNDERD

  ------------------------------------------------------------------------------------------------------------------------
  IN BLOCK STARTING AT LINE    5 OF WUNDERD

      INTEGER I       =  21
      INTEGER J       =  4
      INTEGER NUL     =  0
      INTEGER AKTUEEL =  1111
      INTEGER METEENL = - INDEFINITE
      REAL    ONEINDI = + INFINITE
      REAL    ONGEDEF = - INDEFINITE
      REAL    GROOT   =  1.0≠+250
      REAL    KLEIN   =  1.0≠-250
      BOOLEAN ARRAY ONDERGR [ -5 :  5 ]
  ONDERGR [ * ] =
           -5 : ≠TRUE≠  (  11 * )
      INTEGER ARRAY ALPHATE [  10 :  17 ]
  ALPHATE [ * ] =
           10 :  125170908010659       160226959425530       262300840667768       245444241482839       232490829465l2
           15 :  267307802579723       44537838907482        105808360037229
      REAL    ARRAY DRIEDIM [  1 :  3 ,  1 :  3 ,  1 :  20 ]
```

Figure 14-2. Sample Dump (Sheet 2 of 3)

```
DRIEDIM [  1 ,  1 , * ] =
          1 :  2.0                    3.0                    4.0                    5.0                    6.0
          6 :  7.0                    8.0                    9.0                    1.0≠+1                 1.1≠+1
         11 :  1.2≠+1                 1.3≠+1                 1.4≠+1                 1.5≠+1                 1.6≠+1
         16 :  1.7≠+1                 1.8≠+1                 1.9≠+1                 2.0≠+1                 2.1≠+1
DRIEDIM [  1 ,  2 , * ] =
          1 : - INDEFINITE (  20 * )
DRIEDIM [  1 ,  3 , * ] =
          1 : - INDEFINITE (  20 * )
DRIEDIM [  2 ,  1 , * ] =
          1 : - INDEFINITE (  20 * )
DRIEDIM [  2 ,  2 , * ] =
          1 :  3.0                    4.0                    5.0                    6.0                    7.0
          6 :  8.0                    9.0                    1.0≠+1                 1.1≠+1                 1.2≠+1
         11 :  1.3≠+1                 1.4≠+1                 1.5≠+1                 1.6≠+1                 1.7≠+1
         16 :  1.8≠+1                 1.9≠+1                 2.0≠+1                 2.1≠+1                 2.2≠+1
DRIEDIM [  2 ,  3 , * ] =
          1 : - INDEFINITE (  20 * )
DRIEDIM [  3 ,  1 , * ] =
          1 : - INDEFINITE (  20 * )
DRIEDIM [  3 ,  2 , * ] =
          1 : - INDEFINITE (  20 * )
DRIEDIM [  3 ,  3 , * ] =
          1 :  4.0                    5.0                    6.0                    7.0                    8.0
          6 :  9.0                    1.0≠+1                 1.1≠+1                 1.2≠+1                 1.3≠+1
         11 :  1.4≠+1                 1.5≠+1                 1.6≠+1                 1.7≠+1                 1.8≠+1
         16 :  1.9≠+1                 2.0≠+1                 2.1≠+1                 2.2≠+1                 2.3≠+1
```

Figure 14-2. Sample Dump (Sheet 3 of 3)

This section contains some examples of programs that illustrate various features of ALGOL 5. It also contains some examples of typical jobs for execution of ALGOL programs.

COMPLEX SQUARE ROOT

Program COMPL (figure 15-1) computes the square root of a complex number. Although ALGOL does not support complex arithmetic directly, it can be simulated by using a pair of real numbers to represent the real and imaginary parts of a complex number. In the sample program, X and Y stand for the real and imaginary parts, respectively.

The program uses two procedures: CABS and CSQRT. CABS computes the absolute value of the number, and CSQRT computes the square root. The formula for the absolute value is:

$$CABS = \sqrt{X^2 + Y^2}$$

The equivalent method used in the procedure CABS is designed to produce more accurate results.

The square root is represented by the two numbers A and B, where A is the real part and B is the imaginary part. In order to ensure real values for A and B, one of two formulas is used, depending on the value of X. The first formula, used when X is positive or zero, is:

$$A = \pm\sqrt{\frac{X \pm CABS\ (X,Y)}{2}}, B = \frac{Y}{2A}$$

In the program, the positive root is used.

The second formula, used when X is negative, is:

$$B = \pm\sqrt{\frac{-X \pm CABS\ (X,Y)}{2}}, A = \frac{Y}{2B}$$

The positive root is used when Y is positive or zero, and the negative root is used when Y is negative. In both cases, the sign before the CABS within the radical is always taken as positive.

In the main program of COMPL, the square root of (1.0, 0.0) is computed and the result, (1.0, 0.0), is output.

CIRCUMLUDES

Figure 15-2 shows an example of a circumlude and a separately compiled main program and procedure. The program creates a histogram in the array HIST by reading values into the array DATA and computing the frequency of occurrence of each value.

The control statements needed to run the example under NOS/BE and SCOPE are shown in figures 15-3 and 15-4. The circumlude is compiled using the N option on the ALGOL5 control statement. The output from this compilation is written to the file ONE.

A library named CIRCUML is created using the appropriate library utility: EDITLIB under NOS/BE, and LIBEDT under SCOPE. The directives used with the EDITLIB and LIBEDT utilities are identical. They are shown in figure 15-5. The library contains two programs: an absolute program containing tables derived from the declarations in the circumlude, and a relocatable program containing the executable code from the circumlude.

The main program and procedure are then compiled. The N option is omitted from the ALGOL5 control statement, and the S option is included to specify the name of the library and circumlude. The output from this compilation is written to the file TWO.

When the binary file TWO is executed, the procedure HISTOGRAM is called. This procedure is defined in the circumlude. The body of the procedure is a code part with the name HISTOGRAM, which is truncated to HISTOGR. Within the code part, the procedure is named HISTG.

The control statements needed to run the program under NOS are shown in figure 15-6. The circumlude is compiled as before using the N option.

NOS control statements are used to separate the absolute program from the relocatable program, both of which are on the file ONE after compilation. They are separated by backspacing two records, then copying each record to different files. The name of the first file, the absolute part, must correspond to the name of the prelude truncated to 7 characters.

The LIBGEN utility is used to place the relocatable program, which is on file RELREC, in a library. The library is placed on the file CIRCUML.

The main program and procedure are compiled as before, by omitting the N option and specifying the S option on the ALGOL5 control statement.

The output from the second compilation is written to the file TWO, and is executed as before.

```
      1.            COMPLEX;                                                                    0  COMMENT
      2.            ≠BEGIN≠                                                                     1
      3.            ≠COMMENT≠     THIS PROGRAM COMPUTES THE SQUARE ROOT OF A COMPLEX NUMBER.     1  COMMENT
      4.                          THE COMPLEX NUMBER IS REPRESENTED AS (X,Y).                    1  COMMENT
      5.                          COPYRIGHT 1967, ASSOCIATION FOR COMPUTING MACHINERY.           1  COMMENT
      6.                          USED BY PERMISSION;                                            1
      7.                                                                                         1
      8.               ≠REAL≠ ≠PROCEDURE≠ CABS (X,Y);                                            1
      9.                  ≠COMMENT≠ CALCULATE THE ABSOLUTE VALUE OF COMPLEX NUMBER (X,Y);         1
     10.                  ≠VALUE≠ X,Y; ≠REAL≠ X,Y;                                               1
     11.                  ≠BEGIN≠                                                                2
     12.                     X := ABS(X); Y := ABS(Y);                                           2
     13.                     CABS := ≠IF≠ X = 0 ≠THEN≠ Y                                         2
     14.                        ≠ELSE≠ ≠IF≠ Y = 0 ≠THEN≠ X                                       2
     15.                        ≠ELSE≠ ≠IF≠ X > Y                                                2
     16.                           ≠THEN≠ X * SQRT(1+(Y/X)**2)                                   2
     17.                        ≠ELSE≠ Y * SQRT(1+(X/Y)**2);                                     2
     18.                  ≠END≠ CABS;                                                            1
     19.                                                                                         1
     20.               ≠PROCEDURE≠ CSQRT (X,Y,A,B);                                              1
     21.                  ≠COMMENT≠ CALCULATE THE SQUARE ROOT OF THE COMPLEX NUMBER;              1
     22.                  ≠VALUE≠ X,Y; ≠REAL≠ X,Y,A,B;                                           1
     23.                  ≠BEGIN≠                                                                2
     24.                     ≠IF≠ X = 0 ≠AND≠ Y = 0 ≠THEN≠ A := B := 0                           2
     25.                     ≠ELSE≠                                                              2
     25.                     ≠BEGIN≠                                                             3
     27.                        A := SQRT((ABS(X)+CABS(X,Y))*0.5);                               3
     28.                        ≠IF≠ X >= 0 ≠THEN≠ B := Y/(A+A)                                  3
     29.                        ≠ELSE≠                                                           3
     30.                        ≠BEGIN≠                                                          4
     31.                           B := ≠IF≠ Y < 0 ≠THEN≠ -A ≠ELSE≠ A;                           4
     32.                           A := Y/(B+B);                                                 4
     33.                        ≠END≠                                                            3
     34.                     ≠END≠                                                               2
     35.                  ≠END≠ CSQRT;                                                           1
     36.                                                                                         1
     37.            ≠BEGIN≠                                                                     2
     38.               ≠COMMENT≠ MAIN PROGRAM.....................                              2  COMMENT
     39.                           CHOOSE A VALUE FOR (X,Y) AND FIND ITS SQUARE ROOT (A,B);      2
     40.               ≠REAL≠ X,Y,A,B; ≠INTEGER≠ OUTP;                                           2
     41.                  OUTP := 61;                                                            2
     42.                  X := 1.0; Y := 0.0;                                                    2
     43.                  CSQRT(X,Y,A,B);                                                        2
     44.                  OUTPUT (OUTP,≠(≠2B,+ZD.DD,2B,+ZD.DD≠)≠, A, B);                         2
     45.            ≠END≠ MAIN;                                                                 1
     46.            ≠END≠                                                                       0
PROGRAM LENGTH      0001648 WORDS
REQUIRED  CM 036700. CP    .692 SEC.
```

Figure 15-1. Sample Program

CIRCUMLUDE:

```
    STATISTIC:
    BEGIN                                          COMMENT  OUTERMOST BLOCK ..... ;
      INTEGER OUT ;                                COMMENT  OUTPUT CHANNEL IDENTIFIER ;
      INTEGER BOTTOMBOUND,TOPBOUND ;               COMMENT  DYNAMIC ARRAY BOUND LIST ;
      OUT := 61 ;                                  COMMENT  OUTPUT CHANNEL NUMBER ;
      BOTTOMBOUND := 0 ;  TOPBOUND := 99 ;         COMMENT  DYNAMIC ARRAY BOUNDARY ;
      BEGIN                                        COMMENT  INNER BLOCK ..... ;
        INTEGER ARRAY HIST[BOTTOMBOUND:TOPBOUND] ;
        PROCEDURE HISTOGRAM (AA,LO,UP) ;
          VALUE LO,UP ;
          REAL ARRAY AA ;
          INTEGER LO,UP ;
          CODE HISTOGRAM ;
        INTEGER I,J ;
        REAL MIN,SPAN ;
        FOR I := 0 STEP 1 UNTIL 99 DO HIST[I] := -0 ;
        COMMENT  THIS SPOT IS FOR THE ACTUAL ALGOL MAIN PROGRAM (PROG SYMBOL).     ;
        PROG ;
        OUTPUT (OUT,"("/"("HISTOGRAM")",10(/,10(6ZD))")",HIST) ;
      END                                          COMMENT  INNER BLOCK ..... ;
    END                                          COMMENT OUTERMOST BLOCK ..... ;
```

MAIN PROGRAM AND PROCEDURE:

```
    HISMAIN:
    BEGIN
      INTEGER LOW,UPP ;
      INTEGER INP ;
      INP := 60 ;
      INPUT (INP,"("N,N")",LOW,UPP) ;
      BEGIN
        ARRAY DATA[LOW:UPP] ;
        INPUT (INP,"("N")",DATA) ;
        HISTOGRAM (DATA,LOW,UPP) ;
      END
    END
                                                                              EOP

    HISPROC:
    CODE HISTOGRAM ;
      PROCEDURE HISTG (A,LOWER,UPPER) ;
      VALUE LOWER,UPPER ;  INTEGER LOWER,UPPER ;
      REAL ARRAY A ;
      BEGIN
        MIN := MAXREAL ;  SPAN := -MAXREAL ;
        FOR I := LOWER STEP 1 UNTIL UPPER DO
          BEGIN
            IF A[I] < MIN THEN MIN := A[I] ;
            IF A[I] > SPAN THEN SPAN := A[I] ;
          END;
        SPAN := SPAN-MIN ;
        FOR I := LOWER STEP 1 UNTIL UPPER DO
          BEGIN
            J := (A[I]-MIN)/SPAN ;
          HIST[J] := HIST[J]+1 ;
          END
      END
```

Figure 15-2. Circumlude Example

SAMPLE JOBS

A typical deck setup for compilation and execution of an ALGOL program under NOS/BE is as follows:

job statement
ALGOL5 control statement
Execution control statement
7/8/9 card (7,8,9 multipunched in column 1)
ALGOL program
7/8/9 card
data
6/7/8/9 card (6,7,8,9 multipunched in column 1)

A typical deck setup for compilation and execution of an ALGOL program under NOS is as follows:

job statement
USER control statement
CHARGE control statement
ALGOL5 control statement
execution control statement
7/8/9 card
ALGOL program
7/8/9 card
data
6/7/8/9 card

A typical deck setup for compilation and execution of an ALGOL program under SCOPE 2 is as follows:

job statement
ACCOUNT statement
ALGOL5 control statement
execution control statement
7/8/9 card
ALGOL program
7/8/9 card
data
6/7/8/9 card

```
.
.
.
ALGOL5,N.
EDITLIB,USER.
ALGOL5,B=LGO2,S=CIRCS-STATIS.
LGO2.
.
.
.
```

Figure 15-3. NOS/BE Control Statements

```
.
.
.
ALGOL5,N.
REWIND,LGO.
LIBEDT(M)
ALGOL5,B=LGO2,S=CIRCS-STATIS.
LGO2.
.
.
.
```

Figure 15-4. SCOPE Control Statements

```
LIBRARY(CIRCUML,NEW)
SKIPB(2,ONE)
ADD(*,ONE)
FINISH.
ENDRUN.
```

Figure 15-5. NOS/BE and SCOPE Library Directives

```
.
.
.
ALGOL5(SW=80,RES,N,B=ONE)
BKSP,ONE,2.
COPYBR,ONE,STATIST.
COPYBR,ONE,RELREC.
UNLOAD(ONE)
LIBGEN(F=RELREC,P=CIRCUML,N=CIRC)
UNLOAD(RELREC)
ALGOL5(SW=80,RES,S=CIRCUML-STATIST,B=TWO)
TWO.
.
.
.
```

Figure 15-6. NOS Control Statements

STANDARD CHARACTER SETS

CONTROL DATA operating systems offer the following variations of a basic character set:

CDC 64-character set

CDC 63-character set

ASCII 64-character set

ASCII 63-character set

The set in use at a particular installation was specified when the operating system was installed.

Depending on another installation option, the system assumes an input deck has been punched either in 026 or in 029 mode (regardless of the character set in use). Under NOS/BE or SCOPE 2, the alternate mode can be specified by a 26 or 29 punched in columns 79 and 80 of the job statement or any 7/8/9 card. The specified mode remains in effect throughout the job unless it is reset by specification of the alternate mode on a subsequent 7/8/9 card.

Under NOS, the alternate mode can be specified by a 26 or 29 punched in columns 79 and 80 of any 6/7/9 card, as described above for a 7/8/9 card. In addition, 026 mode can be specified by a card with 5/7/9 multipunched in column 1, and 029 mode can be specified by a card with 5/7/9 multipunched in column 1 and a 9 punched in column 2.

Graphic character representation appearing at a terminal or printer depends on the installation character set and the terminal type. Characters shown in the CDC Graphic column of the standard character set table are applicable to BCD terminals; ASCII graphic characters are applicable to ASCII-CRT and ASCII-TTY terminals.

The table entitled ALGOL Special Symbols and Alternate Representations shows the ALGOL symbols in the form required if the RES parameter of the ALGOL control statement (section 11) is not selected. If RES is selected, the ≠ character can be omitted, and blanks delimit the symbol. No blanks can appear within the symbol in this case. The ≠ character is always required for string quotes ≠(≠ and ≠)≠, and for the alternate representation of the integer divide operator ≠/≠.

TABLE A-1. STANDARD CHARACTER SETS

Display Code (octal)	CDC			ASCII		
	Graphic	Hollerith Punch (026)	External BCD Code	Graphic Subset	Punch (029)	Code (octal)
00†	: (colon)††	8-2	00	: (colon) ††	8-2	072
01	A	12-1	61	A	12-1	101
02	B	12-2	62	B	12-2	102
03	C	12-3	63	C	12-3	103
04	D	12-4	64	D	12-4	104
05	E	12-5	65	E	12-5	105
06	F	12-6	66	F	12-6	106
07	G	12-7	67	G	12-7	107
10	H	12-8	70	H	12-8	110
11	I	12-9	71	I	12-9	111
12	J	11-1	41	J	11-1	112
13	K	11-2	42	K	11-2	113
14	L	11-3	43	L	11-3	114
15	M	11-4	44	M	11-4	115
16	N	11-5	45	N	11-5	116
17	O	11-6	46	O	11-6	117
20	P	11-7	47	P	11-7	120
21	Q	11-8	50	Q	11-8	121
22	R	11-9	51	R	11-9	122
23	S	0-2	22	S	0-2	123
24	T	0-3	23	T	0-3	124
25	U	0-4	24	U	0-4	125
26	V	0-5	25	V	0-5	126
27	W	0-6	26	W	0-6	127
30	X	0-7	27	X	0-7	130
31	Y	0-8	30	Y	0-8	131
32	Z	0-9	31	Z	0-9	132
33	0	0	12	0	0	060
34	1	1	01	1	1	061
35	2	2	02	2	2	062
36	3	3	03	3	3	063
37	4	4	04	4	4	064
40	5	5	05	5	5	065
41	6	6	06	6	6	066
42	7	7	07	7	7	067
43	8	8	10	8	8	070
44	9	9	11	9	9	071
45	+	12	60	+	12-8-6	053
46	-	11	40	-	11	055
47	*	11-8-4	54	*	11-8-4	052
50	/	0-1	21	/	0-1	057
51	(	0-8-4	34	(	12-8-5	050
52	)	12-8-4	74	)	11-8-5	051
53	$	11-8-3	53	$	11-8-3	044
54	=	8-3	13	=	8-6	075
55	blank	no punch	20	blank	no punch	040
56	, (comma)	0-8-3	33	, (comma)	0-8-3	054
57	. (period)	12-8-3	73	. (period)	12-8-3	056
60	≡	0-8-6	36	#	8-3	043
61	[	8-7	17	[	12-8-2	133
62	]	0-8-2	32	]	11-8-2	135
63	% ††	8-6	16	% ††	0-8-4	045
64	≠	8-4	14	" (quote)	8-7	042
65	←	0-8-5	35	_ (underline)	0-8-5	137
66	∨	11-0 or 11-8-2†††	52	!	12-8-7 or 11-0†††	041
67	∧	0-8-7	37	&	12	046
70	↑	11-8-5	55	' (apostrophe)	8-5	047
71	↓	11-8-6	56	?	0-8-7	077
72	<	12-0 or 12-8-2†††	72	<	12-8-4 or 12-0†††	074
73	>	11-8-7	57	>	0-8-6	076
74	≤	8-5	15	@	8-4	100
75	≥	12-8-5	75	\	0-8-2	134
76	¬	12-8-6	76	~ (circumflex)	11-8-7	136
77	; (semicolon)	12-8-7	77	; (semicolon)	11-8-6	073

†Twelve zero bits at the end of a 60-bit word in a zero byte record are an end of record mark rather than two colons.

††In installations using a 63-graphic set, display code 00 has no associated graphic or card code; display code 63 is the colon (8-2 punch). The % graphic and related card codes do not exist and translations yield a blank (55_8).

†††The alternate Hollerith (026) and ASCII (029) punches are accepted for input only.

TABLE A-2. ALGOL SPECIAL SYMBOLS AND ALTERNATE REPRESENTATIONS

CDC Graphic	ASCII Subset Graphic	Alternate Representations (using CDC graphics)	CDC Graphic	ASCII Subset Graphic
+	+		≠FALSE≠	"FALSE"
-	-		≠FOR≠	"FOR"
*	*		≠GOTO≠	"GOTO"
/	/		≠IF≠	"IF"
//	//	≠/≠†, ≠DIV≠	≠INTEGER≠	"INTEGER"
**	**	≠POWER≠, ↑	≠LABEL≠	"LABEL"
<	<	≠LESS≠, ≠LT≠	≠OWN≠	"OWN"
⩽	@	≠NOT GREATER≠ , ≠LE≠<=	≠PROCEDURE≠	"PROCEDURE"
=	=	≠EQUAL≠ , ≠EQ≠	≠REAL≠	"REAL"
¬=	^=	≠NOT EQUAL≠ , ≠NE≠	≠STEP≠	"STEP"
⩾	\	≠NOT LESS≠ , ≠GE≠	≠STRING≠	"STRING"
>	>	≠GREATER≠ , ≠GT≠>=	≠SWITCH≠	"SWITCH"
∧	&	≠AND≠	≠THEN≠	"THEN"
∨	!	≠OR≠	≠TRUE≠	"TRUE"
≡	#	≠EQV≠††	≠UNTIL≠	"UNTIL"
¬	^ (circumflex)	≠NOT≠	≠VALUE≠	"VALUE"
→	_ (underline)	≠IMPL≠	≠WHILE≠	"WHILE"
,	,		≠ALGOL≠	"ALGOL"
:	:	. .	≠CODE≠	"CODE"
;	;	. ,	≠CHECKOFF≠	"CHECKOFF"
≠	"		≠CHECKON≠	"CHECKON"
(	(		≠EJECT≠	"EJECT"
)	)		≠EOP≠	"EOP"
[	[	(/	≠FORMAT≠	"FORMAT"
]	]	/)	≠FORTRAN≠	"FORTRAN"
:=	:=	.= or . .=	≠INCLUDE≠	"INCLUDE"
≠(≠	"("		≠LIST≠	"LIST"
≠)≠	")"		≠NOLIST≠	"NOLIST"
≠ARRAY≠	"ARRAY"		≠OBJLIST≠	"OBJLIST"
≠BEGIN≠	"BEGIN"		≠OBJNOLIST≠	"OBJNOLIST"
≠BOOLEAN≠	"BOOLEAN"		≠PROG≠	"PROG"
≠COMMENT≠	"COMMENT"		≠RJ≠	"RJ"
≠DO≠	"DO"		≠SIMPLE≠	"SIMPLE"
≠ELSE≠	"ELSE"		≠VARIABLE≠	"VARIABLE"
≠END≠	"END"			

† ≠ character required for ≠/≠, ≠(≠, and ≠)≠ in reserved word mode.
†† Former version ≠EQUIV≠ is no longer supported.

Two types of diagnostics are issued by ALGOL:

- Compile time diagnostics are issued by the ALGOL compiler whenever an incorrect or suspicious usage is detected during compilation.
- Execution time diagnostics are issued by the ALGOL execution time system whenever an invalid action is requested or takes place during execution.

COMPILE TIME DIAGNOSTICS

Compile time diagnostics are dynamically constructed by the compiler when an error condition occurs during compilation. The compiler tries to recover from the error condition by making an assumption about what was intended instead of the incorrect usage. Because these assumptions do not necessarily reflect the user's actual intentions, recovery from one error might lead to other errors further along in the program. For example, an incorrectly spelled identifier in a declaration means that the identifier is not declared in the block containing the declaration; each remaining use of that identifier in the block results in a diagnostic stating that the identifier is not declared.

In other cases, a mispunched character can cause the compiler to prematurely terminate a statement by substituting a semicolon for the character. The remainder of the original statement is then read as the beginning of a new statement, often producing further diagnostics.

In general, if the user receives multiple diagnostics in a program, the first line in error should be checked to see if a mistake on that line ultimately resulted in other diagnostics as well. Since the compiler attempts to be as explicit as possible in the wording of diagnostics, it is usually possible to correct the error on the next recompilation.

Each diagnostic consists of three asterisks, the line number where the error was detected (which could be later than the actual line in error), a letter indicating severity level (T, W, F, or C) and the text of the diagnostic. In addition, header lines are listed before groups of diagnostics, indicating the stage of compilation during which the error was detected.

The severity level of a diagnostic is one of the following:

- T (trivial). In this case, the program is syntactically or semantically correct, but a suspicious usage has occurred. An example is an incorrect specification of a comment directive in which the directive is invalid, but the syntax of the comment string is still correct. Another example is the message "SCAN RESUMED" which informs the user that the compiler has recovered from a previous error and is attempting to continue compilation of the program.
- W (warning). In this case, an incorrect usage has occurred, but by making an assumption about what was intended, the compiler is able to continue. An example is a real constant out of range; the constant is replaced with MAXREAL, and compilation continues. For both T and W level diagnostics, the binary produced is executable but possibly incorrect.
- F (fatal). In this case, an error has occurred that prevents the compiler from compiling the statement in which the error occurred. An example is use of an identifier that has not been declared. When any fatal errors have occurred, the binary produced from compilation is not executable.
- C (catastrophic). In this case, the compiler cannot continue compiling the program unit in which the error occurred, and advances to the next program unit. An example is symbol table overflow, in which too many identifiers have been declared for the symbol table to be kept within the field length.

Diagnostics can be issued during any of five phases of compilation. If any errors are detected during phase x, the following header is issued before the diagnostics detected during that phase:

ERRORS DETECTED DURING x

In addition, if errors have already been found in previous phases, the following message is issued:

(NOTE: ERRORS MAY BE PROPAGATED FROM THE PREVIOUS SCAN)

The five phases are as follows:

- Lexical scan. During this phase, the characters constituting the program are grouped into the appropriate syntactic entities. Errors detected include invalid placement of delimiters or comments.
- Syntactical scan. During this phase, the symbols are checked to see that they are correctly grouped into statements. Errors detected during this phase include badly formed expressions and statements.
- Semantic scan. During this phase, the scope of identifiers is determined and calling sequences for procedures are set up. Errors detected include failure to declare identifiers and invalid correspondence between actual and formal parameters.
- Code generation. During this phase, the actual relocatable code is generated. If fatal errors have already occurred, code generation does not take place. Errors detected include invalid actual/formal parameter correspondence (for cases that have not been detected during the semantic scan), and some types of format string errors.

- Optimization. During this phase, the code already generated is rewritten in some cases to improve its performance. All errors detected during this phase are catastrophic and likely to be the result of internal compiler malfunction. If the user receives an error during this phase, a CDC analyst should be notified.

An example of an incorrect program and the error messages it causes is shown in figure B-1.

EXECUTION TIME DIAGNOSTICS

Errors detected during execution include the use of arithmetically invalid values such as infinite and indefinite, type or kind mismatch between actual and formal parameters (in the case where this cannot be detected during compilation, such as with a separately compiled procedure), and badly formatted input data. All execution time errors are fatal; if the user has called ERROR, CHANERROR, or ARTHOFLW with the appropriate parameters, recovery to a user-defined label is possible. Otherwise, the program terminates after issuing the diagnostic.

Execution time diagnostics, like compile time diagnostics, are dynamically constructed with appropriate information added to the text of the diagnostic when the error occurs.

For most errors detected by the operating system, the ALGOL execution time system regains control after the error and outputs its own diagnostic message, together with a post-mortem dump; this information is placed into the file specified by the E parameter on the execution control statement. An example of a program producing an execution time error is shown in figure B-2, along with the messages written to the file OUTPUT and the dayfile. In addition to an error message, a traceback (unless D=0 was specified on the execution control statement) and possibly a dump are also output. The formats of these listings are described in section 14.

```
      1.          SCHLEMAZEL;                                                          0  COMMENT
      2.          ≠COMMENT≠ INTENDED TO PRODUCE DIAGNOSTICS;                           0  COMMENT
      3.          ≠COMMENT≠ ≠INCLUDE IS MISSING A NOT-EQUAL;                           0  COMMENT
      4.          ≠BEGIN≠                                                              1
      5.             ≠REAL≠ ≠PROCEDURE≠ LIFE (CHAOS,ARRAY); ≠REAL≠ CHAOS;              1
      6.                ≠REAL≠ ≠ARRAY≠ ARRAY;                                         1
      7.                ≠BEGIN≠                                                       2
      8.                ARRAY[3,4] := CHAOS**2;                                       2
      9.                LIFE := CHAOS;                                                2
     10.             ≠END≠                                                           1
     11.             ≠INTEGER≠ I;                                                    1
     12.             ≠REAL≠ ≠ARRAY≠ HOPELESS [1:2,3:4];                              1
     13.             ≠FOR≠ I := 10,15,20 ≠DO≠                                        1
     14.             OUTPUT (61,≠(       ≠N≠)≠,I);                                   1
     15.             OUTPUT (61, ≠(≠ZD,,ZD≠)≠, LIFE (I,HOPELESS);                    1
     16.          ≠END≠                                                              0

ERRORS DETECTED DURING THE LEXICAL SCAN

***      2 W     DELIMITER BEFORE START OF PROGRAM
***      3 W     DELIMITER BEFORE START OF PROGRAM
***      3 T     - FOUND TERMINATING COMPOUND DELIMITER, QUOTE INSERTED BEFORE IT
***     11 W     DELIMITER IN COMMENT
***     14 F     ≠ ILLEGAL IN THIS CONTEXT, DELETED
***     14 F     ≠ ILLEGAL IN THIS CONTEXT, DELETED

ERRORS DETECTED DURING THE SYNTACTICAL SCAN       (NOTE: ERRORS MAY BE PROPAGATED FROM THE PREVIOUS SCAN)

***     14 F     INCORRECT OPERAND FOR MONADIC OPERATOR
                     ) , IS DELETED
***     14 F     ILLEGAL DELIMITER IN ACTUAL PARAMETER LIST
                     ;  OUTPUT (               61 IS DELETED
***     15 T     SCAN RESUMED
***     15 F     ILLEGAL DELIMITER IN ACTUAL PARAMETER LIST
                     ; ≠END≠ IS DELETED
***     16 T     SCAN RESUMED
***     16 F     PRELIMINARY END OF SOURCE TEXT REACHED

ERRORS DETECTED DURING THE SEMANTIC SCAN          (NOTE: ERRORS MAY BE PROPAGATED FROM THE PREVIOUS SCAN)

***     14 F     OPERATOR ≠N≠ HAS NOT BEEN DECLARED
***     14 F     IDENTIFIER I HAS NOT BEEN DECLARED
***     15 F     FORMAL/ACTUAL TYPE MISMATCH IN CALL OF  LIFE PARAMETER NUMBER    1
***     15 F     IDENTIFIER I HAS NOT BEEN DECLARED
REQUIRED  CM 035500. CP   .159 SEC.  BAD BINARY
```

Figure B-1. Compile Time Diagnostic Example

PROGRAM:

```
        1.              SCHLEMIEL;
        2.              #BEGIN#
        3.              #COMMENT# PRODUCES EXECUTION TIME ERRORS;
        4.                 #REAL# X,Y,Z;
        5.                    X := 0;    Y := 0;     Z := X/Y;
        6.                    X := Z + Y;
        7.                    OUTPUT (61,#(#5D.5D#)#,Z)
        8.              #END#
   PROGRAM LENGTH       000030B WORDS
   REQUIRED  CM 036500. CP   .223 SEC.
```

MESSAGES ON OUTPUT:

```
   MODE ERROR 04        FOUND BY ALGOL5
   ARITHMETIC USE OF INDEFINITE VALUE


   POST-MORTEM DUMP STARTED

   PM-DUMP CALLED AT LINE    7 IN PROGRAM   SCHLEMI DECLARED AT LINE   0 IN MODULE SCHLEMI
```

NOS/BE DAYFILE:

```
     MFS  NM1-  CYB74-SN108    5C/P0B     11/14/78
   10.00.51.ALEXD5T  FROM     /KQ
   10.00.51.IP  00000192 WORDS - FILE INPUT  , DC 04
   10.00.51.ALEXD,P5,T10.
   10.00.54.REWIND,OUTPUT.
   10.00.54.MAP,OFF.
   10.00.54.ALGOL5,EL=T.
   10.00.57.  -SCHLEMI- NO ERRORS DETECTED
   10.00.57. CM 036500. CP   .223 SEC.
   10.00.58.LGO.
   10.01.01. ALGOL 5.0 78318
   10.01.01.MODE ERROR
   10.01.01.JOB REPRIEVED
   10.01.01.MODE ERROR 04        FOUND BY ALGOL5
   10.01.01. CM 024300. CP   0.036 SEC.
   10.01.01.EXECUTION ERROR, JOB ABORTED
   10.01.01.OP  00000320 WORDS - FILE OUTPUT , DC 40
   10.01.01.MS    3584  WORDS  (   25088 MAX USED)
   10.01.01.CPA        .832 SEC.        .832 ADJ.
   10.01.01.CPP        .229 SEC.        .229 ADJ.
   10.01.01.IO        1.011 SEC.       1.011 ADJ.
   10.01.01.CM       31.652 KWS.       1.931 ADJ.
   10.01.01.SS                         4.005
   10.01.01.PP        5.520 SEC.     DATE 12/04/78
   10.01.01.EJ  END OF JOB, KQ


   **********          I014GFB  //// END OF LIST ////
   **********          I014GFB  //// END OF LIST ////
```

Figure B-2. Execution Error Example

GLOSSARY C

This glossary does not include terms defined in either of the ALGOL standards (see preface).

BEGINNING-OF-INFORMATION (BOI) -
Record Manager defines beginning-of-information as the start of the first user record in a file. System-supplied information, such as an index block or control word, does not affect beginning-of-information. Any label on a tape exists prior to beginning-of-information.

BINARY SEQUENTIAL -
An ALGOL file type consisting of a sequential file in which data is in the same format as in memory.

BLOCK -
In the context of input/output, a physical grouping of data on a file that provides faster data transfer. Record Manager defines four block types on sequential files: I, C, K, and E. Blocks are not defined for word addressable files.

CALLING SEQUENCE -
Conventions established for linkage between subprograms, especially as to where parameters are located on entry and returned on exit.

CHANNEL -
Synonymous with file. Represented in the ALGOL program by a channel number, which is linked with the file through the CHANNEL procedure or by default.

CIRCUMLUDE -
Declarations and executable code logically forming an outer block to a program, but compiled separately.

CODED SEQUENTIAL -
An ALGOL file type consisting of a sequential file in which data is converted to coded format. Conversion takes place either according to a format string or by standard format.

COMMENT DIRECTIVE -
A series of symbols occurring in a comment, governing options such as object and source code listing, array bounds checking, and omission of portions of source code. The directive is either honored or ignored, depending on a control statement option.

COMMON BLOCK -
An area of memory that can be declared by more than one relocatable program and used for storage of shared data.

COMPILE TIME -
The period during which a program is being compiled. Contrast with execution time.

END-OF-INFORMATION (EOI) -
Record Manager defines end-of-information on a sequential file in terms of file residence. See table C-1.

TABLE C-1. END-OF-INFORMATION

File Residence	Physical Position
Mass storage	After last user record.
Labeled tape in SI,I,X,S,L format	After last user record and before any file trailer labels.
Unlabeled tape in SI,I,X format	After last user record and before any file trailer labels.
Unlabeled tape in S or L format	Undefined.

ENTRY POINT -
A location within a program unit that can be referenced from other program units. Each entry point has a unique name.

EXECUTION CONTROL STATEMENT -
The control statement used to execute, or load and execute, the binary output from an ALGOL compilation. The file name and control statement LGO are frequently used for this purpose.

EXECUTION TIME -
The period during which a program is executing. Contrast with compile time.

EXTERNAL IDENTIFIER -
An identifier, limited to seven characters, used as a module name by the loader and for separately compiled procedures.

EXTERNAL REFERENCE -
A reference in one program unit to an entry point in another program unit.

FIELD LENGTH -
The number of memory words assigned to a job.

FILE -
A logically related set of information; the largest collection of information that can be addressed by a file name. Starts at beginning-of-information and ends at end-of-information.

FILE CONTROL STATEMENT -
A control statement, processed by Record Manager, that contains parameters used to build the file information table.

FILE INFORMATION TABLE -
A table through which a user program communicates with Record Manager. All file processing executes on the basis of fields in the table.

LOGICAL FILE NAME -
The name by which a file is identified; consists of one to seven letters and digits, the first a letter.

LOGICAL OPERATION -
An operation performed on one or two Boolean operands and yielding a Boolean value. The operation performed on one operand is ¬ (negation). The operations performed on two operands are ∧ (logical and), ∨ (logical or), → (implication), and ≡ (equivalence).

OBJECT LISTING -
A compiler-generated listing of the object code produced for a program, represented in COMPASS code.

PAGED FILE -
A file in which ALGOL has inserted carriage control characters, making it in valid format for printing.

PARTITION -
Record Manager defines a partition as a division within a file with sequential organization. Generally, a partition contains several records or sections. Implementation of a partition boundary is affected by file structure and residence. See table C-2.

Notice that in a file with W type records a short PRU of level 0 terminates both a section and a partition.

POSTLUDE -
The portion of a circumlude consisting of code to be executed after the program represented by the ≠PROG≠ symbol.

PRELUDE -
The portion of a circumlude consisting of declarations and code to be executed before the program represented by the ≠PROG≠ symbol.

RECORD -
Record Manager defines a record as a group of related characters. A record or a portion thereof is the smallest collection of information passed between Record Manager and a user program. Eight different record types exist, as defined by the RT field of the file information table.

RELATION -
A comparison of two arithmetic expressions by one of the operators = (equal), ¬ = (not equal), < (less than), ≤ (less than or equal to), > (greater than, ≥ (greater than or equal to). A relation yields the Boolean value ≠TRUE≠ or ≠FALSE≠.

RELOCATION -
Placement of object code into central memory in locations that are not predetermined and adjusting the addresses accordingly.

SECTION -
Record Manager defines a section as a division within a file with sequential organization. Generally, a section contains more than one record and is a division within a partition of a file. A section terminates with a physical representation of a section boundary. See table C-3.

The NOS and NOS/BE operating systems equate a section with a system-logical-record of level 0 through 16 octal.

TABLE C-2. PARTITION BOUNDARIES

Device	RT	BT	Physical Representatation
PRU device	W	I	A short PRU of level 0 containing one-word deleted record pointing back to last I block boundary, followed by a control word with flag indicating partition boundary.
	W	C	A short PRU of level 0 containing word with a flag indicating partition boundary.
	D,F,R, S,T,U, Z	C	A short PRU of level 0 followed by a zero-length PRU of level 17.
S or L format tape	W	I	Separate tape block containing as many deleted records of record length 0 as required to exceed noise record size, followed by a deleted one-word record pointing back to the last I block boundary, followed by a control word with a flag indicating a partition boundary.
	W	C,E	Separate tape block containing as many deleted records of record length 0 as required to exceed noise record size, followed by a control word with a flag indicating a partition boundary.
	D,F,T, R,S,U, Z	C,K,E	Tapemark.
Any other tape format			Undefined.

SEQUENTIAL -
A file organization in which the location of each record is defined only as occurring immediately after the preceding record. A file position is defined at all times, which specifies where the next record is to be read or written.

SEPARATELY COMPILED PROCEDURE -
A procedure, written in ALGOL, COMPASS, FORTRAN, or SYMPL, and compiled or assembled separately from the ALGOL program in which it is called.

SIMPLE INPUT/OUTPUT -
The standard procedures that perform input/output on coded sequential files, other than INLIST and OUTLIST.

TABLE C-3. SECTION BOUNDARIES

Device	RT	BT	Physical Representatation
PRU device	W	I	Deleted one-word record pointing back to last I block boundary followed by a control word with flags indicating a section boundary. At least the control word is in a short PRU of level 0.
	W	C	Control word with flags indicating a section boundary. The control word is in a short PRU of level 0.
	D,F,R, T,U,Z,	C	Short PRU with level less than 17 octal.
	S	Any	Undefined.
S or L format tape	W	I	A separate tape block containing as many records of record length 0 as required to exceed noise record size, followed by a deleted one-word record pointing back to the last I block boundary, followed by a control word with flags indicating a section boundary.
	W	C,E	A separate tape block containing as many deleted records of record length 0 as required to exceed noise record size, followed by a control word with flags indicating a section boundary.
	D,F,R, T,U,Z,	C,K,E	Undefined.
	S	Any	Undefined.
Any other tape format			Undefined.

SOURCE LISTING -
A compiler-produced listing, in a particular format, of the user's original source program.

STACK -
The portion of the execution time field length used for allocation of currently active variables, arrays, and other program data entities.

STANDARD CIRCUMLUDE -
The circumlude provided by the ALGOL system by default. It contains definitions of the standard procedures as well as system-defined variables.

STATE DIAGRAM -
A method of indicating the syntax of a construct in a form similar to a flowchart. See Notations Used in this Manual.

SYMBOL -
An indivisible component of an ALGOL program. Symbols include letters, digits, special characters (;, =, and so forth) and special symbols delimited by the character (≠BEGIN≠, ≠ELSE≠, and so forth).

SYMBOLIC DUMP -
A printout, in a specific format, displaying the names and values of program entities such as variables and arrays.

SYSTEM-LOGICAL-RECORD -
Under NOS/BE, a data grouping that consists of one or more PRUs terminated by a short or zero-length PRU. These records can be transferred between devices without loss of structure.

UNPAGED FILE -
A coded sequential file in which no carriage control characters are inserted by ALGOL.

WORD ADDRESSABLE -
A file organization in which the location of each record is defined by the ordinal of the first word in the record, relative to the beginning of the file.

SYNTAX SUMMARY D

This appendix defines the syntax of ALGOL entities in a notation known as Backus Normal Form (BNF). In this notation, entities are defined in terms of other entities in the following format:

<a> ::= <b> <c>

This notation means that the entity <a> consists of the entity <b> followed by the entity <c>. An entity bracketed by < and > is defined further elsewhere in the syntax. Anything not appearing between < and > stands for itself. For example:

<a> ::= <d> ≠BEGIN≠

If one entity can be defined by more than one sequence, the sequences are separated by vertical lines:

<a> ::= <b> <c> | <d> ≠BEGIN≠

In this example, the entity <a> consists of either <b> followed by <c>, or <d> followed by the characters ≠BEGIN≠.

The entity being defined can also appear in the definition, in which case the definition is recursive. For example:

<a> ::= <b> <c> | <d> ≠BEGIN≠ | <a><e>

This definition means that <a> consists of either <b> followed by <c>, or <d> followed by ≠BEGIN≠, followed in either case by any number (including none) of occurrences of <e>.

The representation of syntax in this appendix does not correspond precisely to the representation of syntax in the main text of the manual, but the ultimate syntax of an ALGOL program is the same in both cases.

<empty> ::=

<basic symbol> ::= <letter> | <digit> | <logical value> | <special symbol> | <delimiter>

<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

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

<logical value> ::= ≠TRUE≠ | ≠FALSE≠

<special symbol> ::= <any symbol in CDC 64-character set>

<delimiter> ::= <operator> | <separator> | <bracket> | <declarator> | <specificator>

<operator> ::= <arithmetic operator> | <relational operator> | <logical operator> | <sequential operator>

<arithmetic operator> ::= + | - | * | / | // | ** | ↑

<relational operator> ::= < | ≤ | = | ≥ | > | ¬=

<logical operator> ::= ≡ | → | ∧ | ∨ | ¬

<sequential operator> ::= ≠GO TO≠ | ≠IF≠ | ≠THEN≠ | ≠ELSE≠ | ≠FOR≠ | ≠DO≠

<separator> ::= ≠ | , | . | : | ; | := | ≠STEP≠ | ≠UNTIL≠ | ≠WHILE≠ | ≠COMMENT≠ | ≠CODE≠ | ≠ALGOL≠ | ≠FORTRAN≠ | ≠RJ≠

<bracket> ::=) | (|] | [| ≠(≠ | ≠)≠ | ≠BEGIN≠ | ≠END≠

<declarator> ::= ≠OWN≠ | ≠BOOLEAN≠ | ≠INTEGER≠ | ≠REAL≠ | ≠ARRAY≠ | ≠SWITCH≠ | ≠PROCEDURE≠

<specificator> ::= ≠STRING≠ | ≠LABEL≠ | ≠VALUE≠ | ≠VARIABLE≠ | ≠SIMPLE≠ | ≠FORMAT≠ | ≠LIST≠

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

<ld> ::= <letter> | <digit>

<tail> ::= <ld> | <ld><ld> | <ld><ld><ld> | <ld><ld><ld><ld> | <ld><ld><ld><ld><ld> | <ld><ld><ld><ld><ld><ld>

<external identifier> ::= <letter> | <letter><tail>

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

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

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

<exponent part> ::= ≠ <integer>

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

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

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

<proper string> ::= <any sequence of characters not containing ≠(≠ or ≠)≠> | <empty>

<open string> ::= <proper string> | <proper string>≠(≠<open string>≠)≠<open string>

<string> ::= ≠(≠<open string>≠)≠ | ≠ (≠<open string>≠)≠<string>

CONSTITUENTS OF EXPRESSIONS

<variable identifier> ::= <identifier>

<simple variable> ::= <variable identifier>

<subscript expression> ::= <arithmetic expression>

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

<array identifier> ::= <identifier>

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

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

<procedure identifier> ::= <identifier>

<actual parameter> ::= <string> | <expression> | <array identifier> | <switch identifier> | <procedure identifier>

<letter string> ::= <letter> | <letter string><letter>

<parameter delimiter> ::= , |)<letter string>:(

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

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

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

EXPRESSIONS

<expression> ::= <arithmetic expression> | <Boolean expression> | <designational expression>

<adding operator> ::= + | -

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

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

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

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

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

<if clause> ::= ≠IF≠<Boolean expression>≠THEN≠

<arithmetic expression> ::= <simple arithmetic> | <if clause><simple arithmetic>≠ELSE≠<arithmetic expression>

<relational operator> ::= < | ≤ | = | ≥ | > | ¬=

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

<Boolean primary> ::= <logical value> | <variable> | <function designator> | <relation> | (<Boolean expression>)

<Boolean secondary> ::= <Boolean primary> |¬<Boolean primary>

<Boolean factor> ::= <Boolean secondary> | <Boolean factor> ∧ <Boolean secondary>

<Boolean term> ::= <Boolean factor> | <Boolean term> ∨ <Boolean factor>

<implication> ::= <Boolean term> | <implication>→<Boolean term>

<simple Boolean> ::= <implication> | <simple Boolean> ≡ <implication>

<if clause> ::= ≠IF≠<Boolean expression>≠THEN≠

<Boolean expression> ::= <simple Boolean> | <if clause> <simple Boolean>≠ELSE≠<Boolean expression>

<label> ::= <identifier>

<switch identifier> ::= <identifier>

<switch designator> ::= <switch identifier>[<subscript expression>]

<subscript expression> ::= <arithmetic expression>

<simple designational> ::= <label> | <switch designator> | (<designational expression>)

<designational expression> ::= <simple designational> | <if clause><simple designational>≠ELSE≠<designational expression>

COMPOUND STATEMENTS AND BLOCKS

<compound tail> ::= <statement>≠END≠ | <statement>;<compound tail>

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

<unlabeled compound> ::= ≠BEGIN≠<compound tail>

<unlabeled block> ::= <block head>;<compound tail>

<compound statement> ::= <unlabeled compound> | <label>:<compound statement>

<block> ::= <unlabeled block> | <label>:<block>

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

STATEMENTS AND BASIC STATEMENTS

<unlabelled basic statement> ::= <assignment statement> | <go to statement> | <dummy statement> | <procedure statement>

<basic statement> ::= <unlabelled basic statement> | <label>:<basic statement>

<destination> ::= <variable> | <procedure identifier>

<left part> ::= <destination>:=

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

<assignment statement> ::= <left part list><arithmetic expression> | <left part list><Boolean expression>

<go to statement> ::= ≠GO TO≠<designational expression>

<dummy statement> ::= <empty>

<procedure identifier> ::= <identifier>

<actual parameter> ::= <string> | <expression> | <array identifier> | <switch identifier> | <procedure identifier>

<letter string> ::= <letter> | <letter string><letter>

<parameter delimiter> ::= , |)<letter string>:(

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

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

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

<statement> ::= <unconditional statement> | <conditional statement> | <for statement>

<for list element> ::= <arithmetic expression> | <arithmetic expression>≠STEP≠<arithmetic expression>≠UNTIL≠ <arithmetic expression> | <arithmetic expression> ≠WHILE≠<Boolean expression>

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

<for clause> ::= ≠FOR≠<variable identifier>:=<for list>≠DO≠

<for statement> ::= <for clause><statement> | <label>:<for statement>

<if clause> ::= ≠IF≠<Boolean expression>≠THEN≠

<unconditional statement> ::= <basic statement> | <compound statement> | <block>

<if statement> ::= <if clause><unconditional statement>

<conditional statement> ::= <if statement> | <if statement>≠ELSE≠<statement> | <if clause><for statement> | <label>:<conditional statement>

DECLARATIONS

<declaration> ::= <type declaration> | <array declaration> | <switch declaration> | <procedure declaration>

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

<type> ::= ≠REAL≠ | ≠INTEGER≠ | ≠BOOLEAN≠

<local or own> ::= <empty> | ≠OWN≠

<type declaration> ::= <local or own><type><type list>

<lower bound> ::= <arithmetic expression>

<upper bound> ::= <arithmetic expression>

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

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

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

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

<array declarer> ::= <type>≠ARRAY≠ | ≠ARRAY≠

<array declaration> ::= <local or own><array declarer> <array list>

<switch list> ::= <designational expression> | <switch list> <designational expression>

<switch declaration> ::= ≠SWITCH≠<switch identifier>:=<switch list>

<formal parameter> ::= <identifier>

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

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

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

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

<separate specifier> ::= ≠STRING≠ | <type> | <array declarer> | ≠LABEL≠ | ≠SWITCH≠ | ≠PROCEDURE≠ | <type>≠PROCEDURE≠ | ≠VARIABLE≠ | ≠SIMPLE≠ | ≠FORMAT≠ | ≠LIST≠

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

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

<code number> ::= <digit> | <digit><digit> | <digit><digit> <digit> | <digit><digit><digit><digit> | <digit><digit> <digit><digit><digit>

<code identifier> ::= <empty> | <code number> | <external identifier>

<code specifier> ::= ≠RJ≠ | <empty>

<code> ::= ≠CODE≠<code specifier><code identifier> | ≠ALGOL≠<code specifier><code identifier> | ≠FORTRAN≠<code identifier>

<separate procedure body> ::= <code>

<separate procedure declaration> ::= ≠PROCEDURE≠ <separate procedure heading><separate procedure body> | <type>≠PROCEDURE≠ <separate procedure heading><separate procedure body>

<specifier> ::= ≠STRING≠ | <type> | <array declarer> | ≠LABEL≠ | ≠SWITCH≠ | ≠PROCEDURE≠ | <type>≠PROCEDURE≠

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

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

<procedure body> ::= <statement>

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

MEMORY LAYOUT

The memory layout allows dynamic field length adjustment and file creation during program execution.

Among the components maintained in memory are the stack and the heap. The stack starts at some address above the program and grows to higher addresses. It contains all entries of which the size can be determined at compile time and of which the lifetime depends only on the block structure of the program as well as arrays that possess dynamic bounds and format strings that cannot be converted at compile time. The heap is allocated in the area between the top of the stack and the highest address (FL) available to the job step. The heap can grow in both directions and contains all entries of which the lifetime does not fully depend on the block structure of the program.

The heap can be moved and obsolete entries discarded whenever memory requirements make it necessary. When the heap is moved, all references to entries in the heap are adjusted accordingly.

The memory layout is depicted in figure E-1.

ALLOCATION OF VARIABLES AND STACK LAYOUT

For simple variables one word is allocated.

For arrays three entries are allocated separately:

Array descriptor	One word
Dope vector	One word plus one word for each dimension
Array elements	One word for each element

All entries are allocated in the stack or the labeled common block associated with the module generated.

In the general case, the stack is allocated dynamically in blank common. Entries in the outer program level in the stack are accessed directly in blank common. Efficiency of access is optimal for common blocks. Optimization of allocation is shown in table E-1.

TABLE E-1. OPTIMAL ALLOCATION

Type of Entry	Normal Allocation	Condition for Optimization	Optimized Allocation
Simple variable	Stack	Own or declared in prelude	Common
Array descriptor	Stack	Bounds are known at compile time	Not allocated
		Own	In code
		Declared in prelude	Common
Dope vector	Stack	Bounds are known at compile time	Common
		Array is used only locally	Not allocated
		Declared in prelude	Common
Elements of array	Stack	Bounds are known at compile time	Stack
		Own array or declared in prelude	Common

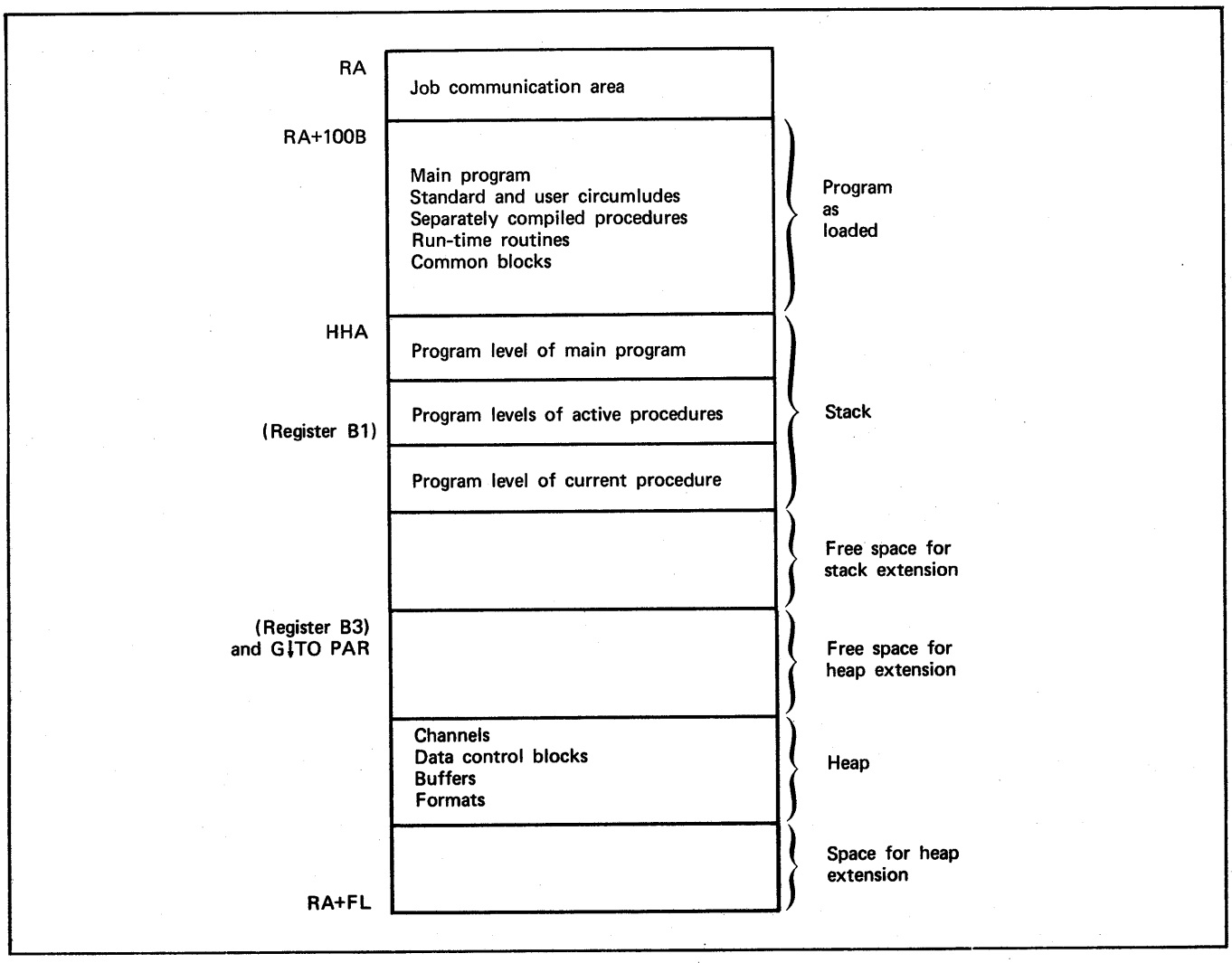


Figure E-1. Memory Layout of Program

The stack is organized in program levels: one for the main program and one for each activation of a procedure. The layout of a program level in the stack is shown in figure E-2. The program level contains return information, a block section for each active block, the static working stack, and dynamic arrays if any exist. The size of the program level is determined at compile-time, except for dynamic arrays.

A block section contains a block header and the variables declared in the block. The lowest address of dynamic arrays in the stack relative to the top of the stack is stored in the block header. The block header is required when the block contains arrays with dynamic bounds or when labels defined in the block are referenced from outside the block. The block header is suppressed if not needed. Simple variables occupy one word in the block section. For declarations of arrays with dynamic bounds the block section in the stack contains an array descriptor for each array identifier and a common dope vector of n+1 words, where n is the dimension of the arrays being declared. Declarations of arrays with static bounds are represented in the stack by an array descriptor only for those arrays that appear as actual parameters in a procedure call. The dope vector is allocated in the code when at least one of the arrays appears as an actual parameter in a procedure call, but appears otherwise in the stack.

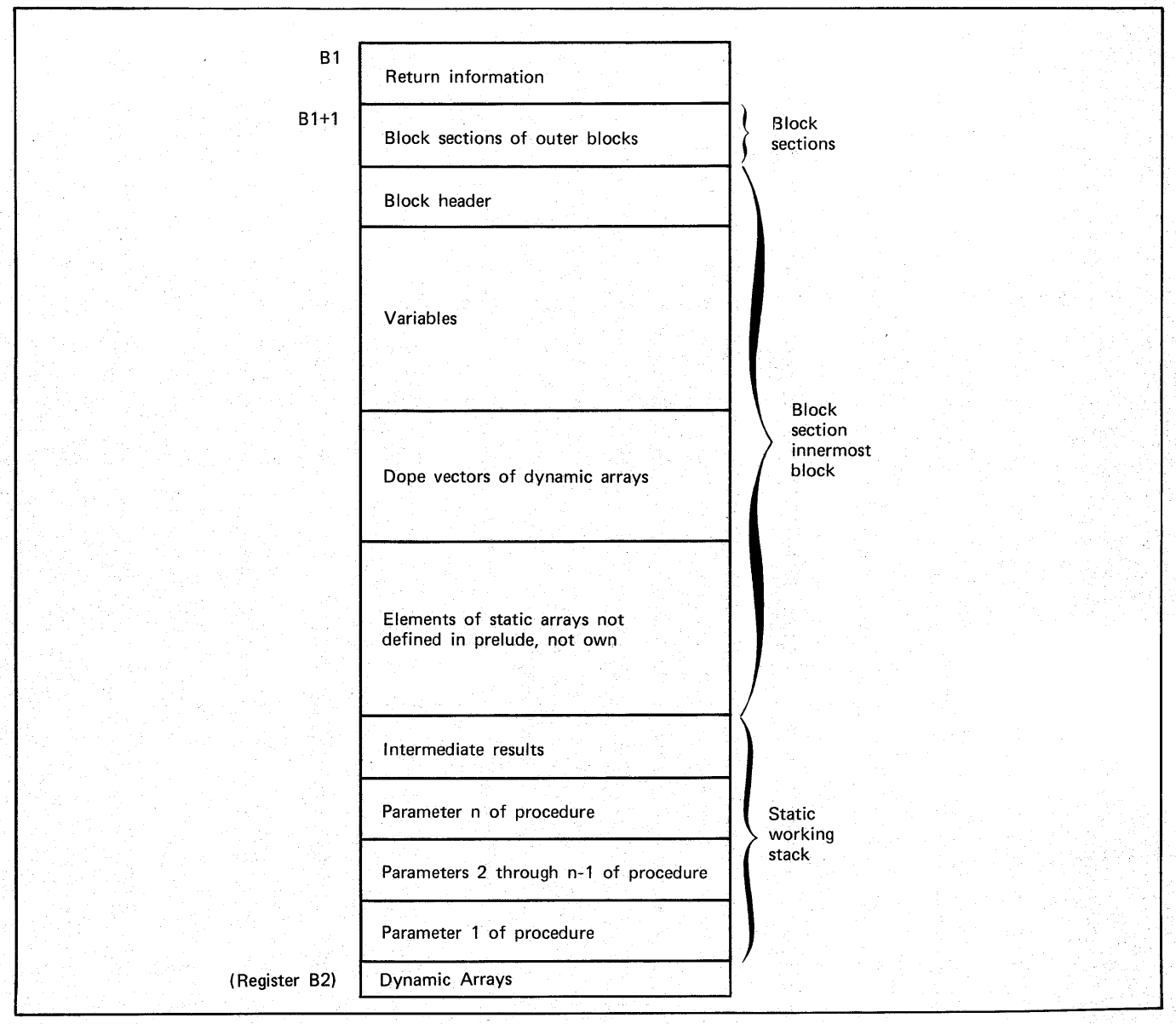


Figure E-2. Program Level in Scalar Stack

The stack also contains the following entities:

- The elements of an array with static bounds are allocated in the block section above the variables.
- A for statement with multiple forlist elements has one loop control word in the stack.
- The static working stack contains the parameters for a procedure being called and intermediate results of formulas that cannot be kept in the registers.
- The elements of arrays with dynamic bounds in the order of declaration. The lower address of the stack segment associated with any block is contained in a block header.

Labels, procedures, and switches are not represented in the stack at all.

STACK ENTRIES

The scalar stack contains:

Arithmetic and Boolean values.

Control words and descriptors of internal objects.

All values occupy one word. Real values are represented in standard normalized floating point format. Integer values are represented in standard fixed point format with 12 leading sign bits. Boolean values are represented by a full word of which the most significant bit is 1 for ≠TRUE≠ and 0 for ≠FALSE≠ and the other bits are irrelevant.

The formats of control words and descriptors are shown in figure E-3.

CCCCCC An 18-bit data address in code or common.

SSSSSS An 18-bit address of a program level in the scalar stack.

BBBBBB A 17-bit relative address of a block header within its program level.

PPPPPP An 18-bit address in executable code.

LLLLLL An 18-bit length.

NNNN A 12-bit count.

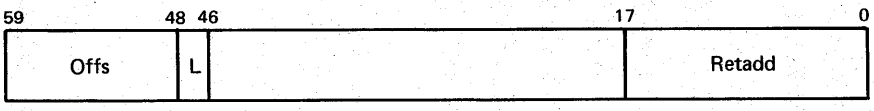
NNN A 9-bit count.

JJJJ A 12-bit instruction code with i and j designators.

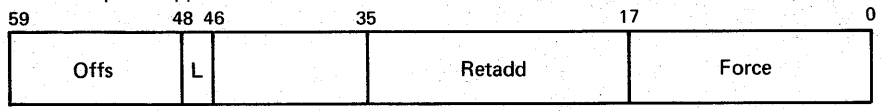
VVVVVV An 18-bit address of a variable in the scalar stack.

ZZZZZZZ A 21-bit address in small or large core.

Return information:
If SGM = 0:



If SGM option appears on ALGOL5 control statement:



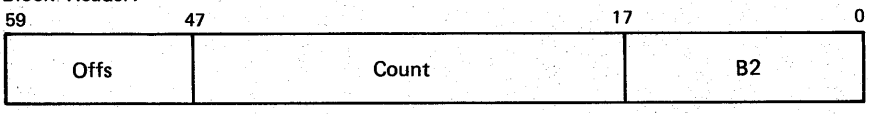
Offs Offset of current block header relative to register B1. The value is stored with a pack instruction, to that bit 58 is always 1. Present only if procedure has dynamic arrays.

L 0 INLIST/OUTLIST currently active
1 INLIST/OUTLIST not currently active

Retadd Return address

Force Address of code to force loading of the module containing the address specified by Retadd.

Block Header:

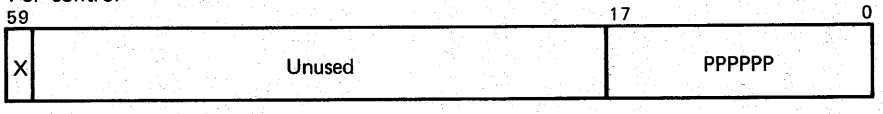


Offs Same as for return information.

Count Number of block headers. Used to check for accessibility of error labels.

B2 The value of register B2 associated with this block header.

For control



X 0 No addition of ≠STEP≠ on next cycle.
1 Addition of ≠STEP≠ on next cycle.

PPPPPP Program address for the current forlist element.

Figure E-3. Control Words and Descriptors in Scalar Stack (Sheet 1 of 4)

T is a 2-bit code (bits 54-55) for type as follows:

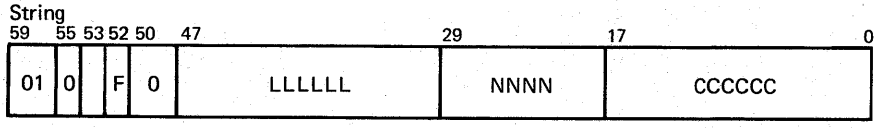
0 Void

1 Boolean

2 Integer

3 Real

String



F Flag indicator for usage and kind of string:

0 The string has not been analyzed as format string.

1 The string has been converted to another format internally used as format string; in this case the length is in words.

2 The string cannot be used as format string.

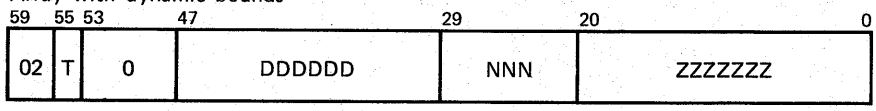
3 The string can be used as format string.

LLLLLL Length of the string in characters or words.

NNNN Level of replicator nesting in the format; used only when F = 1.

CCCCCC Address of the string.

Array with dynamic bounds



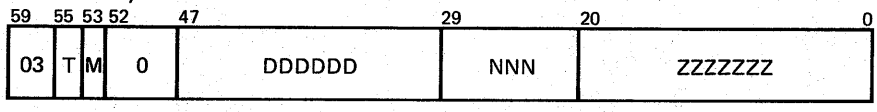
DDDDDD Address of the dope vector.

NNN Number of dimensions of the array.

ZZZZZZZ Address of the element of the array with all subscripts equal to zero.

For a dynamic array, ZZZZZZZ is relative to the top of the array stack for this level.

Virtual array



DDDDDD Address of the dope vector in stack or code.

NNN Number of dimensions of the array.

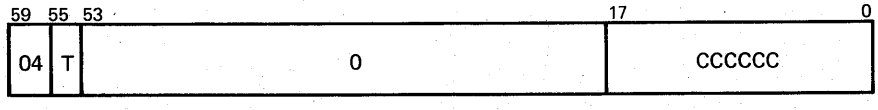
ZZZZZZZ Address of the element of the array with all subscripts equal to zero.

M Memory residence indicator:

0 Central memory
1 Extended memory

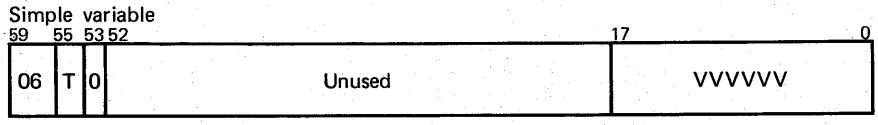
Figure E-3. Control Words and Descriptors in Scalar Stack (Sheet 2 of 4)

Constant



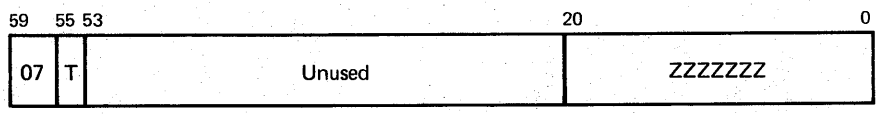
CCCCCC Address of the constant in common.

Simple variable



VVVVVV Address of the variable in stack or common.

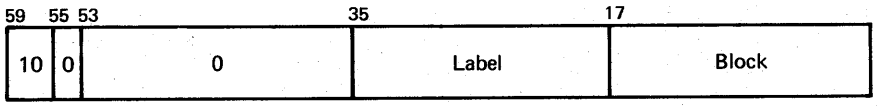
Subscripted variable fixed subscripts



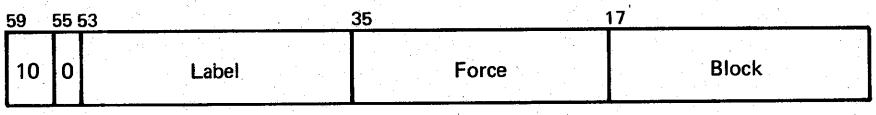
ZZZZZZZ Address of the array element in CM/SCM or LCM/ECS.

Label

If SGM = 0:



If SGM option is present on ALGOL5 control statement:

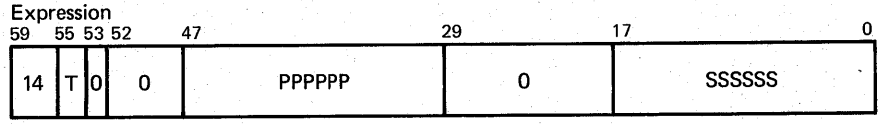


Label Program address of label.

Block Address of block header associated with label.

Force Same as for return information.

Expression

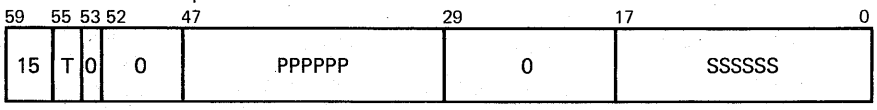


PPPPPP Program address where evaluation of the expression begins.

SSSSSS The address of the innermost program level from which identifiers are referenced in the expression.

Figure E-3. Control Words and Descriptors in Scalar Stack (Sheet 3 of 4)

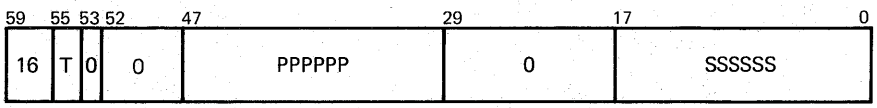
Procedure without parameters



PPPPPP Program address of the procedure's entry point.

SSSSSS The address of the innermost program level outside the procedure from which identifiers are referenced in the procedure body.

Subscripted variable

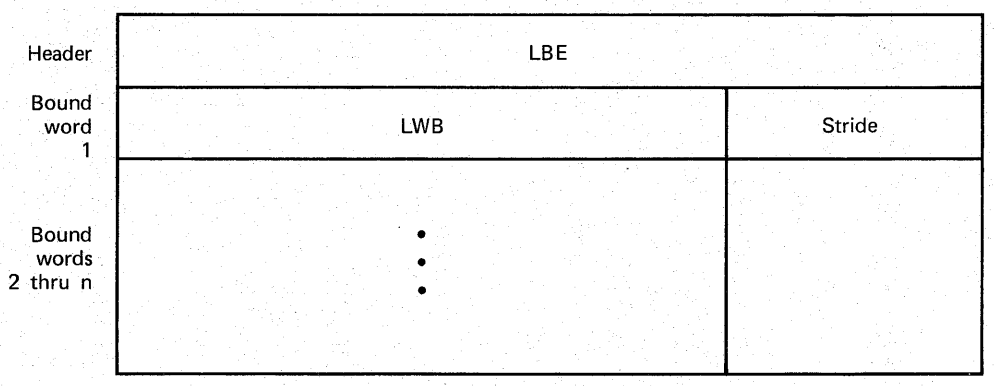


PPPPPP Program address where evaluation of the subscripted variable begins.

SSSSSS The address of the innermost program level from which identifiers are referenced in the subscripted variable.

Dope vector

The dope vector of an n-dimensional array occupies n+1 words, namely a one-word header and a bound word for each dimension.



LBE Lower bound effect; that is FWA-ZZZZZZ, when FWA is the address of the element with all subscripts equal to the lower bound.

LWB Lower bound.

Stride The multiplier for the preceding subscript position in subscripting. The stride field of the first bound word is then the total size of the array. The multiplier for the last subscript is always 1.

Figure E-3. Control Words and Descriptors in Scalar Stack (Sheet 4 of 4)

HEAP ORGANIZATION

All entries in the heap are preceded by a header word in the format shown in figure E-4.

There are four classes of heap entries:

1. Channels
   Purpose: Definition of a channel number and associated data control block.

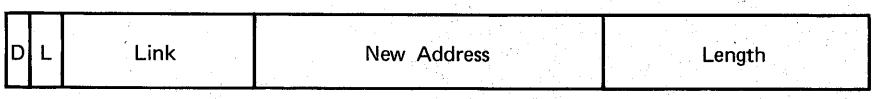
   Channel entries are chained through their LINK-field; the head of the chain is in G↓CHN.

2. Channel definition block
   Purpose: A data control block is created for each file defined in the program (by means of a call to the procedure CHANNEL).

   References to data control blocks are found in the list of files, in the channel entries, and in the program levels of INLIST or OUTLIST activations in the stack.

3. Input/Output buffers (NOS and NOS/BE only)
   Purpose: Provide buffers for files to be used by Record Manager for NOS or NOS/BE. SCOPE 2 Record Manager provides its own buffer for user files outside the user's field length.

   All references to buffers are in the channel definition block the buffer is associated with.



D
0 Entry is valid.
1 Entry is obsolete.

L
0 Entry belongs to an activation of INLIST or OUTLIST.
1 Life time of entry does not depend on activation of INLIST or OUTLIST.

Link — Address of the next entry of the same class. This field is used only for Channel entries.

New Address — The address where the header word will be moved to. This field is used only during the move/compact process.

Length — Length in words of the entry including the header word.

Figure E-4. Heap Organization

4. Formats
   Purpose: To contain converted format string, created by a call to the procedure FORMAT.

   References are in the scalar stack in the program level of an INLIST or OUTLIST activation.

CODE GENERATED FOR SPECIFIC LANGUAGE CONSTRUCTS

The code generated for parameter transmission and procedure calls is discussed here.

PARAMETER TRANSMISSION

Parameters are transmitted on top of the working stack, in the reverse order of the source text, each parameter occupying one word.

Parameters called by value are evaluated before transmission and the resulting value is stored on the stack.

For a parameter called by name, a parameter descriptor is constructed and stored on the stack. When the procedure called is itself a formal parameter, all parameters are transmitted as if called by name.

PROCEDURE CALL

The call of a procedure that is not a formal parameter is performed in the following steps:

1. Place the parameters on the stack.
2. When the procedure being called has nonglobal scope (that is, the procedure body contains references to nonglobal variables outside itself) obtain the static link of the procedure.
3. Construct the return information, using the static link if applicable, and store it on the stack.
4. Set the register B1 to the address of the return information in the stack.
5. Transfer control to the procedure by a simple jump to its start address.
6. Upon return from the procedure, reset B1 to its value before the call by subtracting the size of the current program level from it.

The call of a procedure that is a formal parameter is performed in the following steps:

1. Place the parameters on the stack. All parameters are handled as if called by name.
2. Fetch the descriptor of the procedure being called in register X1.
3. Fetch a jump instruction to the return address in register X2.
4. Set register B1 to the first address in the stack above the parameters.
5. Transfer control to the execution routine G↓CALL that will:
   a. Construct the return information from the return instructions and the static link extracted from the procedure descriptor and store it on the stack.
   b. Check the actual parameters against the specifications of the formal parameters.
   c. Evaluate all parameters called by value and overwrite their descriptors in the stack with the result of the evaluation.
   d. Transfer control to the procedure being called.
6. Upon return from the procedure, reset B1 to its value before the call by subtracting the size in words of the program level from it.

ENTRIES IN THE CODE

Entries in the code include the module info, procedure info, specification info, and call info.

MODULE INFO

A module info is generated for each relocatable module. It consists of a two-word header, a block table (if the D option is selected) and a line table.

The format of the header is shown in figure E-5.

PROCEDURE INFO

A two-word procedure info is generated for each procedure body in the program. The procedure info can be preceded by the specification info.

The format is shown in figure E-6.

SPECIFICATION INFO

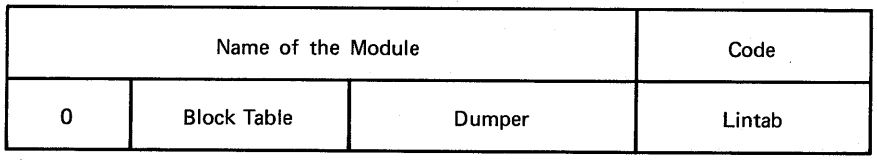
The specification info consists of 15-bit bytes. The first byte holds the number of formal parameters of the procedure. The check bytes are used to check the specifications and some attributes of the actual parameters. One check byte is used for each parameter. A second byte is used for an array parameter for which the number of dimensions of the actual array parameter must be checked.

The format is shown in figure E-7.

CALL INFO

A one-word call info is generated for each call in the program in front of the return address.

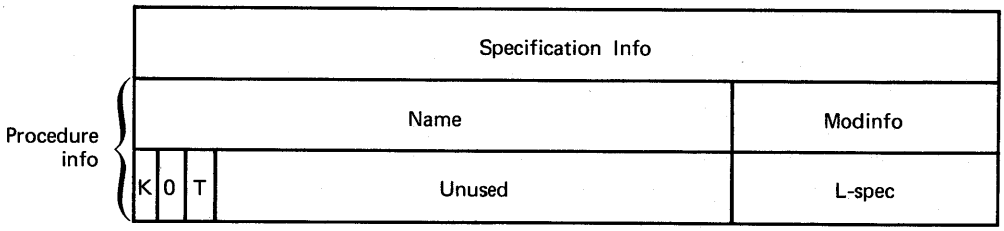
The format is shown in figure E-8.



Name of the module	Module name consisting of seven display code characters.
Lintab	The address of a table used to determine the line number in the source text from a relative address in the program code (line table).
Code	The address of the start of executable code in the program, used to convert an absolute address into a relative address in the program code.
Dumper	The address of the execution time symbolic dump routine. This field is zero when the D option is not selected.
Block Table	The address of the block table for this module. This field is zero when the D option is not selected.

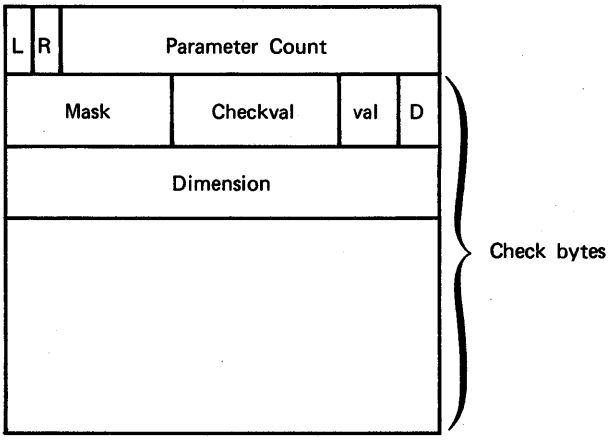
The module information is generated at the beginning of the module in a special segment, TPLC.

Figure E-5. Module Info Header



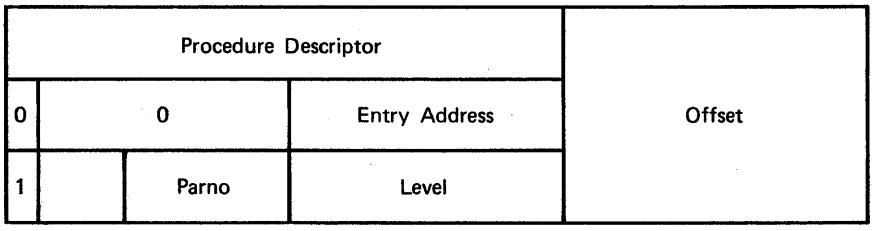
Name	The first seven characters of the identifier for this procedure.
Modinfo	The address of the module info for the program this procedure is part of.
K	1 Variable number of parameters. 0 Fixed number of parameters.
T	Type of the procedure.
L-spec	The length in words of the specification info.

Figure E-6. Procedure Info Format



L	1 Actual number of parameters can exceed parameter count. 0 Otherwise.
Parameter count	The minimum number of actual parameters to the procedure.
Mask	A mask used to extract information from the kind and type fields of the descriptor of the actual parameter.
Checkval	A value to be compared with the information extracted from the kind and type fields of the descriptor of the actual parameters.
Val	0 Name parameter; otherwise, type of value parameter.
D	1 This specification byte is followed by a dimension byte. 0 This specification byte is not followed by a dimension byte.
Dimension	The number of the dimensions that the actual parameter array must have.

Figure E-7. Specification Info Format



Offset	The negative value to be added to the stack pointer upon return of the procedure.
Procedure descriptor	Used to locate the entry point of the procedure being called. Its interpretation depends on the left-most bit.
Entry address	Entry point of the called procedure.
Parno	The negated number of parameters of the called procedure.
Level	The difference in static levels of the calling procedure and the procedure of which the called procedure is a parameter.

Figure E-8. Call Info Format

LINE TABLE

The line table is part of the module info. It contains information about the conversion from relative address (in the executable code) to source line number. The table consists of a sequence of groups of 6-bit bytes.

A count is a group of 6-bit bytes with the format of either a byte with value 1 through 62 or a sequence of bytes with value 63, followed by a byte with value 0 through 62.

A count denotes a value which is the sum of all its bytes (always greater than 0).

Each group in the line table has one of the following formats:

- A count, which denotes the number of words of code generated for the current line, if any.
- A zero byte, followed by a count, which denotes the number of lines for which no code was generated.
- Two zero bytes, which denote the end of the table.

BLOCK TABLE

The block table is generated only when the dump option is specified on the ALGOL5 control statement. Each block in the program has its own subtable, consisting of a two-word header and a one-word entry for each identifier declared in the range.

The format is shown in figure E-9.

DEDICATED B-REGISTERS

The dedicated B-registers are as follows:

B1 Stack pointer for the current program level.

B2 Pointer to the end of the last dynamic array in the stack +1.

B3 Pointer to a location approximately 260 words below the heap; only modified if the heap is moved.

Register B1 must be reloaded on every procedure entry and exit and on a goto statement that discards the current program level.

Register B2 must be reloaded on every array declaration and upon exit of a block containing arrays.

Register B3 is modified only when the stack is moved, because the field length is modified, or when a new file information table and buffer must be allocated and the field length cannot be adjusted.

Header

0	Offset	Length	Start Address
0	Own Start	Total Length	End Address

Offset	The relative address in the program level of the first identifier in the block.
Length	The length of the subtable for this block.
Start Address	The address of block entry relative to the start of executable code (field Code in the module info).
Own Start	Address in the own common block of the first own identifier of the range.
Total Length	The length of the consecutive subtables for this block and all its inner blocks.
End Address	The address of block exit relative to the start of executable code (field Code in the module info).

Figure E-9. Block Table Format (Sheet 1 of 2)

Identifier Entry

Identifier	Reserved	Own		Name	K	T

Identifier	The first seven characters of the identifier.
Reserved	Reserved for later use.
Own	1 For own identifiers, 0 otherwise.
Name	1 For call-by-name parameter. 0 For normal identifier, label or call-by value parameter.
K	Kind of the identifier.
T	Type of the identifier.

Block Word

0	Fwa	Lwa	Offset

Fwa	Address where the generated code for the block starts.
Lwa	Address where the generated code for the block ends.
Offset	The offset of the block header in the program level.

Figure E-9. Block Table Format (Sheet 2 of 2)

INDEX

COMMENT SHEET

MANUAL TITLE: ALGOL-60 Version 5 Reference Manual

PUBLICATION NO.: 60481600 **REVISION:** C

NAME: _____

COMPANY: _____

STREET ADDRESS: _____

CITY: _____ **STATE:** _____ **ZIP CODE:** _____

This form is not intended to be used as an order blank. Control Data Corporation welcomes your evaluation of this manual. Please indicate any errors, suggested additions or deletions, or general comments below (please include page number references).

CUT ALONG LINE

AA3419 REV. 4/79 PRINTED IN U.S.A.

NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

FOLD ON DOTTED LINES AND STAPLE

TAPE

TAPE

FOLD

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 8241 MINNEAPOLIS, MINN.

POSTAGE WILL BE PAID BY

CONTROL DATA CORPORATION

Publications and Graphics Division
215 Moffett Park Drive
Sunnyvale, California 94086

CUT ALONG LINE

FOLD

FOLD

TAPE

TAPE