Burroughs

B5500

Information
Processing Systems

COMPATIBLE ALGOL
REFERENCE MANUAL

Burroughs

B 5500

INFORMATION PROCESSING SYSTEMS

COMPATIBLE ALGOL
REFERENCE MANUAL

Burroughs Corporation
Detroit, Michigan 48232

Printed in U. S. America 5-69 1038643

AA32499

This manual contains material from
"Burroughs B 5500 Information Processing Systems Extended ALGOL Reference Manual"

AA596952 AA739491

Correspondence regarding this document should be forwarded using the Remarks Form at the back of the manual, or may be addressed directly to Systems Documentation, Sales Technical Services, Burroughs Corporation, 6071 Second Avenue, Detroit, Michigan 48232.

TABLE OF CONTENTS

TABLE OF CONTENTS (cont)

TABLE OF CONTENTS (cont)

TABLE OF CONTENTS (cont)

TABLE OF CONTENTS (cont)

LIST OF ILLUSTRATIONS

LIST OF TABLES

NOTE

The various elements of Compatible ALGOL are discussed in paragraphs labeled Syntax, Semantics, and Restrictions immediately following each pertinent subject heading. To avoid needless repetition, these subordinate headings were omitted from the Table of Contents.

INTRODUCTION

Burroughs B 5500 Compatible ALGOL is based on the definitive "Revised Report on the Algorithmic Language ALGOL 60" (Communications of the ACM, Vol. 6, No. 1; January, 1963). This manual describes the ALGOL language implemented by the B 5500 Compatible ALGOL Compiler. Compatible ALGOL represents a subset of B 6500 Extended ALGOL and should be useful for facilitating conversion to a B 6500 System. This language is intended to be used where character mode manipulation is required without jeopardizing the integrity of the operating system or other multiprocessing programs.

SECTION 1
STRUCTURE OF THE LANGUAGE

GENERAL.

The ALGOL 60 Language deals with the formation of rules for calculation of a value or values by means of a computer. B 5500 Compatible ALGOL contains additional language constructs which allow programmers to perform input and output operations and efficiently manipulate data in the form of character strings.

Compatible ALGOL employs a vocabulary of reserved words and symbols. The use of these reserved words and symbols in a program is defined by the language description in this manual.

Reserved words and symbols are grouped in ways prescribed by the syntax to form the various constructs of the language. These constructs can be divided into five major categories: basic components, general components, expressions, statements, and declarations.

Basic components may be combined in accordance with the rules of the language to form general components and expressions. Four different forms of expressions are defined in the language: arithmetic, Boolean, designational, and pointer.

The results produced by the evaluation of arithmetic, Boolean, and pointer expressions can be assigned as the values of variables by means of assignment statements. These assignment statements are the principle active elements of the language.

In addition, to provide control of the computational processes and external communication for a program, certain additional statements are defined. These statements provide iterative mechanisms, conditional and unconditional program control transfers, and input/output operations. In order to provide control points for transfer operations, statements may be labeled.

Declarations are provided in the language for giving the compiler

information about the constituents of the program, such as array sizes, the types of values that variables may assume, or the existence of subroutines. Each such construct must be named by an identifier, and all identifiers must be declared before they are used.

A series of statements enclosed by the reserved words BEGIN and END is called a compound statement. If a declaration of identifiers appears immediately after the word BEGIN and prior to the related statements, the statement group is called a block. Both compound statements and blocks provide a method for grouping related statements, and they therefore can be the constituents of still more compound statements and blocks. A program is a grouping of such statements.

CONVENTIONS USED IN THE DESCRIPTION OF THE LANGUAGE.

A metalanguage is a language used to talk about other languages. A metalinguistic symbol is a symbol used in a metalanguage to define the syntax of a language. The following metalinguistic symbols will be used in this manual:

a. ⟨⟩ Left and right broken brackets are used to contain one or more characters representing a metalinguistic variable whose value is given by a metalinguistic formula.

b. ::= The symbol ::= means "is defined as." It separates the metalinguistic variable on the left of a metalinguistic formula from the definition on the right.

c. | The symbol | means "or." This symbol separates multiple definitions of a metalinguistic variable.

d. {} Braces are used to enclose metalinguistic variables which are defined by the meaning of the English language expression contained within the braces. This formulation is used only when it is impossible or impractical to use a metalinguistic formula.

Metalinguistic symbols are used in forming a metalinguistic formula. A metalinguistic formula is a rule which will produce an allowable sequence of characters and/or symbols. These formulae are used to define the syntax of the B 5500 Compatible ALGOL language. The syntax, in conjunction with the semantics contained in this manual, defines the B 5500 Compatible ALGOL language.

Any mark or symbol in a metalinguistic formula which is not one of the above metalinguistic symbols denotes itself. The juxtaposition of metalinguistic variables and/or symbols in a metalinguistic formula denotes juxtaposition of these elements in the construct indicated.

An example of a metalinguistic formula is:

⟨identifier⟩ ::= ⟨letter⟩ | ⟨identifier⟩ ⟨letter⟩ |
⟨identifier⟩ ⟨digit⟩

This metalinguistic formula is read: an identifier is defined as a letter, or an identifier followed by a letter, or an identifier followed by a digit.

The metalinguistic formula given above defines a recursive relationship by which a construct called an identifier may be formed. That is, evaluation of the formula shows that an identifier begins with a letter. The letter may stand alone, or may be followed by any mixture of letters and digits.

CHARACTER SET.

SYNTAX.

⟨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

⟨special character⟩ ::= .|,|[|]|(|)|+|-|⊗|/|<|>|≤|≥|=|≠|
←|%|$|*|#|@|:|;|&

⟨string character⟩ ::= ⟨letter⟩ | ⟨digit⟩ | ⟨special character⟩ | ⟨single space⟩

⟨string bracket character⟩ ::= "

⟨single space⟩ ::= {one horizontal blank position}

⟨space⟩ ::= ⟨single space⟩ | ⟨space⟩ ⟨single space⟩

⟨invalid character⟩ ::= ?

⟨character⟩ ::= ⟨string character⟩ | ⟨string bracket character⟩ | ⟨invalid character⟩

SEMANTICS.

The Burroughs Common Language character set consists of 64 characters: letters, digits, special characters, the space, the string bracket character, and the invalid character.

SECTION 2

BASIC COMPONENTS:

BASIC SYMBOLS, IDENTIFIERS, NUMBERS, AND STRINGS

GENERAL.

SYNTAX.

The syntax for ⟨basic symbol⟩ is as follows:

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

⟨logical value⟩ ::= TRUE | FALSE

⟨empty⟩ ::= {the null string of symbols}

SEMANTICS.

Only upper case letters are permitted.

Individual letters do not have individual meanings.

DELIMITERS.

The syntax for ⟨delimiter⟩ is as follows:

⟨delimiter⟩ ::= ⟨operator⟩ | ⟨separator⟩ | ⟨bracket⟩ |
⟨declarator⟩ | ⟨specificator⟩

⟨operator⟩ ::= ⟨arithmetic operator⟩ | ⟨relational operator⟩ |
⟨logical operator⟩ | ⟨sequential operator⟩
⟨replacement operator⟩ | ⟨concatenate operator⟩

⟨arithmetic operator⟩ ::= + | - | ⊗ | / | DIV | MOD |
* | TIMES

⟨relational operator⟩ ::= < | > | ≤ | ≥ | = | ≠ | LSS | LEQ |
EQL | GEQ | GTR | NEQ

⟨logical operator⟩ ::= EQV | IMP | OR | AND | NOT

⟨sequential operator⟩ ::= GO | IF | THEN | ELSE | FOR | DO |
CASE | FILL | WHILE | REPLACE | SCAN

⟨replacement operator⟩ ::= ← | :=

⟨concatenate operator⟩ ::= &

⟨separator⟩ ::= , | . | : | ; | @ | ⟨space⟩ | STEP | UNTIL |
COMMENT | WITH | OF | BY | TO

⟨bracket⟩ ::= (|) | [|] | " | BEGIN | END | # | LB | RB

⟨declarator⟩ ::= OWN | BOOLEAN | INTEGER | REAL | ARRAY |
SWITCH | LABEL | FORWARD | SAVE | PROCEDURE |
LIST | MONITOR | DUMP | FILE | ALPHA | DEFINE |
POINTER | FORMAT | SET

⟨specificator⟩ ::= VALUE

SEMANTICS.
Delimiters are the class of operators, separators, brackets, declarators, and specificators. As the word "delimiter" indicates, an important function of these elements is to separate the various entities which make up a program.

In order to accept input from equipment not having the full character set as shown on page 1-3, alternate representations of certain delimiters are provided as follows:

LSS	<
LEQ	≤
EQL	=
GEQ	≥
GTR	>
NEQ	≠
TIMES	⊗

LB	[
RB	]
:=	←

Throughout the text of this manual, the symbols in the right-hand column are used.

Delimiters have fixed meanings which will be made clear as they appear in various constructs in this manual. Delimiters and logical values are considered basic symbols of the language, having no relation to the individual letters of which they are composed. Consequently, the words which constitute the basic symbols are reserved for specific use in the language. A complete list of these words and details of the applicable restrictions are given in appendix A.

SPACING.

In ALGOL 60, spaces have no significance, since basic components of the language such as BEGIN are construed as one symbol. In a machine implementation of such a language, however, this approach is not practical. In Compatible ALGOL, for instance, BEGIN is composed of five letters, TRUE is composed of four, and PROCEDURE of nine. No space may appear between the letters of a reserved word; otherwise, it will be interpreted as two or more elements.

The basic components (reserved words and symbols) are used, together with variables and numbers, to form expressions, statements, and declaratives. Because some of these constructs place quantities which have been defined by the programmer next to delimiters composed of letters, it is necessary to separate one from the other. The space is used as a delimiter in these cases; therefore, a space must separate any two basic components of the following forms:

a. Multicharacter delimiter.*

b. Identifier.

c. Logical value.

d. Unsigned number.

Aside from these requirements, a space may appear (if desired) between any two basic components without affecting their meaning.

THE USE OF COMMENTS.

In order to include explanatory material at various points in the program, several conventions exist as defined below. The reserved word COMMENT indicates that the information following is explanatory rather than part of the program structure.

Sequence of Basic Symbols	Equivalent
; COMMENT {any sequence of characters not containing ;} ;	;
BEGIN COMMENT {any sequence of characters not containing ;} ;	BEGIN
END {any sequence of letters and/or digits, including blanks, but excluding the symbols END, ELSE, UNTIL, or ;}	END

The above conventions mean that any construct which appears on the left may be used in place of the corresponding construct on the right without any effect on the operation of the program.

BASIC COMPONENTS.

SYNTAX.

The syntax for ⟨basic component⟩ is as follows:

⟨basic component⟩ ::= ⟨identifier⟩ | ⟨number⟩ | ⟨string⟩

* Except those multicharacter delimiters which begin or end with special characters.

SEMANTICS.

Basic components are the most primitive structures of Compatible ALGOL.

IDENTIFIERS.

SYNTAX.

The syntax for ⟨identifier⟩ is as follows:

⟨identifier⟩ ::= ⟨letter⟩ | ⟨identifier⟩ ⟨letter⟩ |
⟨identifier⟩ ⟨digit⟩

Examples:

I

ID

A5

G76D3

ARITHMETICMEAN

SEMANTICS.

Identifiers have no absolute meaning.

Identifiers are used to name labels, variables, arrays, switches, procedures, files, formats, lists, and so forth. The identifiers used in a program may be chosen freely.

RESTRICTIONS.

Type 1 reserved words of Compatible ALGOL may not be used as identifiers (see appendix A).

An identifier must start with a letter, which can be followed by any combination of letters or digits, or both. The latter restriction also applies to labels, since integer labels are specifically disallowed.

No space may appear within an identifier.

Identifiers may be as short as one letter or as long as 63 letters and digits.

The same identifier cannot be used to denote two different entities simultaneously.

The Type 2 standard function designators and the Type 3 multi-character delimiters listed in appendix A may be declared as identifiers. An identifier so declared may not be used as a function or a delimiter within the scope of the declaration.

NUMBERS.

SYNTAX.

The syntax for ⟨number⟩ is as follows:

⟨number⟩ ::= ⟨sign⟩ ⟨unsigned number⟩ | ⟨string⟩

⟨unsigned number⟩ ::= ⟨decimal number⟩ | ⟨exponent part⟩ |
⟨decimal number⟩ ⟨exponent part⟩

⟨decimal number⟩ ::= ⟨unsigned integer⟩ | ⟨decimal fraction⟩ |
⟨unsigned integer⟩ ⟨decimal fraction⟩ |
⟨unsigned integer⟩.

⟨exponent part⟩ ::= @⟨integer⟩

⟨decimal fraction⟩ ::= .⟨unsigned integer⟩

⟨integer⟩ ::= ⟨sign⟩ ⟨unsigned integer⟩

⟨unsigned integer⟩ ::= ⟨digit⟩ | ⟨unsigned integer⟩
⟨digit⟩

⟨sign⟩ ::= ⟨empty⟩ | + | -

Examples:

Numbers:	Unsigned Numbers:	Decimal Numbers:
0	1354.543	1354
549755813887	@68	.546
8.758@-47	1354.54@68	1354.543
4.314@68		

Exponent Parts:	Decimal Fractions:	Integers:
@68	.5	+546
@-46	.69	-62256
@+54		12

Unsigned Integers:
5
69

SEMANTICS.
Numbers may be of two basic types: INTEGER or REAL. Integers are of type INTEGER; all other numbers are of type REAL.

The number sets are symmetrical with respect to zero (i.e., the negative number corresponding to any valid positive number may also be expressed in the language and the object program).

The exponent part is a scale factor expressed as an integral power of 10.

No space may appear within an unsigned number; an embedded space will cause it to be interpreted as more than one number.

SIZE LIMITATIONS OF NUMBERS.
In general, the number of digits (disregarding the decimal point and exponent part, if any) in an unsigned number may not exceed eleven; otherwise, the value will be truncated to the most significant eleven digits. Twelve digits are allowed if, disregarding

the decimal point and exponent part, they do not exceed 549755813887 in value.

The maximum absolute value of a single precision real number is approximately 4.314@68 and the minimum value is approximately 8.758@-47.

A string may be used to represent a number. The length of such a string is limited to seven BCL characters or 15 octal characters.

STRINGS.

SYNTAX.

The syntax for ⟨string⟩ is as follows:

⟨string⟩ ::= ⟨ALPHA string⟩ | ⟨numeric string⟩

⟨ALPHA string⟩ ::= "⟨BCL string⟩"

⟨numeric string⟩ ::= 3 " ⟨octal string⟩ "

⟨octal string⟩ ::= ⟨octal character⟩ | ⟨octal string⟩ ⟨octal character⟩

⟨octal character⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

⟨BCL string⟩ ::= ⟨string character⟩ | " | ⟨BCL string⟩ ⟨string character⟩

SEMANTICS.

Strings may consist of:

three-bit characters (octal)
six-bit characters (BCL)

The type of string is indicated by the presence (or absence) of a special code which precedes the string. A code of 3 indicates that the string following is an octal string. The absence of a string code indicates that the string is a BCL string of six-bit

characters. The string is normally justified right with leading zeros. However, if the string contains more than 48 bits, it is justified left with trailing zeros.

The quote character (") may be used in a BCL string only if it immediately follows the initial bracketing quote.

The permissible length of a string depends upon the context in which the string is used. When used as an operand, it is normally not greater than seven BCL characters or 15 octal characters.

The maximum length of a BCL string is 63 characters. The maximum length of an octal string is 16 octal characters.

CONSTITUENTS AND SCOPES.

The following kinds of quantities must be declared before they may be referred to in Compatible ALGOL programs: simple variables, arrays, labels, switches, procedures, files, formats, definitions, lists, forward references, and diagnostics.

The scope of any quantity is the block in which the quantity is declared.

VALUES AND TYPES.

Certain syntactical units have values. The value of an arithmetic expression is a number, the value of a Boolean expression is a logical value, and the value of a designational expression is a label. The value of an array identifier is the ordered set of values of the associated subscripted variables; this may be a set of numbers, a set of logical values, or a set of proper strings.

The types (INTEGER, REAL, BOOLEAN, and ALPHA) associated with syntactical units refer to the values of these units.

SECTION 3
GENERAL COMPONENTS

GENERAL.

SYNTAX.

The syntax for ⟨general component⟩ is as follows:

⟨general component⟩ ::= ⟨variable⟩ | ⟨partial word designator⟩ | ⟨switch file designator⟩ | ⟨switch format designator⟩ | ⟨switch list designator⟩ | ⟨function designator⟩

SEMANTICS.

General components are constituents of expressions. Normally, general components are less complex structures than expressions.

It should be understood, however, that no sharp dividing line can be drawn between general components and expressions since they are used recursively; i.e., expressions are formed from general components, but general components also use expressions in their definitions.

VARIABLES.

SYNTAX.

The syntax for ⟨variable⟩ is as follows:

⟨variable⟩ ::= ⟨simple variable⟩ | ⟨subscripted variable⟩

⟨simple variable⟩ ::= ⟨variable identifier⟩

⟨variable identifier⟩ ::= ⟨identifier⟩

⟨subscripted variable⟩ ::= ⟨array identifier⟩ [⟨subscript list⟩]

⟨array identifier⟩ ::= ⟨identifier⟩

⟨subscript list⟩ ::= ⟨subscript⟩ | ⟨subscript list⟩,⟨subscript⟩

⟨subscript⟩ ::= ⟨arithmetic expression⟩

Examples:

Simple Variables:

ALPHAINFO

BETA4

Q

Subscripted Variables:

A[5]

A [ITH]

KRONECKER [ITH + 2, JTH - ITH]

MAXQ [IF BETA = 30 THEN -2 ELSE K + 2]

Subscript Lists:

5

ITH

ITH, JTH

ITH + 2, JTH - ITH

IF BETA = 30 THEN -2 ELSE K + 2

SEMANTICS.

A variable is the symbolic representation of a particular value. A variable may be used in an expression in order to produce another value. The value designated by a variable may be changed through the use of an assignment statement (see page 6-6, assignment statement). There are two forms of variables: simple and subscripted.

SIMPLE VARIABLES.

A simple variable is defined as being composed of a single variable identifier used to reference some quantity. The type of value that a simple variable may represent is defined by its type declaration (see page 7-2, type declarations).

SUBSCRIPTED VARIABLES.

A subscripted variable is an array identifier followed by a subscript list. The array identifier refers to a set of values (see array declaration, page 7-4). An array identifier with a subscript list refers to a single value within that set. The subscript list specifies one element of the array.

A subscript expression is defined as an arithmetic expression. Each arithmetic expression occupies a subscript position in the subscript list and is referred to as a subscript.

NUMBER OF SUBSCRIPTS.

The total number of subscripts in a subscript list must equal the number of dimensions given in the array declaration or array specification.

EVALUATION OF SUBSCRIPTS.

Each subscript expression in the subscript list is evaluated from left to right. Each subscript expression is treated as a variable of type INTEGER. If, upon evaluation, the subscript expression yields a value of type REAL, it will be rounded automatically as follows (see page 3-11, type transfer functions):

integral subscript value = ENTIER (subscript value + 0.5)

The values which result from the evaluation of the subscript expressions provide the actual integral values of the subscripts by which the array component is referenced. If the value of a subscript falls outside the limits declared for the array, the value of the element so referenced is undefined and an invalid index error is generated at run time.

FILE DESIGNATORS.

SYNTAX.

The syntax for ⟨file designator⟩ is as follows:

⟨file designator⟩ ::= ⟨switch file identifier⟩
[⟨subscript⟩] | ⟨file identifier⟩

⟨switch file identifier⟩ ::= ⟨identifier⟩

⟨file identifier⟩ ::= ⟨identifier⟩

SEMANTICS.
A file designator specifies a file.

A switch file identifier refers to a set of files.

SWITCH FILE DESIGNATORS.
SYNTAX.
The syntax for ⟨switch file designator⟩ is as follows:

3 ⟨switch file designator⟩ ::= ⟨switch file identifier⟩
[⟨subscript⟩]
3 ⟨switch file identifier⟩ ::= ⟨identifier⟩

Examples:

```
SWHFI[I]
SWIFI[IF X > N THEN 0 ELSE I]
FISW[REAL (X ≤N)]
```

SEMANTICS.
Switch file designators are used in I/O statements in the same fashion as file identifiers.

A switch file designator is used in conjunction with the SWITCH FILE declaration specified by the switch file identifier. The value of the subscript expression determines which file identifier in the related switch file list is to be selected for use in the I/O statement. The value of the subscript expression must correspond to the position of one of the file identifiers in the switch file list. The values of these positions start with 0. If the value of the expression is other than integer, it will be converted to an integer in accordance with the rules applicable to subscript expressions. If the value of the expression is outside the scope of

the switch file list, the file so referenced is undefined.

FORMAT DESIGNATORS.

SYNTAX.

The syntax for ⟨format designator⟩ is as follows:

```
⟨format designator⟩ ::= ⟨switch format identifier⟩
                        [⟨subscript⟩] |
                        ⟨format identifier⟩
```

SEMANTICS.

A ⟨format designator⟩ specifies a file.

A ⟨switch format identifier⟩ refers to a set of formats.

SWITCH FORMAT DESIGNATORS.

SYNTAX.

The syntax for ⟨switch format designator⟩ is as follows:

```
3 ⟨switch format designator⟩ ::= ⟨switch format identifier⟩
                                 [⟨subscript⟩]
3 ⟨switch format identifier⟩ ::= ⟨identifier⟩
```

Examples:

```
SF[I]
SWHFT [IF X > N THEN 0 ELSE 1]
```

SEMANTICS.

Switch format designators are used in I/O statements in the same fashion as are format identifiers.

A switch format designator is used in conjunction with the SWITCH FORMAT declaration specified by the switch format identifier. The value of the subscript expression determines which editing specification part in the related switch format list is to be selected for use in the I/O statement. The value of the subscript expression must correspond to the position of one of the specification parts in the switch format list. The values of these positions start with 0. If the value of the expression is other than integer, it will be

converted to integer in accordance with the rules applicable to subscript expressions.

If the value of the expression is outside the scope of the switch format list, the editing specification so designated is undefined.

LIST DESIGNATORS.

SYNTAX.

The syntax for ⟨list designator⟩ is as follows:

⟨list designator⟩ ::= ⟨switch list identifier⟩ [⟨subscript⟩] |
⟨list identifier⟩

SEMANTICS.

A ⟨list designator⟩ specifies a list.

A ⟨switch list identifier⟩ refers to a set of lists.

SWITCH LIST DESIGNATORS.

SYNTAX.

The syntax for ⟨switch list designator⟩ is as follows:

3 ⟨switch list designator⟩ ::= ⟨switch list identifier⟩
[⟨subscript⟩]

3 ⟨switch list identifier⟩ ::= ⟨identifier⟩

Examples:

```
SWLST [I]
SWLI [IF A > B THEN 2 ELSE 3]
```

SEMANTICS.

Switch list designators are used in I/O statements in the same fashion as list identifiers.

A switch list designator is used in conjunction with the SWITCH LIST declaration specified by the switch list identifier. The value of the subscript expression determines which list identifier will be used from the switch list.

The value of the subscript expression must correspond to the posi-

tion of one of the list identifiers in the switch list. The values of these positions start with 0. If the value of the expression is other than integer, it will be converted in accordance with the rules applicable to subscript expressions. If the value of the subscript expression is outside the scope of the switch list, the list identifier so referenced is undefined.

FUNCTION DESIGNATORS.

SYNTAX.

The syntax for ⟨function designator⟩ is as follows:

⟨function designator⟩ ::= ⟨procedure identifier⟩ ⟨actual parameter part⟩

⟨actual parameter part⟩ ::= ⟨empty⟩ | (⟨actual parameter list⟩)

⟨actual parameter list⟩ ::= ⟨actual parameter⟩ | ⟨actual parameter list⟩ ⟨parameter delimiter⟩ ⟨actual parameter⟩

⟨actual parameter⟩ ::= ⟨expression⟩ | ⟨subarray designator⟩ | ⟨array identifier⟩ | ⟨switch identifier⟩ | ⟨file designator⟩ | ⟨format designator⟩ | ⟨switch file identifier⟩

⟨parameter delimiter⟩ ::= , |) "⟨letter string⟩" (

⟨letter string⟩ ::= ⟨letter⟩ | ⟨letter string⟩ ⟨letter⟩ | ⟨space⟩ | ⟨letter string⟩ ⟨space⟩

⟨procedure identifier⟩ ::= ⟨identifier⟩

⟨subarray designator⟩ ::= ⟨array identifier⟩ [⟨subscript part⟩ ⟨subarray part⟩]

⟨subscript part⟩ ::= ⟨empty⟩ | ⟨subscript list⟩ ,

⟨subarray part⟩ ::= * | ⟨subarray part⟩ , *

Examples:

Function Designators:

```
J(A, B + 2, Q [I,L])
GASVOL(K)"TEMPERATURE"(T)"PRESSURE"(P)
RANDOMNO
```

Actual Parameter Parts:

```
(A, B + 2, Q[I,J])
(K)"TEMPERATURE"(T)"PRESSURE"(P)
```

SEMANTICS.

A function designator is a procedure which returns a single value. The value is produced from the actual parameter(s) by the application of a given set of rules defined by a typed PROCEDURE declaration (see section 8, PROCEDURE declarations).

A function designator may be used, depending upon its type, in either arithmetic or Boolean expressions (see page 4-1, arithmetic expressions; page 4-10, Boolean expressions).

STANDARD FUNCTIONS.

The standard ("intrinsic") functions supplied for Compatible ALGOL are listed below. AE stands for arithmetic expression.

ABS(AE)	Produces the absolute value of AE.
SIGN(AE)	Produces one of three values, depending upon the value of AE (+1 for AE > 0, 0 for AE = 0, -1 for AE < 0).
SQRT(AE)	Produces the square root of the value of AE.
SIN(AE)	Produces the sine of the value of AE.
COS(AE)	Produces the cosine of the value of AE.
ARCTAN(AE)	Produces the principle value of the arctangent of the value of AE.

LN(AE)	Produces the natural logorithm of the value of AE.
EXP(AE)	Produces the exponential function of the value of AE, i.e., e^{AE}.
DELTA(P1,P2)	Yields an integer value representing the number of characters between the two pointer expressions (P2-P1). If P1 is greater than P2, the sign of delta is minus. P1 and P2 must refer to the same string; otherwise, a value of 2^{20} is returned.

Except for DELTA, which requires pointer expressions as arguments, these functions are understood to operate indifferently on arguments both of type REAL and type INTEGER. All these functions yield values of type REAL, except for SIGN(AE) which produces a result of type INTEGER. The function ABS(AE) also produces a result of type INTEGER when the value which results from the evaluation of AE is of type INTEGER.

For SIN, COS, and ARCTAN, the angle is considered to be in radians.

These functions may be used without a specific PROCEDURE declaration since they are an integral part of the compiler itself.

TIME FUNCTIONS.

TIME(AE) makes available the time registered on the internal timing device of the system. This feature may be used to measure the time required by the system, or certain components of it, to execute a program, or parts of a program (see table 3-1). (AE) must yield an integer value of zero through four. The result of the function is determined by the parameter.

Table 3-1
Results of Different TIME Parameters

Parameter	Result	Type
TIME (0)	Returns the current date in ALPHA (in the format "YYDDD").	ALPHA
TIME (1)	Returns as an integer value the time of day, in sixtieths of a second.	INTEGER
TIME (2)	Returns as an integer value the elapsed processor time of the job, in sixtieths of a second.	INTEGER
TIME (3)	Returns as an integer value the elapsed I/O time of the job, in sixtieths of a second.	INTEGER
TIME (4)	Returns as an integer value the contents of a 6-bit machine clock which increments every sixtieth of a second.	INTEGER

If the value of (AE) is not one of the integers indicated above, the result of the function will be undefined.

MAX AND MIN FUNCTIONS.

SYNTAX.

The syntax for MAX and MIN functions is as follows:

⟨limit function⟩ ::= ⟨limit function ID⟩ (⟨limit list⟩)
⟨limit function ID⟩ ::= MAX | MIN
⟨limit list⟩ ::= ⟨arithmetic expression⟩ | ⟨limit list⟩ ,
⟨arithmetic expression⟩

SEMANTICS.

The semantics of each function, MAX and MIN, is implied by its name, i.e., the value returned by the MAX function is the maximum value

of the arithmetic expression evaluated and the value returned by the MIN function is the minimum value so obtained.

Example:

```
Y :=MAX(3,5,I+J)
```

TYPE TRANSFER FUNCTIONS.

In addition to the set of standard functions provided for Compatible ALGOL, a set of type transfer functions is also provided. These type transfer functions are listed below, with their definitions following. The value returned by a type transfer function is a primary of the type indicated.

ENTIER (AE) — Transfers an expression of type REAL to an integral value which is the largest integer not greater than the value of AE.

INTEGER (AE) — Transfers an expression of the type REAL to an integral value equal to ENTIER (AE + 0.5).

BOOLEAN (AE) — Yields a value of type BOOLEAN. Permits arithmetic expressions to be used in BOOLEAN operations.

REAL (BE) — Yields a value of type REAL. Permits BOOLEAN expressions to be used in arithmetic operations.

```
REAL (TRUE) = 1
REAL (FALSE) = 0
```

REAL (P,N) — Yields N characters, starting at P, as an arithmetic value. P is a pointer expression, N is an arithmetic expression whose value must not specify more than 48 bits. If the value of N is equal to 8 and if the most-significant string bit referenced by P is equal to 1, a flag-bit error will result.

INTEGER (P,N) Yields an integer value represented by a string of N characters starting at P. P must be a BCL pointer expression. N is an arithmetic expression and must not be greater than 8.

NOTE

The functions REAL and BOOLEAN, used in conjunction, allow for handling masking operations since the logical operators (page 4-16) operate on the entire word in the system.

SECTION 4
EXPRESSIONS

GENERAL.

SYNTAX.

The syntax for ⟨expression⟩ is as follows:

⟨expression⟩ ::= ⟨arithmetic expression⟩ | ⟨Boolean expression⟩ | ⟨designational expression⟩ | ⟨pointer expression⟩

SEMANTICS.

Expressions, which are basic to any algorithmic process, are structures used to obtain values of different kinds and types.

As mentioned on page 3-1, expressions are used to define certain general components (subscripted variables and function designators), and these quantities in turn are used to define expressions. The definition of expressions is therefore necessarily recursive.

ARITHMETIC EXPRESSIONS.

SYNTAX.

The syntax for ⟨arithmetic expression⟩ is as follows:

⟨arithmetic expression⟩ ::= ⟨simple arithmetic expression⟩ | ⟨if clause⟩ ⟨arithmetic expression⟩ ELSE ⟨arithmetic expression⟩ | ⟨simple prefix⟩ ⟨term prefix⟩ ⟨factor prefix⟩ ⟨arithmetic assignment⟩

⟨simple arithmetic expression⟩ ::= ⟨simple prefix⟩ ⟨term⟩

⟨term⟩ ::= ⟨term prefix⟩ ⟨factor⟩

⟨factor⟩ ::= ⟨factor prefix⟩ ⟨primary⟩

⟨primary⟩ ::= ⟨unsigned number⟩ | ⟨string⟩ | ⟨partial word operand⟩ | ⟨partial word operand⟩ . ⟨field description⟩ | ⟨primary⟩ & ⟨arithmetic expression⟩ [⟨concatenation⟩]

⟨partial word operand⟩ ::= ⟨arithmetic variable⟩ | ⟨arithmetic function designator⟩ | (⟨arithmetic expression⟩)

⟨concatenation⟩ ::= [⟨left bit-to⟩ : ⟨left bit-from⟩ : ⟨number of bits⟩] | [⟨left bit-to⟩ : ⟨number of bits⟩]

⟨left bit-to⟩ ::= ⟨arithmetic expression⟩

⟨left bit-from⟩ ::= ⟨arithmetic expression⟩

⟨number of bits⟩ ::= ⟨arithmetic expression⟩

⟨simple prefix⟩ ::= ⟨sign⟩ | ⟨simple arithmetic expression⟩ ⟨adding operator⟩

⟨term prefix⟩ ::= ⟨empty⟩ | ⟨term⟩ ⟨multiplying operator⟩

⟨factor prefix⟩ ::= ⟨empty⟩ | ⟨factor⟩ *

⟨adding operator⟩ ::= + | -

⟨multiplying operator⟩ ::= x | / | DIV | MOD | TIMES

⟨arithmetic assignment⟩ ::= ⟨arithmetic variable⟩ ⟨replacement operator⟩ ⟨arithmetic expression⟩

⟨if clause⟩ ::= IF ⟨Boolean expression⟩ THEN

⟨arithmetic variable⟩ ::= ⟨variable⟩

⟨arithmetic function designator⟩ ::= ⟨function designator⟩

Examples:

Arithmetic Expressions:

```
A + B ← M - N
A + B ← M - N + Z ← X/Y
Q*V*2
P MOD 2
+3
(IF X = 1 THEN 5.5 ELSE Y/2)
IF ERROR[I] = 1 THEN "OVERFL" ELSE "UNFLOW"
IF B = 0 THEN X ELSE Y + 2
```

Simple Arithmetic Expressions:

```
COS(A + B)
Y*3
4 x R DIV S
+3
A[I] -B[J] + 5.3
```

Terms:

```
Y1[1,2]
2*(X + Y)
4 x R DIV S
P MOD 2
```

Factors:

```
5.678
2*(X + Y)
Y*3
Q*V*2
```

Primaries:

```
5.678
Y1[1,2]
COS(A + B)
(IF X = 1 THEN 5.5 ELSE Q/2)
I.[9:10]
"ALPHA"
```

Concatenation:

```
SQRT (C) & 1 [47:0:1]
X & Y [1:1:1] & Z [2:2:1]
M & N [4:4:5]
B[I←I+1] & A[I,J] [47-J MOD 46:J:1]
0 & LOWER[HERE:THISMANY]
```

SEMANTICS.
An arithmetic expression defines a numeric value. Arithmetic expressions may be devided into two categories: simple and conditional.

SIMPLE ARITHMETIC EXPRESSIONS.
A simple arithmetic expression is composed of arithmetic operators and primaries. It is evaluated by performing the indicated arithmetic operations upon the actual numerical values of the primaries from which it is formed. The arithmetic operators are explained in detail on page 4-7, operators and types.

PRIMARIES. Table 4-1 shows the values represented by the primaries in an arithmetic expression.

RESTRICTIONS.
A variable or function designator used as a primary in arithmetic expression must be of an arithmetic type: REAL, INTEGER, or ALPHA.

If the primary is a string, it may not exceed 47 bits in length.

CONCATENATION.
The concatenation form of arithmetic expression provides an efficient method of forming a primary from selected bits of two or more expressions.

The concatenation operator is the ampersand. A concatenation expression is formed by following a primary with & ⟨arithmetic expression⟩ [⟨concatenation⟩]. A concatenate expression may contain any number of concatenation terms. The terms are evaluated from left to right in the expression. Each concatenate operator causes

a concatenated result to be formed. The concatenated result may be the final result of the expression, or a primary.

Table 4-1

Represented Values of Primaries in Arithmetic Expression

Name of Primary	Value Represented
Number	The number itself.
Variable	The current value of the variable.
Partial word designator	The value of the field specified.
Function designator	Value obtained by applying the computing rules of the respective PROCEDURE declaration.
Arithmetic expression in parentheses	The value derived, which must be described in terms of the primaries from which it is formed.
Concatenate expression	The value of the newly formed primary.
String	The numerical value of the string characters.
Assignment statement	Value derived, which must be described in terms of the primaries from which it is formed.

A concatenated result is formed by obtaining the value of the primary and then replacing a portion of it with a field made up of bits from the concatenation term. The field is placed in the primary starting at the bit specified by the ⟨left bit-to⟩

expression. The field is obtained from the concatenation term, starting with the bit designated by the ⟨left bit-from⟩ expression of the concatenation term. The number of bits in the field is determined by the value of the ⟨number of bits⟩ term in the ⟨concatenation⟩ part. If the ⟨concatenation⟩ part does not specify a ⟨left bit-from⟩ part, the right-most (low order) field is used.

RESTRICTIONS.
The ⟨number of bits⟩ term must be an integer between 1 and 47. The sum of the ⟨left bit-to⟩ expression and the ⟨number of bits⟩ term in the field, or the ⟨left bit-from⟩ expression and the ⟨number of bits⟩ term in the field, must not exceed 48.

CONDITIONAL ARITHMETIC EXPRESSIONS.
A conditional arithmetic expression is of the form:

⟨if clause⟩ ⟨arithmetic expression⟩ ELSE ⟨arithmetic expression⟩

The evaluation of the conditional arithmetic expression proceeds as described in the following paragraphs.

The Boolean expression in the ⟨if clause⟩ is evaluated (see page 4-10, Boolean expressions). If the value of the Boolean expression is TRUE, the arithmetic expression following THEN is evaluated and the evaluation of the conditional arithmetic expression is complete.

If the value of the Boolean expression is FALSE, the arithmetic expression following the delimiter ELSE is evaluated, thus completing the evaluation of the expression.

The arithmetic expressions following the delimiters THEN and ELSE

may also be conditional arithmetic expressions. As a result, a conditional arithmetic expression could contain a series of IF clauses in the expression following either or both of the delimiters.

In the case of a conditional arithmetic expression following the delimiter THEN, the Boolean expression(s) in the IF clause(s) are evaluated from left to right as long as they yield a logical value of TRUE. If they all yield a logical value of TRUE, the expression following the last delimiter THEN is executed, thus completing the evaluation of the whole expression. If any of the Boolean expressions yields a logical value of FALSE, the expression following the corresponding delimiter ELSE is executed.

In the case of the conditional arithmetic expression following the delimiter ELSE, the respective Boolean expressions in the IF clauses are evaluated from left to right until a logical value of TRUE is found. Then the value of the succeeding arithmetic expression is the value of the entire arithmetic expression. If no TRUE value is found, the value of the whole expression is that of the expression following the last ELSE.

In nested IF clauses, the first THEN corresponds to the last ELSE, and the innermost THEN to the following (i.e., the innermost) ELSE. The delimiters THEN and ELSE between these extremes follow the logical pattern established, i.e., the next outermost THEN corresponds to the next outermost ELSE, and so on until the innermost THEN-ELSE pair has been matched.

Appropriate positioning of parentheses may serve to establish a different order of execution of operations within an expression.

OPERATORS AND TYPES.

No two operators may be adjacent. Implied multiplication is not allowed.

ARITHMETIC OPERATORS. The operators +, -, ⊗, and / have the conventional mathematical meanings of addition, subtraction, multiplication, and division, respectively. The operator DIV yields a result defined as follows (integer division):

$$Y \text{ DIV } Z = \text{SIGN } (Y/Z) \otimes \text{ENTIER } (\text{ABS } (Y/Z))$$

In the case of the operators /, DIV, and MOD, the operation is undefined if the value of the operand on the right equals zero. The operator MOD produces a result defined as follows (remainder division):

$$Y \text{ MOD } Z = Y - Z \otimes (\text{SIGN } (Y/Z) \otimes \text{ENTIER } (\text{ABS } (Y/Z)))$$

The operator * denotes exponentiation. Its meaning depends on the types and values of operands involved, as shown below (consider Y*Z in table 4-2).

Table 4-2

Meaning of *

	IF Z IS TYPE INTEGER AND			IF Z IS TYPE REAL AND		
	Z > 0	Z = 0	Z < 0	Z > 0	Z = 0	Z < 0
IF Y > 0	Note 1	1	Note 2	Note 3	1	Note 3
IF Y < 0	Note 1	1	Note 2	Note 4	1	Note 4
IF Y = 0	0	Note 4	Note 4	0	Note 4	Note 4

Note 1: $Y * Z = Y \otimes Y \otimes \ldots \otimes Y$ (Z times).

Note 2: Y * Z = the reciprocal of $Y \otimes Y \otimes \ldots \otimes Y$ (Z times).

Note 3: $Y * Z = \text{EXP}(Z \otimes \text{LN}(Y))$.

Note 4: Value of expression is undefined.

ARITHMETIC EXPRESSION TYPES. The type of a value resulting from an arithmetic operation depends upon the types of operands as

well as the arithmetic operators used in obtaining that value, unless that value is undefined.

Table 4-3

Types of Values Resulting from an Arithmetic Operation

OPERAND ON LEFT	OPERAND ON RIGHT	+, -, ⊗	/	DIV	MOD	*
Integer	Integer	Note 3	Real	Integer	Real	Note 1
Integer	Real	Real	Real	Integer	Real	Note 2
Real	Integer	Real	Real	Integer	Real	Note 2
Real	Real	Real	Real	Integer	Real	Note 2

Note 1: If the operand on the right is negative or the absolute value of the result is not less than 2*39, real; otherwise, integer.

Note 2: If the operand on the right is zero, integer; otherwise, real.

Note 3: If the absolute value of the result is less than 2*39, integer; otherwise, real.

PRECEDENCE OF OPERATORS.

In regard to evaluating a simple arithmetic expression, two distinct operations should be understood: the determination of the numerical values of the primaries, and the arithmetic operations involved when combining two operands according to the rules associated with the arithmetic operators.

First, the numerical values of the primaries are determined from left to right, yielding a number of values equal to the number of primaries in the simple arithmetic expression. Next, these values are used two at a time as operands in arithmetic operations, reducing the number of values by one for each operation until all operators have been utilized and a single value remains.

The sequence in which the arithmetic operations are performed is

determined by rules of precedence. Each arithmetic operator has one of three orders of precedence associated with it, as follows:

a. First: *
b. Second: ⊗ / DIV MOD
c. Third: + -

When operators have the same order of precedence, the sequence of operation is determined by the order of their appearance, from left to right.

An expression between parentheses is evaluated by itself and this value is used in subsequent calculations. That is, the normal order of precedence of operators can be overridden by the judicious placement of parentheses.

BOOLEAN EXPRESSIONS.

SYNTAX.

The syntax for ⟨Boolean expression⟩ is as follows:

⟨Boolean expression⟩ ::= ⟨simple Boolean⟩ | ⟨if clause⟩
⟨Boolean expression⟩ ELSE ⟨Boolean expression⟩ |
⟨simple Boolean prefix⟩ ⟨implication prefix⟩
⟨Boolean term prefix⟩ ⟨Boolean factor prefix⟩
⟨secondary prefix⟩ ⟨Boolean assignment⟩

⟨simple Boolean⟩ ::= ⟨simple Boolean prefix⟩ ⟨implication⟩

⟨implication⟩ ::= ⟨implication prefix⟩ ⟨Boolean term⟩

⟨Boolean term⟩ ::= ⟨Boolean term prefix⟩ ⟨Boolean factor⟩

⟨Boolean factor⟩ ::= ⟨Boolean factor prefix⟩ ⟨Boolean secondary⟩

⟨Boolean secondary⟩ ::= ⟨secondary prefix⟩ ⟨Boolean primary⟩

⟨Boolean primary⟩ ::= ⟨logical value⟩ | ⟨relation⟩ | ⟨Boolean partial word operand⟩ | ⟨Boolean partial word operand⟩ . ⟨field description⟩ | ⟨Boolean primary⟩ & ⟨Boolean expression⟩ [⟨concatenation⟩] | ⟨alpha test⟩ | ⟨string relation⟩ | ⟨pointer relation⟩

⟨alpha test⟩ ::= ⟨arithmetic expression⟩ IN ALPHA

⟨string relation⟩ ::= ⟨update pointer⟩ ⟨pointer expression⟩ ⟨relational operator⟩ ⟨update pointer⟩ ⟨pointer expression⟩ FOR ⟨arithmetic expression⟩ | ⟨update pointer⟩ ⟨pointer expression⟩ ⟨relational operator⟩ ⟨string⟩ | ⟨update pointer⟩ ⟨pointer expression⟩ ⟨relational operator⟩ ⟨string⟩ FOR ⟨arithmetic expression⟩

⟨pointer relation⟩ ::= ⟨pointer expression⟩ ⟨equality operator⟩ ⟨pointer expression⟩

⟨equality operator⟩ ::= ≠ | = | NEQ | EQL

⟨Boolean partial word operand⟩ ::= ⟨Boolean variable⟩ | ⟨Boolean function designator⟩ | (⟨Boolean expression⟩)

⟨simple Boolean prefix⟩ ::= ⟨empty⟩ | ⟨simple Boolean⟩ EQV

⟨implication prefix⟩ ::= ⟨empty⟩ | ⟨implication⟩ IMP

⟨Boolean term prefix⟩ ::= ⟨empty⟩ | ⟨Boolean term⟩ OR

⟨Boolean factor prefix⟩ ::= ⟨empty⟩ | ⟨Boolean factor⟩ AND

⟨secondary prefix⟩ ::= ⟨empty⟩ | NOT

⟨Boolean assignment⟩ ::= ⟨Boolean variable⟩ ⟨replacement operator⟩ ⟨Boolean expression⟩

⟨Boolean variable⟩ ::= ⟨variable⟩

⟨Boolean function designator⟩ ::= ⟨function designator⟩

⟨relation⟩ ::= ⟨simple arithmetic expression⟩ ⟨relational operator⟩ ⟨arithmetic expression⟩

⟨field description⟩ ::= [⟨left bit of field⟩ : ⟨bits in field⟩]

⟨left bit of field⟩ ::= ⟨unsigned integer⟩

⟨bits in field⟩ ::= ⟨unsigned integer⟩

Examples:

Boolean Expressions:

TRUE

NOT A ≠ 0

Q.[16:1] AND GATE[1,2]

A = C AND (IF B = 4 THEN TRUE ELSE FALSE) OR GATE[1,2]

IF B = 4 THEN TRUE EQV GATE [1,2] ELSE Q.[16:1]

Implications:

TRUE

GATE[1,2]

NOT A ≠ C IMP GATE[1,2]

Boolean Terms:

TRUE

NOT A ≠ C

GATE[1,2]

A ≠ C AND (IF B = 4 THEN TRUE ELSE FALSE) OR GATE[1,2]

Boolean Factors:

GATE[1,2]

NOT A ≠ C

Q.[16:1] AND GATE[1,2]

Boolean Primaries:

TRUE

DIODE

GATE[1,2]

J(A,B + 2,GATE[1,2])

A ≠ C

(IF A ≠ C THEN TRUE ELSE FALSE)

Q.[16:1]

(DIODE ← GATE[1,2])

Boolean Secondaries:

TRUE

NOT A ≠ C

SEMANTICS.

A Boolean expression defines a logical value.

Boolean expressions can be divided into two categories: simple Boolean expressions and conditional Boolean expressions.

SIMPLE BOOLEAN EXPRESSIONS.

A simple Boolean expression is formed by logical operators (see page 4-16) and Boolean primaries. It is evaluated by carrying out the operations indicated by the logical operators upon the associated Boolean primaries. The evaluation of a simple Boolean expression is carried out according to the rules of precedence defined for the logical operators (see page 4-16).

The value which results upon evaluation of a simple Boolean expression depends upon the primary or primaries which are used to form the expression. Table 4-4 shows the values represented by the primaries in a Boolean expression.

CONCATENATION.

The concatenation form of Boolean expression is identical to that of arithmetic expression (see page 4-4) except that the resulting value is treated as type Boolean. In other words, only the low order bit (bit 0) is significant unless a type conversion function designator is used.

Table 4-4

Values Represented by Primaries in a Boolean Expression

Name of Primary	Value Represented
Logical value	TRUE or FALSE.
Boolean variable	The current value of the variable.
Partial word designator	The value of the field specified.
Function designator	The value obtained by applying the computing rules of the respective PROCEDURE declaration.
Relation	The value obtained by testing the simple arithmetic expressions against each other, according to the operation of the specific relational operator involved.
Boolean expression enclosed in parentheses	The value derived, which must be described in terms of the Boolean primaries from which it is formed.
Concatenate expression	The value of the newly formed primary.

BOOLEAN PRIMARY.

A ⟨string relation⟩ Boolean primary compares two strings according to the BCL collating sequence.* The arithmetic expression specifies the number of characters to compare. If a literal string

* This will cause any comparisons other than = or ≠ to be incompatible with the B 6500.

follows the relational operator and FOR (arithmetic expression) is present, then the string is treated as specified for source strings in the string transfer statement (see page 6-11). Otherwise, the string characters are used for comparison one time only and the number of characters compared is equal to the number of characters in the string.

A ⟨pointer relation⟩ Boolean primary determines whether or not two pointer expressions refer to the same character position of the same string. If the character sizes of the two pointers are not equal, the comparison will always be unequal.

The ⟨alpha test⟩ construct has the value TRUE if a given character is a letter or digit. The character is the value of the arithmetic expression.

CONDITIONAL BOOLEAN EXPRESSIONS.

The simplest form of the conditional Boolean expression occurs when the IF clause contains a simple Boolean expression. The evaluation of the conditional Boolean expression in this case proceeds as follows. The simple Boolean expression of the IF clause is evaluated according to the methods described previously (page 4-13, simple Boolean expressions). If the resulting logical value is TRUE, the Boolean expression following the delimiter THEN is evaluated, thus completing the evaluation of the conditional Boolean expression. If the logical value produced in the IF clause is FALSE, the evaluation of the conditional Boolean expression is completed by evaluating the Boolean expression following the delimiter ELSE.

The Boolean expression in the IF clause, or the one following the delimiter THEN or the delimiter ELSE, or all three, can be conditional Boolean expressions. In this event, any of the IF clauses consist of a series of IF clauses. Such a construct is said to be nested. The evaluation of such nested expressions occurs in the same manner as that of analogous constructs in arithmetic expressions.

TYPES.

A variable or function designator used as a Boolean primary must be of type BOOLEAN (see page 7-2, type declarations, and page 8-5, special rules of typed procedures), with the exception of the constituents of relations and those quantities which are under the influence of type transfer functions (see page 3-7, type transfer functions).

RELATIONAL AND LOGICAL OPERATORS.

Two types of operators are defined for Boolean expressions: relational and logical.

RELATIONAL OPERATORS. The relational operators denote the following relations:

a. < or LSS (is less than).
b. ≤ or LEQ (is less than or equal to).
c. = or EQL (is equal to).
d. ≥ or GEQ (is greater than or equal to).
e. > or GTR (is greater than).
f. ≠ or NEQ (is not equal to).

A relation is evaluated by comparing the values of the two simple arithmetic expressions as designated by the relational operator. If the relation is satisfied, the value of the Boolean primary is TRUE; otherwise, it is FALSE.

LOGICAL OPERATORS. The operation of the logical operators is defined in the following truth table.

Table 4-5

Logical Operators Truth Table

Operand A	Operand B	NOT A	A AND B	A OR B	A IMP B	A EQV B
TRUE	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE
FALSE	TRUE	TRUE	FALSE	TRUE	TRUE	FALSE
FALSE	FALSE	TRUE	FALSE	FALSE	TRUE	TRUE

PRECEDENCE OF OPERATORS.

The sequence of operations within a simple Boolean expression is determined by the precedence of the operators (generally from left to right, with the additional rules shown below). When operators are of the same order of precedence, the sequence of operations is determined by the left-to-right order of appearance of the operators. The following specific rules of precedence are defined:

a. First: Arithmetic expressions, according to the rules given on page 4-9.

b. Second: Relational operators ($<$, $\leq$, $=$, $>$, $\geq$, $\neq$)

c. Third: NOT

d. Fourth: AND

e. Fifth: OR

f. Sixth: IMP

g. Seventh: EQV

A Boolean expression contained in parentheses is evaluated by itself; this value is then used in any subsequent evaluation. Therefore, the desired order of execution of operations within an expression can always be effected by appropriate positioning of parentheses.

DESIGNATIONAL EXPRESSIONS.

SYNTAX.

The syntax for ⟨designational expression⟩ is as follows:

⟨designational expression⟩ ::= ⟨label⟩ | ⟨switch designator⟩ | ⟨if clause⟩ ⟨designational expression⟩ ELSE ⟨designational expression⟩

⟨label⟩ ::= ⟨identifier⟩

⟨switch designator⟩ ::= ⟨switch identifier⟩ [⟨subscript⟩]

⟨switch identifier⟩ ::= ⟨identifier⟩

Examples:

Designational Expressions:

```
START
CHOOSEPATH [I + 2]
(START)
IF K = 1 THEN SELECT [2] ELSE START
```

Simple Designational Expressions:

```
START
SELECT [2]
(START)
```

Switch Designators:

```
SELECT [2]
CHOOSEPATH [I + 3]
```

SEMANTICS.
A designational expression defines a label. As is true of other expressions, designational expressions may be differentiated as simple designational and conditional designational expressions.

SIMPLE DESIGNATIONAL EXPRESSIONS. The process of evaluating a simple designational expression depends upon the constructs from which it is formed. If a simple designational expression is a label, the value of the expression is the label. When a simple designational expression is a switch designator, the actual numerical value of the subscript expression (see page 3-3) designates one of the elements in the switch list. The element selected may be any form of simple designational expression which is evaluated as stated above, or it may be a conditional designational expression which is evaluated as stated below.

A switch designator is a switch identifier and a subscript. A switch identifier refers to a set of designational expressions. A switch identifier with a subscript refers to a designational expression.

The elements of a switch identifier correspond to the ordinal numbers from 1 to N (where N is the number of elements in the set). If the value of the subscript of a switch designator is not an integer, the subscript is rounded:

Integral subscript value = ENTIER (value of subscript + 0.5)

If N is the number of elements referred to by a switch identifier and the subscript value is not between 1 and N, the switch designator is not defined.

If a simple designational expression is formed from a designational expression in parentheses, the latter is evaluated according to the applicable rules.

CONDITIONAL DESIGNATIONAL EXPRESSIONS. The evaluation of a conditional designational expression proceeds as follows. The Boolean expression contained in the IF clause is evaluated (see page 4-10, Boolean expressions). If a logical value of TRUE results, the designational expression following the IF clause is evaluated, thus completing the evaluation of the conditional designational expression. If the logical value produced by the IF clause is FALSE, the designational expression following the delimiter ELSE is evaluated, thereby completing the evaluation of the designational expression.

Since the designational expressions following the delimiters THEN and ELSE, or both, can be conditional designational expressions, the analysis of the operation of a designational expression becomes recursive in a manner similar to that of the conditional arithmetic and Boolean expressions. In the case of a designational expression, however, the result produced is always a label.

POINTER EXPRESSIONS.

SYNTAX.

The syntax for ⟨pointer expression⟩ is as follows:

⟨pointer expression⟩ ::= ⟨if clause⟩ ⟨pointer expression⟩ ELSE ⟨pointer expression⟩ | ⟨simple pointer expression⟩

⟨simple pointer expression⟩ ::= ⟨pointer primary⟩ ⟨skip⟩ | ⟨pointer assignment⟩

⟨pointer primary⟩ ::= ⟨pointer designator⟩ | ⟨pointer identifier⟩ | (⟨pointer expression⟩)

⟨skip⟩ ::= ⟨empty⟩ | ⟨adding operator⟩ ⟨ arithmetic expression⟩

⟨pointer identifier⟩ ::= ⟨identifier⟩

⟨pointer designator⟩ ::= POINTER (⟨pointer parameters⟩)

⟨pointer parameters⟩ ::= ⟨array part⟩ | ⟨array part⟩ ⟨parameter delimiter⟩ ⟨character size⟩

⟨array part⟩ ::= ⟨array row⟩ | ⟨subscripted variable⟩

⟨array row⟩ ::= ⟨array identifier⟩ [⟨row designator⟩]

⟨character size⟩ ::= 6 | 8

⟨row designator⟩ ::= * | ⟨row⟩ , *

⟨row⟩ ::= ⟨arithmetic expression⟩ | ⟨row⟩ , ⟨arithmetic expression⟩

Examples:

```
NEXTCHAR ← POINTER (ACCUM[1],6) + 1
CHARPOSITION ← POINTER (OUTARRAY[PAGENO, LINENO, *],8)
```

```
CHARPOSITION:=CHARPOSITION-NCHARS
CARDCOL:=POINTER (CARDARRAY[0]) + 6
CHARPOSITION ← IF PERCENT AND NOT ENDOFCARD THEN
               CHARPOSITION + 1 ELSE CHARPOSITION
```

SEMANTICS.

A pointer expression defines a character position within an array.

An identifier used as a pointer primary must be of type POINTER.

If a pointer expression is enclosed in parentheses, it is evaluated first and its value used as a primary.

If ⟨skip⟩ is not empty, the pointer value is adjusted by L characters to the right or left, where L is the absolute value of the arithmetic expression. If the adding operator is +, skipping is to the right. If the operator is -, skipping is to the left.

POINTER DESIGNATORS.

The following pointer designator constructs are used to associate pointer identifiers with array row character positions.

POINTER (A,L)	Yields a pointer value pointing to A. A is an array row or a subscripted variable. L is a character size in bits (6 or 8).
POINTER (A)	Same as POINTER (A,6).

Associated with a pointer expression is a "pointer level" defined as follows:

a. ⟨pointer identifier⟩. The pointer level determined by the declaration of the pointer identifier (see pointer declarations, page 7-6).

b. ⟨pointer assignment⟩. The level of pointer variable preceding the assignment operator.

c. ⟨pointer primary⟩ ⟨skip⟩. The level of the pointer primary.

d. ⟨if clause⟩ ⟨pointer expression⟩ ELSE ⟨pointer expression⟩. The greater level of the two expressions.

The pointer level of a pointer expression is used by the compiler to determine the validity of the pointer assignments and pointer update operations (see pointer assignment, page 6-7; Boolean expressions, page 4-10; and string scan and transfer statements, page 6-11).

SECTION 5
PROGRAMS, BLOCKS, AND COMPOUND STATEMENTS

GENERAL.

SYNTAX.

The syntax for ⟨program⟩ is as follows:

⟨program⟩ ::= ⟨block⟩ . ⟨space⟩ | ⟨compound statement⟩ . ⟨space⟩

⟨block⟩ ::= ⟨block head⟩ ; ⟨compound tail⟩

⟨block head⟩ ::= BEGIN ⟨declaration⟩ | ⟨block head⟩ ; ⟨declaration⟩

⟨compound tail⟩ ::= ⟨statement⟩ END | ⟨statement⟩ ; ⟨compound tail⟩

⟨compound statement⟩ ::= BEGIN ⟨compound tail⟩

Examples:

The syntactical structure of the compound statement and the block can be illustrated in the following manner.

Given:

S = statement

S_c = compound statement

L = label

D = declaration

B = block

Then:

Compound Statement:

S_c = BEGIN S;S;S;...S END

= L:S_c

Block:

B = BEGIN D;D;...;D;S;S;...;S END

= L:B

Because of the syntactical definition of statements (section 6), S in the above examples could itself be a compound statement or a block.

SEMANTICS.

A series of statements which are common to each other by virtue of the defining declarations, and which are bounded by the bracket symbols BEGIN and END, constitutes the active elements of a block. Every block automatically introduces a new level of nomenclature. Therefore, any identifier occurring within the block may, through a suitable declaration (see section 7, declarations), be specified to be local to the block in question. Such a declaration means that:

a. The entity represented by the identifier inside the block will not be recognized by that identifier outside the block.

b. Conversely, any entity represented by the identifier outside the block will not be recognized by that identifier inside the block.

An identifier occurring within an inner block and not declared within that block is "global" to it; that is, the identifier represents the same entity inside the block and in the level or levels immediately outside it, up to and including the level at which the identifier is declared.

Since a statement within a block may itself be a block, the concepts of local and global to a block must be understood recursively. Thus, an identifier which is global to block A may or may not be global to block B in which block A is one statement.

NESTED BLOCKS. Block B is said to be nested in block A if block B is a statement in the compound tail of block A.

DISJOINT BLOCKS. Block A and block B are said to be disjoint if neither is a statement in the compound tail of the other.

SECTION 6
STATEMENTS

GENERAL.

SYNTAX.

The syntax for ⟨statement⟩ is as follows:

⟨statement⟩ ::= ⟨conditional statement⟩ | ⟨unconditional statement⟩

⟨conditional statement⟩ ::= ⟨unlabeled conditional statement⟩ | ⟨label⟩ : ⟨conditional statement⟩

⟨unlabeled conditional statement⟩ ::= ⟨if clause⟩ ⟨statement⟩ | ⟨if clause⟩ ⟨unconditional statement⟩ ELSE ⟨conditional statement⟩ | ⟨iteration clause⟩ ⟨conditional statement⟩

⟨unconditional statement⟩ ::= ⟨unlabeled unconditional statement⟩ | ⟨label⟩ : ⟨unconditional statement⟩

⟨unlabeled unconditional statement⟩ ::= ⟨iteration clause⟩ ⟨unconditional statement⟩ | ⟨compound statement⟩ | ⟨block⟩ | ⟨go to statement⟩ | ⟨procedure statement⟩ | ⟨I/O statement⟩ | ⟨do statement⟩ | ⟨case statement⟩ | ⟨string transfer statement⟩ | ⟨fill statement⟩ | ⟨assignment statement⟩ | ⟨string scan statement⟩ | ⟨if clause⟩ ⟨unconditional statement⟩ ELSE ⟨unconditional statement⟩

SEMANTICS.

Statements are the units of operation of the language. A statement denotes an action to be performed.

The definition of statement is recursive because statements may be grouped in compound statements and blocks.

A conditional statement causes certain statements to be executed or skipped depending upon the value produced by a Boolean expression.

UNLABELED CONDITIONAL STATEMENTS.

SYNTAX.

One form of ⟨unlabeled conditional statement⟩ shown in the syntax on page 6-1 is:

⟨unlabeled conditional statement⟩ ::= ⟨if clause⟩ ⟨statement⟩

In this form, the statement following the sequential operator THEN is executed if the value of the preceding Boolean expression is TRUE. Otherwise, the statement is ignored.

Another form of ⟨unlabeled conditional statement⟩ is:

⟨unlabeled conditional statement⟩ ::= ⟨if clause⟩ ⟨unconditional statement⟩ ELSE ⟨conditional statement⟩

In this form, if the value of the Boolean expression is TRUE, the unconditional statement following the sequential operator ELSE is ignored. If the Boolean expression evaluates FALSE, the conditional statement following ELSE is executed and the unconditional statement following THEN is ignored.

UNLABELED UNCONDITIONAL STATEMENTS.

SYNTAX.

One form of ⟨unlabeled unconditional statement⟩ on page 6-1 is:

⟨unlabeled unconditional statement⟩ ::= ⟨if clause⟩ ⟨unconditional statement⟩ ELSE ⟨unconditional statement⟩

In this form, if the value of the Boolean expression is TRUE, the unconditional statement following THEN is executed and the unconditional statement following ELSE is ignored. If the Boolean expression evaluates FALSE, the unconditional statement following ELSE is executed and the unconditional statement following THEN is ignored.

Since the definitions of conditional statements and unconditional statements are recursive, nested conditional statements may occur.

A GO TO statement may lead to a labeled statement within a conditional statement. Subsequent action is the same as would occur if the conditional statement were entered at the beginning and subsequent evaluation of the Boolean expression led to the labeled statement.

An unlabeled unconditional statement of the form ⟨empty⟩ executes no operation. This form may serve to place a label.

GO TO STATEMENT.

SYNTAX.

The syntax for ⟨go to statement⟩ is as follows:

⟨go to statement⟩ ::= GO TO ⟨designational expression⟩ |
GO ⟨designational expression⟩

Examples:

```
GO TO START
GO TO SELECT[2]
GO TO IF K = 1 THEN SELECT [2] ELSE START
```

SEMANTICS.

The GO TO statement transfers control to the label that is the value of the designational expression.

If the designational expression is undefined, control continues to the statement which would have been the successor of the GO TO statement if the GO TO statement were empty.

PROCEDURE STATEMENT.

SYNTAX.

The syntax for ⟨procedure statement⟩ is as follows:

⟨procedure statement⟩ ::= ⟨procedure identifier⟩ ⟨actual parameter part⟩

Examples:

ALGORITHM123 (A + 2)
ALGORITHM546 (A + 2) "AVERAGE PLUS TWO"(CALCRULE)

SEMANTICS.
A procedure statement causes a previously defined procedure to be executed.

The actual parameter list of the procedure statement must have the same number of entries as the formal parameter list of the procedure declaration heading.

Formal and actual parameters must correspond in number, type, and kind of quantities. The correspondence is obtained by taking the entries of these two lists in the same order.

DO STATEMENT.
SYNTAX.
The syntax for ⟨do statement⟩ is as follows:

⟨do statement⟩ ::= DO ⟨statement⟩ UNTIL ⟨Boolean expression⟩

Example:

DO A[I] ← I UNTIL I ← I - 1 = 0

SEMANTICS.
The DO statement causes the statement to be executed and then the Boolean expression to be evaluated. If this value is FALSE, the statement is executed again and the Boolean expression re-evaluated. The sequence continues until the Boolean expression evaluates TRUE. A concise description is:

LD: S_{do}; IF NOT BE THEN GO TO LD

where:

LD - label

S_{do} - DO statement

BE - Boolean Expression

CASE STATEMENT.

SYNTAX.

The syntax for ⟨case statement⟩ is as follows:

⟨case statement⟩ ::= CASE ⟨arithmetic expression⟩ OF
⟨case body⟩

⟨case body⟩ ::= ⟨compound statement⟩

Examples:

```
CASE I OF
BEGIN (statement0);
      (statement1);
          .
          .
          .
          .
(statementN) END;
```

SEMANTICS.

The CASE statement provides the programmer the means for selective execution of one of a series of statements.

At execution time, the value of the arithmetic expression determines which of the ⟨statement⟩s will be executed within the compound statement.

The sequence of action is:

a. The arithmetic expression is evaluated.

b. If the value is not an integer, the value is rounded:

Integer value = ENTIER (value + 0.5).

c. The value is used as the ordinal number of a statement in the case body.

d. If the value lies outside the range of 0 to N-1 (where N is the number of statements in the case body), an error interrupt occurs.

e. Otherwise, the selected statement is executed.

f. At the end of the selected statement, control is transferred to the point beyond the end of the case body unless the selected statement causes a transfer of control outside the scope of the case statement.

ASSIGNMENT STATEMENT.

SYNTAX.

The syntax for ⟨assignment statement⟩ is as follows:

⟨assignment statement⟩ ::= ⟨pointer assignment⟩ |
⟨arithmetic variable⟩ ⟨partial word part⟩
⟨replacement operator⟩ ⟨arithmetic expression⟩ |
⟨Boolean variable⟩ ⟨partial word part⟩ ⟨replacement operator⟩ ⟨Boolean expression⟩

⟨partial word part⟩ ::= ⟨empty⟩ | . ⟨field part⟩

⟨pointer assignment⟩ ::= ⟨pointer identifier⟩ ⟨replacement operator⟩ ⟨pointer expression⟩

Examples:

A ← A + 1

Q.[30:1] ← P ≥ R

P ← "RESULT"

```
A ← B ← C ← D ← 1
X.[47:1] ← X ← Z ← 0
```

SEMANTICS.

The assignment statement causes the expression to the right of the replacement operator to be evaluated. The value of the expression is assigned to the variable or field on the left.

A form of arithmetic expression is the ⟨arithmetic assignment⟩. A form of Boolean expression is the ⟨Boolean assignment⟩. A form of pointer expression is the pointer assignment.

The action of the assignment statement is as follows:

a. The ⟨variable⟩ is evaluated.

b. The ⟨partial word part⟩, if not empty, is evaluated.

c. The expression following the replacement operator is evaluated.

d. The value of the expression is assigned to the variable (or to the specified part thereof). In an arithmetic assignment, the appropriate implicit type conversion (integer and real) is performed as required.

The ⟨pointer assignment⟩ is invalid if the pointer level of the pointer expression exceeds the pointer level of the pointer identifier (see pointer expressions, page 4-20, and pointer declarations, page 7-6).

TYPES.

All variables in the left part list must be either exclusively of type BOOLEAN or of an arithmetic type, i.e., REAL, INTEGER, or ALPHA (which is treated as type REAL). (See page 7-2, type declarations.)

If the variables are of type BOOLEAN, the value to be assigned must be that of a Boolean expression.

If there is a difference between the declared type of the left part variable and the value to be assigned to it, or the left part variables are of different arithmetic types, the Compiler will reconcile the differences, but this procedure may cause a change (rounding to integer) in the value assigned.

The following rules apply:

a. If the left part list is of type REAL and the expression value is of type INTEGER, the value is stored unchanged.

b. If the left part list is of type INTEGER and the expression value is of type REAL, the transfer function ENTIER (E + 0.5), where E is the value of the expression, is automatically invoked and the value obtained is stored.

c. If the left part list contains variables of different types, assignment of the value is executed from right to left. If, during this process, a real number is transferred to integer, this integer value is assigned to all following variables at the left of the integer variable, regardless of their type.

RESTRICTIONS.
Assignment to a procedure identifier may occur only within the body of a procedure defining the value of a function designator.

ITERATION CLAUSE.
SYNTAX.
The syntax for ⟨iteration clause⟩ is as follows:

⟨iteration clause⟩ ::= FOR ⟨variable⟩ ⟨replacement operator⟩ ⟨for list⟩ DO | WHILE ⟨Boolean expression⟩ DO | THRU ⟨arithmetic expression⟩ DO

⟨for list⟩ ::= ⟨for list element⟩ | ⟨for list⟩ , ⟨for list element⟩

⟨for list element⟩ ::= ⟨initial part⟩ ⟨increment part⟩

⟨initial part⟩ ::= ⟨arithmetic expression⟩

⟨increment part⟩ ::= ⟨empty⟩ | ⟨step part⟩ UNTIL ⟨arithmetic expression⟩ | ⟨step part⟩ WHILE ⟨Boolean expression⟩ | WHILE ⟨Boolean expression⟩

⟨step part⟩ ::= STEP ⟨arithmetic expression⟩

Examples:

FOR Statements:

```
FOR I ← A + 2 DO BETA ← I + BETA
FOR K ← A + 2, 1 STEP 1 UNTIL N DO P [K] ← R [K]
```

FOR Clauses:

```
FOR I ← A + 2 DO
FOR K ← A + 2, 1 STEP 1 UNTIL N DO
```

FOR-Lists:

```
A + 2
A + 2, 1 STEP 1 UNTIL N, A + 2 WHILE A > B, 1 STEP 1
  WHILE A > B
```

FOR-List Elements:

```
A + 2
1 STEP 1 UNTIL N
A + 2 WHILE A > B
1 STEP 1 WHILE A > B
```

SEMANTICS.

The iteration clause provides a means of forming loops in a program. (See statements, section 6.)

The FOR ⟨variable⟩ ⟨replacement operator⟩ ⟨for list⟩ DO form of the iteration clause is evaluated as follows:

a. The initial value assigned to the variable (referred to as the controlled variable) is that of the left-most arithmetic expression in the for list. Subsequent values of the controlled variable depend on the elements of the for list and associated action.

b. The for list elements provide rules for obtaining values to be assigned to the controlled variable; also, the for list elements furnish the tests to be made to decide whether or not to execute the statement following DO.

c. Control is transferred beyond the for clause and its affiliated statement when:

   1) A test is failed, or

   2) An appropriate GO TO statement is executed in the affiliated statement, or

   3) The for list is exhausted.

The WHILE ⟨Boolean expression⟩ DO form of the iteration clause is evaluated as follows:

a. The Boolean expression is evaluated.

b. If the value is true, the statement following is executed.

c. The sequence is repeated until:

1) The value is FALSE, or

2) A change of control is executed in the statement following.

The THRU ⟨arithmetic expression⟩ DO form of the iteration clause is evaluated as follows:

a. The arithmetic expression is evaluated.

b. The statement following the DO is executed the number of times indicated by the initial value of the arithmetic expression.

STRING TRANSFER STATEMENTS.

SYNTAX.

The syntax for ⟨string transfer statement⟩ is as follows:

⟨string transfer statement⟩ ::= REPLACE ⟨destination⟩ BY ⟨source list⟩

⟨string scan statement⟩ ::= SCAN ⟨source⟩ ⟨scan part⟩

⟨source list⟩ ::= ⟨source part⟩ | ⟨source list⟩ , ⟨source part⟩

⟨source part⟩ ::= ⟨source⟩ ⟨transfer part⟩ | ⟨string⟩ ⟨optional unit count⟩ | ⟨arithmetic expression⟩ ⟨optional unit count⟩ | ⟨output convert⟩

⟨scan part⟩ ::= FOR ⟨maxcount⟩ ⟨condition⟩ | ⟨condition⟩

⟨transfer part⟩ ::= ⟨scan part⟩ | ⟨unit count⟩

⟨unit count⟩ ::= FOR ⟨arithmetic expression⟩ ⟨units⟩

⟨condition⟩ ::= WHILE ⟨relational operator⟩ ⟨arithmetic expression⟩ | UNTIL ⟨relational operator⟩

⟨arithmetic expression⟩ | WHILE IN ALPHA |
UNTIL IN ALPHA

⟨optional unit count⟩ ::= ⟨empty⟩ | ⟨unit count⟩

⟨maxcount⟩ ::= ⟨update count⟩ ⟨arithmetic expression⟩

⟨destination⟩ ::= ⟨update pointer⟩ ⟨pointer expression⟩

⟨source⟩ ::= ⟨update pointer⟩ ⟨pointer expression⟩

⟨update pointer⟩ ::= ⟨empty⟩ | ⟨pointer identifier⟩ :

⟨update count⟩ ::= ⟨empty⟩ | ⟨simple variable⟩ :

⟨units⟩ ::= ⟨empty⟩ | WORDS

⟨output convert⟩ ::= ⟨arithmetic expression⟩ FOR
⟨arithmetic expression⟩ DIGITS

Examples:

```
SCAN CARDCOL:CARDCOL ← POINTER (BUFFARRAY [0]) FOR
  COUNT: 80-COUNT WHILE ≠ " "

REPLACE ID ← POINTER (ACCUM [1]) + 3 BY CARDCOL:
  CARDCOL FOR COUNT:63 WHILE IN ALPHA
```

SEMANTICS.

A string transfer statement transfers characters or words from one or more sources to a destination. If a character transfer is specified, the pointer expressions must refer to characters of the same size.

There are several forms of the string transfer statement, depending upon the form of the ⟨source part⟩.

a. ⟨source⟩ ⟨unit count⟩
   Used to transfer a given number of words or characters.

b. ⟨source⟩ FOR ⟨maxcount⟩ ⟨condition⟩
   This statement transfers characters either until a

maximum count is exhausted, or until (or while) a condition is satisfied. The condition may specify a relation between source characters and a given character (arithmetic expression), or it may specify membership of source characters in the ALPHA character set. The ALPHA character set consists of ⟨letters⟩ and ⟨digits⟩.

c. ⟨source⟩ ⟨condition⟩

This statement does the same as b above, except the maximum count is not given; 8184 is assumed.

d. ⟨string⟩ ⟨unit count⟩

This statement transfers a string under control of a count.

1) If the string represents fewer than 48 bits, the string is extended to 48 bits by concatenating it with itself an appropriate number of times, and the characters of this 48-bit string are transferred repeatedly until the unit count is exhausted; otherwise,

2) The string represents more than 47 bits. In this case, the string is transferred until the count is exhausted. If the count exceeds the string length, the results are undefined.

e. ⟨string⟩

This statement transfers the characters of a string exactly once.

f. ⟨arithmetic expression⟩ ⟨unit count⟩

This statement is the same as e above, except the value is assumed to be a 48-bit string. Its characters are assumed to be equal in size to those of the destination string.

g. ⟨arithmetic expression⟩

This statement transfers the 48 bits of the value of the arithmetic expression exactly once.

h. ⟨output convert⟩

This statement converts the value of the first arithmetic expression into the number of digits specified by the second arithmetic expression and places them into the string. The value of the second expression must be less than or equal to 8. If the converted value requires more than the specified number of digits, the right-most digits will be used.

A string scan statement is identical to statement b or c above, except that no destination is required and no character transfer takes place. It merely examines a given string.

At the completion of a string scan or string transfer statement, certain updated values are available, and are saved as determined by the presence of the optional update constructs. If N characters have been transferred (or scanned), the update pointer values are the original values of the pointer expressions + N. The updated count value is the original value of the count arithmetic expression - N.

The arithmetic expression in ⟨unit count⟩ specifies a word count if WORDS is present; otherwise, it specifies a character count.

Whenever a non-empty update pointer is specified, the statement is invalid if the pointer level of the pointer expression exceeds the pointer level of the pointer identifier preceding the colon (see pointer expressions, page 4-20, and pointer declarations, page 7-6).

Scan and replace statements with a "scan part" may not be used on eight-bit strings. String comparison will also not accept eight-bit characters, although these strings can be compared using the REAL string intrinsic.

I/O STATEMENTS.

SYNTAX.

The syntax for ⟨I/O statement⟩ is as follows:

⟨I/O statement⟩ ::= ⟨read statement⟩ | ⟨write statement⟩ |
⟨space statement⟩ | ⟨close statement⟩ |
⟨lock statement⟩ | ⟨rewind statement⟩ |
⟨fcr statement⟩

⟨fcr statement⟩ ::= ⟨file identifier⟩. ACCESS ← ⟨access media⟩

⟨access media⟩ ::= RANDOM | SERIAL | UPDATE

SEMANTICS.

Input/output statements cause values to be communicated to and from a program and provide programmatic control of most files and their corresponding I/O units. Disk files and data communications files are handled by the disk and data communications I/O statements, respectively.

The ⟨fcr statement⟩ applies only to disk files and is executed only if the last reference to the file was a CLOSE, REWIND, or LOCK statement. It has two functions:

a. It sets the access mode as specified.

b. When it is executed after a LOCK statement, conditions are established such that the file placed in the disk directory by the LOCK statement remains addressable through the same file declaration.

Example.

```
FILE A DISK SERIAL [20:1000] (2,10,150);
WRITE(A,FMT1,LST1);
LOCK(A);  %PUTS A IN DIRECTORY
A. ACCESS←RANDOM;
READ(A[I],FMT2,LST2);  %THE FILE A IN THE DIRECTORY IS ACCESSED
```

The remaining I/O statements are the same as those described in the reference manual for B 5500 ALGOL.

READ STATEMENTS.

SYNTAX.

The syntax for ⟨read statement⟩ is as follows:

```
⟨read statement⟩ ::= READ ⟨direction⟩ (⟨input parameters⟩)
          ⟨action labels⟩

⟨direction⟩ ::= ⟨empty⟩ | REVERSE

⟨input parameters⟩ ::= ⟨file part⟩ ⟨buffer release⟩,
          ⟨format and list part⟩ | ⟨file part⟩
          ⟨buffer release⟩ | ⟨file part⟩
          ⟨buffer release⟩ , ⟨free-field part⟩

⟨file part⟩ ::= ⟨file identifier⟩ | ⟨switch file designator⟩

⟨buffer release⟩ ::= ⟨empty⟩ | [NO]

⟨format and list part⟩ ::= ⟨format⟩ | ⟨format⟩ , ⟨list⟩ |
          ⟨format⟩ , ⟨list identifier⟩ | * , ⟨list⟩ | *,
          ⟨list identifier⟩ | ⟨arithmetic expression⟩ ,
          ⟨array row⟩

⟨format⟩ ::= ⟨format identifier⟩ | ⟨switch format designator⟩

⟨free-field part⟩ ::= / , ⟨list⟩ | / , ⟨list identifier⟩

⟨action labels⟩ ::= [⟨end-of-file label⟩ : ⟨parity label⟩] |
          [⟨end-of-file label⟩] | [: ⟨parity label⟩] | ⟨empty⟩

⟨end-of-file label⟩ ::= ⟨designational expression⟩

⟨parity label⟩ ::= ⟨designational expression⟩
```

Examples:

```
READ (FILEID, FMT, LISTID) [LEOF]
READ (FILEID [NO], FMT, LISTID)
READ REVERSE (FILEID, FMT, A, B, C, ARA[1]) [:LPAR]
READ (FILEID, *, LISTID)
READ (FILEID, X + Y, ARA[*]) [LEOF:LPAR]
READ (FILEID,FMT)
READ REVERSE(FILEID,50,ARA2[1,*]) [:LPAR]
READ (FILEID)
READ (FILEID, /, FOR I ← 0 STEP 1 UNTIL 16 DO A [I])
READ (FILEID[IF X > N THEN 0 ELSE 1], 50, AES[*])
READ (SPO, FRMT, LST)
READ (SPO, /, LST)
```

SEMANTICS.

The READ statement causes values to be assigned to program variables. It can also place information in strings defined in the FORMAT declaration.

Direction must be indicated only when magnetic tape is to be read in the reverse direction. In all other cases, the direction part of the statement must be empty.

The file part specifies which file is to be read.

The buffer release indicates whether the input buffer is to be refilled after it has been read and edited. If [NO] is used, the buffer is not refilled, and the same buffer will be the next one accessed.

The format and list part specifies the action to be taken on input data.

A READ statement with an empty format and list part causes one logical record to be passed without being read; i.e., such a statement acts as a SPACE (FILE, 1) statement.

A format part without a list part indicates that the referenced FORMAT declaration contains a string into which corresponding characters of the input data are to be placed; the string in the FORMAT declaration is replaced by the string in the input data.

A format part with a list or list identifier designates that the input data is to be edited according to the specifications of the referenced FORMAT declaration and assigned to the variables of the list.

The symbol *, together with a list or list identifier, specifies that the input data is to be processed as full words, and that it is to be assigned to the variables of the referenced list without being edited. The number of words read is determined by the number of variables in the list or the maximum record size, whichever is smaller.

An arithmetic expression with an array row designator specifies that input data is to be processed as full words, and that it is to be assigned to the elements of the designated array row without being edited. The number of words read is determined by the number of elements in the array row, the buffer size, or the value of the arithmetic expression, whichever is smallest.

The symbol / specifies free-field input. Such input does not require a FORMAT declaration to provide specifications for data. Editing specifications in this case are determined by the format of the data itself (see Free-Field Data page 6-19).

Action labels provide a means of transferring control from a READ (or SPACE) statement when an End-of-File or irrecoverable parity error occurs. A branch to the label preceding the colon takes place when an End-of-File condition occurs. A branch to the label following the colon takes place if an irrecoverable parity error occurs.

When a READ statement is executed where the file is assigned to the console typewriter, a message is typed on the SPO and the program is temporarily suspended.

The form of the message on the SPO is:

⟨job specifier⟩ ACCEPT

The operator responds to the above message by typing a message as follows:

⟨mix index⟩ AX ⟨input message⟩

The ⟨input message⟩ which follows AX is then read as specified by the READ statement and the program is re-initiated. The buffer will contain an end-of-message character following the last character of the ⟨input message⟩. This end-of-message character has the same code as the code for the character ←.

FREE-FIELD DATA.
SYNTAX. The syntax for ⟨free-field data⟩ is as follows:

⟨free-field data⟩ ::= ⟨field⟩ ⟨field delimiter⟩ | ⟨free-field data⟩ ⟨field⟩ ⟨field delimiter⟩

⟨field⟩ ::= ⟨number⟩ | ⟨string⟩ | % ⟨octal number⟩ | / | * | ⟨empty⟩

⟨field delimiter⟩ ::= , | ⟨letter⟩ {any proper string not containing a comma} , | {if the field is a slash (/), the end of the current record serves as a field delimiter}

Examples:

```
1,
2.5,
2.48 @ -20,
2 @ 34,
"THIS IS A STRING",
%12347,
1 DELIMITER,
2.5 ANY COMMENT OR NOTE NOT CONTAINING A COMMA,
2.48 @ -20 VALUE FOR Z* (-3),
2 @ 34 ET CETERA,
"THIS IS A STRING" THIS IS A COMMENT,
% 12347 AN OCTAL NUMBER,
* TERMINATES READ,
```

SEMANTICS. All Free-Field Input is in the form of free-field data. Each field, except the slash (/), is associated with the list element to which it corresponds according to position.

A free-field data sentence is in no way affected by the end of a record. That is, a field or field delimiter may be carried over from one record to another. Continuation from record to record is automatic until the LIST is exhausted or an asterisk (*) field is encountered. Unused characters (if any) on the last record read are lost.

All blanks in free-field data except those in strings are completely ignored.

Fields are handled as follows:

a. Numbers. A number which is represented as an INTEGER will be converted as an INTEGER unless it is larger than the largest allowable INTEGER, in which case it will be converted as REAL. Numbers which contain a decimal fraction will be converted as REAL.

b. Strings. Strings may be of any length. Each list element will receive six characters until either the list or the string is exhausted. If the number of characters in the string is not a multiple of six, then the last list element receives the remaining characters of the string. The string characters are stored right-justified in the list elements.

c. Octal Numbers. Octal numbers are placed right-justified in the list element, unchanged. The largest octal number allowed is 3777777777777777. A non-octal digit will terminate the number, treating the remainder of the field as comment.

d. Empty. An empty field will cause the corresponding list element to be ignored.

e. Slash (/). The slash (/) field will cause the remainder of the current record to be ignored. The record following the slash is considered the beginning of a new field; therefore, the slash field does not require (or recognize) any field delimiter other than the end of the record in which it occurs. A slash field has no effect on list elements. The slash is a field by itself and must not be placed within another field or between a field and its delimiter.

f. Asterisk (*). The asterisk (*) field terminates the read statement. The program continues with the next statement in sequence. The list element corresponding to the asterisk is left unchanged, as well as any subsequent elements in the list.

LOGICAL VALUES. For the purpose of Free-Field Input, an INTEGER 1 (one) must be used in lieu of the logical value TRUE, and an INTEGER 0 (zero) must be used in lieu of the logical value FALSE.

The example below demonstrates the Free-Field Input facility:

Example:

Consider each of the following lines as individual records:

```
 1
 2
 3
 @
29
 ,
+1   23   .   0   @   +
29
 ,
 +  . 1 2 3 @ 3 2
 , 0, X, A1, 4 A 5 B, / CARD 124
15 IGNORED, ZERO.
 % 177, %30, "THIS IS A STRING", """,
 "STRING", *, 2.7, 8.4,
```

If the above records (free-field data) were read with the statement

```
READ(FILEID,/,FOR I←0 STEP 1 UNTIL 18 DO A [I])
```

values would be assigned to A as follows:

```
A [0] = 123@29
A [1] = 123@29
A [2] = 123@32
A [3] = 0
A [4] = Unchanged
A [5] = Unchanged
A [6] = 4
A [7] = 15
A [8] = Unchanged
A [9] = 177 (octal)
```

```
A [10] = 30 (octal)
A [11] = 00THIS I
A [12] = 00S A ST
A [13] = 0000RING
A [14] = 0000000"
A [15] = 00STRING
A [16] = Unchanged
A [17] = Unchanged
A [18] = Unchanged
```

The occurrence of the asterisk (*) field on the last record terminates the read statement without assigning any values to A [16], A [17], or A [18]. The value of I (the controlled variable of the FOR clause) will remain at 16.

SPACE STATEMENTS.

SYNTAX.

The syntax for ⟨space statement⟩ is as follows:

⟨space statement⟩ ::= SPACE (⟨file part⟩, ⟨number of records⟩)
⟨action labels⟩

⟨number of records⟩ ::= ⟨arithmetic expression⟩

Examples:

```
SPACE (FILEID, 5) [LEONF:LPAR]
SPACE (FILEID, -3) [LEOF:LPAR]
SPACE (FILEID, A + B - C)
```

SEMANTICS.

The SPACE statement is used to bypass input logical records without reading them.

The value of the arithmetic expression determines the number of records to be spaced and the direction of the spacing. If the expression is positive, the records are spaced in a forward direction; if negative, in the reverse direction.

WRITE STATEMENTS.

SYNTAX.

The syntax for ⟨write statement⟩ is as follows:

⟨write statement⟩ ::= WRITE (⟨output parameters⟩)

⟨output parameters⟩ ::= ⟨file part⟩ ⟨carriage control⟩ |
⟨file part⟩ ⟨carriage control⟩ ⟨format and
list part⟩

⟨carriage control⟩ ::= [PAGE] | ⟨skip to channel⟩ | [DBL] |
[NO] ⟨empty⟩ | [STOP]

⟨skip to channel⟩ ::= [⟨arithmetic expression⟩]

Examples:

```
WRITE (FILEID, FMT, LISTID)
WRITE (FILEID [PAGE])
WRITE (FILEID, FMT)
WRITE (FILEID, *, LISTID)
WRITE (FILEID [DBL],FMT, A, B, C ARA[6])
WRITE (FILEID, X+Y+Z, ARA3[1,1*])
WRITE (FILEID)
WRITE (FLE[X + 2], FT, LST)
WRITE (SPO, 10, A[*])
WRITE (SPO, FRMT, LST)
```

SEMANTICS.

The WRITE statement causes output of information in the form of computational results and messages.

The file part specifies the file to be used.

The carriage control may be included to allow for paper control on the line printer. If the specified output unit is not a line printer, carriage control is irrelevant and is ignored.

[PAGE] causes the printer to skip to channel after each line of print.

Skip to channel causes the printer to skip to the channel indicated by the value of the arithmetic expression after each line of print.

[DBL] causes the printer to double space after each line of print.

[NO] causes the printer to suppress spacing after each line of print.

[STOP] causes the automatic carriage return and line feed to be suppressed at the end of a line written on a data communications unit.

The format and list part specifies the action to be taken on the output data.

A format identifier alone indicates that the referenced FORMAT declaration contains one or more strings which constitute the entire output.

A format identifier followed by a list or list identifier designates the variables in the list are to be placed in a format according to the specifications of the FORMAT declaration and written as output. The FORMAT declaration may contain strings as noted above.

The symbol * followed by a list or list identifier specifies that the variables in the list are to be processed as full words, and are to be written as output without being edited. The number of words written is determined by the number of variables in the list or the maximum record length, whichever is smaller. When unblocked records are used, the maximum record length is the buffer size.

An arithmetic expression used with a row designator specifies

that the elements of the designated array row are to be processed as full words and are to be written as output without being edited. The number of words written is determined by the number of elements in the array row, the maximum record length, or the absolute value of the arithmetic expression, whichever is smallest. When unblocked records are being used, the maximum record length is the buffer size.

WRITE statements which do not reference a FORMAT declaration provide a faster output operation than those which require data to be edited.

When a WRITE statement is executed where the file is assigned to the console typewriter, the output will be typed on the SPO. Writing is terminated when the end-of-message character (code for ←) is encountered in the message. This character is placed into the first character of the word immediately following the last output word. However, the program can place the character ← in the output string, if desired.

RESTRICTION.
The arithmetic expression in skip-to-channel requires an integer value from 1 through 11. If the arithmetic expression yields a value other than integer, it will be rounded to an integer in accordance with the rules applicable to the evaluation of subscripts (see page 3-3, evaluation of subscripts).

REWIND STATEMENTS.
SYNTAX.
The syntax for ⟨rewind statement⟩ is as follows:

⟨rewind statement⟩ ::= REWIND (⟨file part⟩)

Example:

```
REWIND (FILEID)
```

SEMANTICS.
The REWIND statement causes the referenced file to be closed and if tape, to be rewound. The I/O unit will remain under program control.

RESTRICTION.
On paper tape files, the REWIND statement may be used only on input.

LOCK STATEMENTS.
SYNTAX.
The syntax for ⟨lock statement⟩ is as follows:

⟨lock statement⟩ ::= LOCK (⟨file part⟩, RELEASE) | LOCK (⟨file part⟩, SAVE) | LOCK (⟨file part⟩)

Examples:

```
LOCK (FILEID, RELEASE)
LOCK (FILEID, SAVE)
```

SEMANTICS.
The LOCK statement causes the referenced file to be closed. If the file is tape, it is rewound and a system message is printed to notify the operator to remove the reel and save it.

If the file is not a disk file, the unit is made inaccessible to the system until the operator resets it again manually.

The three forms of the LOCK statement are equivalent.

CLOSE STATEMENTS.
SYNTAX.
The syntax for ⟨close statement⟩ is as follows:

⟨close statement⟩ ::= CLOSE (⟨file part⟩, RELEASE) | CLOSE (⟨file part⟩, SAVE) | CLOSE (⟨file part⟩) | CLOSE (⟨file part⟩, *) | CLOSE (⟨file part⟩, PURGE)

Examples:

```
CLOSE (FILEID, RELEASE)
CLOSE (FILEID, SAVE)
CLOSE (FILEID, *)
CLOSE (FILEID, PURGE)
```

SEMANTICS.
The CLOSE statement causes the referenced file to be closed. The following actions take place:

a. On a card output file, a card containing an ending label is punched.

b. On a line printer file, the printer is skipped to channel 1, an ending label is printed, and the printer is again skipped to channel 1.

c. On an unlabeled tape output file, a tape mark is written after the last block on tape.

d. On a labeled tape output file, a tape mark and ending label are written after the last block on tape.

If only the file part is used, or the SAVE or RELEASE is used, the I/O unit is released to the system. If the file is a tape file, the tape is rewound.

If the symbol * is used, the file must be a tape file. The I/O unit remains under program control and the tape is not rewound. This construct is used to create multi-file reels.

If PURGE is used, the file is closed, purged, and released to the system.

When the symbol * is used on a labeled multi-file input tape, the following action can take place:

a. If the last reference to a file was a READ or SPACE

FORWARD statement and a CLOSE (⟨file part⟩, *) is executed, the tape is positioned forward to a point just following the ending label of the file.

b. If the last reference to the file was a READ or SPACE REVERSE statement and a CLOSE (⟨file part⟩, *) is executed, the tape is positioned to a point just in front of the beginning label for the file.

c. If the CLOSE (⟨file part⟩, *) is executed after the End-of-File branch has been taken, no action is performed to position the file.

When the CLOSE (⟨file part⟩, *) is used on a single-file reel, the action taken is the same as for a multi-file reel.

FAULT STATEMENT.

SYNTAX.

The syntax for ⟨fault statement⟩ is as follows:

⟨fault statement⟩ ::= ⟨fault type⟩ ← 0 | ⟨fault type⟩ ← ⟨designational expression⟩

⟨fault type⟩ ::= EXPOVR | INTROVR | INDEX | FLAG | ZERO

Examples:

```
EXPOVR ← 0
INTOVR ← INTTOOBIG
INDEX ← SELECTPATH [I]
FLAG ← IF K = 1 THEN FINIS ELSE REDO
```

SEMANTICS.

The fault statement provides the means by which a programmer may specify programmatic action for any of the specific program errors. The program errors are associated with each fault type as shown in table 6-1. The fault statement requires a fault declaration (described on page 7-44).

Table 6-1

Program Errors for Fault Types

⟨fault type⟩	Meaning
EXPOVR	Exponent overflow
INTOVR	Integer overflow
INDEX	Invalid index
FLAG	Flag bit
ZERO	Divide-by-zero

If one of the program errors occurs and there is an associated ⟨fault type⟩ ← ⟨designational expression⟩ statement, transfer of control to the evaluated designational expression will take place provided:

a. The error occurred during the execution of a statement within the scope of the label.

b. The error occurred in a procedure that was called by a procedure call statement that is within the scope of the label.

Transfer of control will not take place if it will result in the entering of a block other than through the block head.

The designational expression is evaluated when the fault statement is executed and not at the time that the error occurs. If multiple fault declarations are made (i.e., in nested blocks) when an error occurs, only the most local declaration for that type will be examined.

The ⟨fault type⟩ ← 0 statement is the means of turning off the transfer control fault statement. After this form of fault statement has been executed, the program will be terminated if the specific error occurs.

ZIP STATEMENT.

SYNTAX.

The syntax for ⟨zip statement⟩ is as follows:

```
⟨zip statement⟩ ::= ZIP WITH ⟨array row⟩ |
                    ZIP WITH ⟨file part⟩
```

Examples:

```
ZIP WITH CONTROLCARD[I,*]
ZIP WITH FILEID
```

SEMANTICS.

The ZIP WITH ⟨array row⟩ statement causes information in the designated array row to be recognized as control and/or program parameter card information. The information in the array row must be in the BCL (6-bit) format as it would appear on the control program parameter cards. The letters CC may be used in lieu of a question mark (?), but only one may appear in the array row. The information in the array row appears as a single punched card, but is not limited to 72 characters. The information that would be contained on more than one control card may be put into the array row, but a semicolon must be used to delimit the end of a card.

The control information to be utilized by the ZIP WITH ⟨array row⟩ statement should pertain to only one Compiler or object program. The last card in the array row must contain the following:

```
END.
```

After the ZIP WITH ⟨array row⟩ statement has been executed, the object program that executed the statement continues processing, while the MCP examines the control information in the array row. If the MCP finds an error in this control information, an appropriate error message is typed on the supervisory printer to notify the operator.

The ZIP WITH ⟨file part⟩ statement causes information in the designated disk file identified by ⟨file part⟩ to be considered as a control deck. Each logical record must be one card, i.e., 10 words. Logical record zero (0) must be a control card and must contain in its tenth word the logical record number (a binary integer) of the next control card in the control deck including LABEL cards. Each successive control card, likewise, points to the next control card. There must be an END control card in the control deck as the last card which points to itself. The proper format of a control deck on disk is illustrated in figure 6-1.

When the ZIP WITH ⟨file part⟩ statement is executed, the object program which executed that statement continues processing, while the file ⟨file part⟩ is passed to the MCP. If a file other than a disk file is referenced, the ZIP statement is ignored. If the referenced disk file is not on disk, the ZIP statement is ignored. The MCP does not check to ensure that the control deck is properly arranged; this is a responsibility of the programmer.

After execution of the ZIP WITH ⟨file part⟩ statement is completed, the control deck referenced by the designated file is purged from the disk directory.

LABEL EQUATION STATEMENT.

SYNTAX.

The syntax for ⟨label equation statement⟩ is as follows:

⟨label equation statement⟩ ::= FILL ⟨file part⟩ WITH
⟨label equation information⟩

⟨label equation information⟩ ::= ⟨multi-file identification⟩ |
⟨multi-file identification⟩, ⟨file identification⟩ |
⟨multi-file identification⟩, ⟨file identification⟩,
⟨reel number⟩ | ⟨multi-file identification⟩, ⟨file
identification⟩, ⟨reel number⟩, ⟨date⟩ | ⟨multi-file
identification⟩, ⟨file identification⟩, ⟨reel number⟩,
⟨date⟩, ⟨cycle number⟩ | ⟨multi-file identification⟩,
⟨file identification⟩, ⟨reel number⟩, ⟨date⟩, ⟨cycle
number⟩, ⟨output media part⟩

ZIP WITH ⟨file id⟩ CONSTRUCT

Logical Record	WD 1	WD 2	WD 3	WD 4	WD 5	WD 6	WD 7	WD 8	WD 9	WD 10
0	? EXECUTE ANY/JOB									1 (in binary)
1	? LABEL INPUT									9 (in binary)
2				(DATA CARDS)						
3				"	"					
4				"	"					
5				"	"					
6				"	"					
7				"	"					
8				"	"					
9	? COMPILE A/B WITH ALGOL									10 (in binary)
10	? DATA CARD									17 (in binary)
11				(SOURCE LANGUAGE CARDS)						
12				"	"	"				
13				"	"	"				
14				"	"	"				
15				"	"	"				
16				"	"	"				
17	? DATA DATA									21 (in binary)
18				(DATA CARDS)						
19				"	"					
20				"	"					
21	? END.									21 (in binary)

Figure 6-1. Format for Control Deck On Disk

⟨multi-file identification⟩ ::= ⟨arithmetic expression⟩ | *

⟨file identification⟩ ::= ⟨arithmetic expression⟩ | *

⟨reel number⟩ ::= ⟨arithmetic expression⟩ | *

⟨date⟩ ::= ⟨arithmetic expression⟩ | *

⟨cycle number⟩ ::= ⟨arithmetic expression⟩ | *

⟨output media part⟩ ::= ⟨arithmetic expression⟩ | * | REMOTE

Examples:

```
FILL FID WITH "MULTI", "FILEID"
FILL FI WITH *, "FILEID", *, 66123
FILL SFI[I]WITH X, Y, R, D, C, 2
```

SEMANTICS.

The label equation statement provides the means to programmatically specify the file LABEL information associated with a file ⟨file part⟩. This statement is a programmatic program parameter card. To have effect, a label equation statement must be executed while the designated file is not open; otherwise, the statement is ignored.

When a label equation statement is executed, the label equation information is assigned to the file ⟨file part⟩ and is used in association with the input/output statements using the specified file. If any part of the label equation information contains an asterisk, that part of the information will remain as it was before the statement was executed.

All label equation information, except the output media part, must be in the format required in a standard label. The values which the output media digit may have and their meanings are listed in table 6-2. The values of the multi-file and file identification parts are interpreted as ALPHA and can contain up to seven characters in the variable or string.

An output media part of REMOTE indicates a data communications file.

Table 6-2

Values for Output Media Digit

⟨output media part⟩	Meaning
0	Card punch
1	Line printer
2	Labeled magnetic tape
4	Line printer or printer backup tape
5	Labeled designated output file
6	Printer backup tape
7	Unlabeled designated output file
8	Unlabeled paper tape
9	Unlabeled magnetic tape
10	Random disk file
11	Supervisory printer
12	Serial disk file
13	Update disk file
REMOTE	Data communications file
15	Printer backup disk
16	Printer backup tape or disk
17	Line printer or printer backup disk
18	Line printer or printer backup tape or disk
32	Special forms message required

EDIT AND MOVE STATEMENT.

SYNTAX.

The syntax for ⟨edit and move statement⟩ is as follows:

⟨edit and move statement⟩ ::= ⟨edit and move read⟩ |
⟨edit and move write⟩

⟨edit and move read⟩ ::= READ (⟨array row⟩, ⟨format

and list part⟩) | READ (⟨array row⟩,
⟨free field part⟩)

⟨edit and move write⟩ ::= WRITE (⟨array row⟩,
⟨format and list part⟩)

Examples:

```
READ (A[*], FMT, LST);
WRITE (XA[I,*], 25, B[*]);
READ (DD[*], /, R, A);
```

SEMANTICS.
The edit and move statement provides the means of utilizing the editing features of READ and WRITE statements without using I/O files and buffer areas. In effect, the ⟨array row⟩ designated in the edit and move statement is analogous to a buffer area.

When an ⟨edit and move read⟩ statement is executed, data in the designated array row is edited and placed in the list. The format part determines what editing is to take place as the data is moved from the array row to the list.

When an ⟨edit and move write⟩ statement is executed, data from the list is edited and placed into the designated array row. The data is edited as specified by the format part as it is moved from the list to the array row.

If the edit and move statement calls for more than one physical record, the array row will be reused when the new record is required.

DISK I/O STATEMENT.
SYNTAX.
The syntax for ⟨disk I/O statement⟩ is as follows:

⟨disk I/O statement⟩ ::= ⟨disk read statement⟩ |
⟨disk write statement⟩ | ⟨disk read seek statement⟩ |

⟨disk space statement⟩ | ⟨disk rewind statement⟩ |
⟨disk close statement⟩ | ⟨disk lock statement⟩

SEMANTICS.
The disk I/O statements allow the programmer to utilize the disk for creating files and using created files. A record pointer is associated with the I/O statements. This record pointer is always set to the address of the logical record that is accessed by a READ or WRITE statement.

DISK READ STATEMENT.
SYNTAX.
The syntax for ⟨disk read statement⟩ is as follows:

⟨disk read statement⟩ ::= READ ⟨direction⟩ ((⟨disk input parameters⟩) ⟨action labels⟩

⟨disk input parameters⟩ ::= ⟨file part⟩ ⟨record address and release part⟩, ⟨format and list part⟩ | ⟨file part⟩ ⟨record address and release part⟩ | ⟨file part⟩ ⟨record address and release part⟩, ⟨free field part⟩

⟨record address and release part⟩ ::= [⟨address⟩] | [NO] | ⟨empty⟩

⟨address⟩ ::= ⟨arithmetic expression⟩

Examples:

```
READ REVERSE (OLDFILE, FRMAT, LST)
READ (FREEFILE, /, FREELIST) [:PAR]
READ (NEWFILE[NO], *, BILST) [EOF:PAR]
READ (DATA[NEXT], NOREC, ARA[I,*])
```

SEMANTICS.
A disk READ statement causes data to be read from a disk record and placed into the list variables as specified by the format. The record pointer may be adjusted by the READ statement.

If a REVERSE direction is used in the READ statement, the value of the record is decreased by one prior to performing the read. If the value of the record pointer is N when a read reverse is executed, the record pointer is set to N-1 before the read is performed. At the completion of the read reverse, the record pointer remains at N-1.

If an ⟨address⟩ is used in the record address and release part, the ⟨address⟩ specifies the relative address in the file of the record to be read and edited as specified in the READ statement. The record pointer is set to ⟨address⟩ before the read is performed. The record pointer is not adjusted after the read is executed. An ⟨address⟩ must be used when a file is declared RANDOM.

If an ⟨address⟩ is not specified and NO is not used, the record read will be the one pointed to by the record pointer. After the read has been executed, the record pointer is adjusted to point to the next record in the file.

If NO is used, the record read will be the one to which the record pointer is set. After the read has been executed, the record pointer will not be adjusted.

The format and list part have the same meaning for disk I/O that they have for all other I/O's.

The action labels provide means of transferring control from a READ statement when an End-of-File or Parity condition occurs. The label preceding the colon is branched to on an End-of-File condition. The label following the colon is branched to on a Parity Error condition.

An End-of-File condition occurs whenever an attempt is made to read a record of which the address is greater than the EOF indicator, or less than zero. The EOF indicator is the address of the highest record address written when the file was created. This indicator is updated whenever additional records are written onto the file.

DISK WRITE STATEMENT.

SYNTAX.

The syntax for ⟨disk write statement⟩ is as follows:

⟨disk write statement⟩ ::= WRITE (⟨disk output parameters⟩)
[⟨action labels⟩]

⟨disk output parameters⟩ ::= ⟨file part⟩ ⟨record address part⟩
| ⟨ file part⟩ ⟨record address part⟩, ⟨format and
list part⟩

⟨record address part⟩ ::= [⟨address⟩] | ⟨empty⟩

Examples:

```
WRITE (FILEX[NEXT], *, LIT)
WRITE (INVNTRY[PARTNO], 60, ARA[*])
WRITE (NEWFILE, FRMT, LST)
```

SEMANTICS.

Disk WRITE statements cause information to occur as output according to the format from the list specified. Whenever the WRITE statement is executed, the record pointer will be adjusted.

The disk file on which the output is to be written is specified by the file part.

If an ⟨address⟩ is specified, the record pointer is set to this relative address prior to executing the WRITE statement. The ⟨address⟩ must be provided if the specified file is declared RANDOM.

If the record address part is empty, the WRITE statement will cause the output to be written onto the file at the present record pointer location.

The record pointer is always adjusted to the next record location following the execution of the WRITE statement.

The format and list part have the same meaning for disk I/O as

it has for other I/O's. However, if it is empty, the contents of the current buffer are written onto the disk. An empty format and list part should only be used with unblocked files.

An End-of-File condition occurs if an attempt is made to write a record which has an address outside of the file, as declared. The End-of-File action label provides the programmer with the means of branching to a label if this condition occurs.

DISK READ SEEK STATEMENT.

SYNTAX.

The syntax for ⟨disk read seek statement⟩ is as follows:

⟨disk read seek statement⟩ ::= READ SEEK (⟨file part⟩ [⟨address⟩])

Example:

READ SEEK (PARTFILE[NEXT])

SEMANTICS.

The principle use of the READ SEEK statement is with files declared RANDOM. It provides the means of filling a buffer in anticipation of a READ or WRITE action on the record as specified by the ⟨address⟩.

When each READ SEEK statement is executed, records are subsequently read into buffer areas. The records are queued according to the order in which they were requested. If more READ SEEK statements are executed than there are buffers, records are lost, starting at the head of the queue.

When a READ is executed, the record addressed is searched for, starting at the head of the queue. If the first record in the queue is not the desired record, that record is released or lost

and the next record becomes the head of the queue. This sequence continues until the record is found or the queue is empty. If the record is not in the queue, the addressed record is then read from the disk file.

When a WRITE statement is executed, a copy of the record may be required in core before the WRITE is performed (explained under file declarations for disk files in section 7). If a WRITE is performed, one of the following may occur:

a. If a copy of the record is not required, the record at the head of the queue would be lost.

b. If a copy of the record is in a buffer area, that buffer will be used as an output buffer and all records in the queue preceding the record written are lost.

c. If a copy of the record is required, an implicit READ takes place and all records in the queue are lost.

An example of the misuse of a READ SEEK statement follows:

```
READ (PARTFILE[3], . . .);
  .
  .
  .
READ SEEK (PARTFILE[18]);
  .
  .
  .
WRITE (PARTFILE[3], . . .);
```

Consider the file to be declared RANDOM, blocked, and with one buffer area. The actions that would take place as the statements shown are executed would be:

a. The physcial record containing record [3] would be read.

b. The READ SEEK on record [18] would cause record [3] to be lost.

c. The WRITE of record [3] would require the physical record containing record [3] to be reread into the buffer, destroying record [18].

Programs containing such statements are not desirable and should be avoided. Used properly, the READ SEEK statement can be of great value.

If a READ SEEK statement is performed on a SERIAL or UPDATE file, the READ SEEK statement specifies that the next record to be processed is given by the ⟨address⟩.

DISK SPACE STATEMENT.

SYNTAX.

The syntax for ⟨disk space statement⟩ is as follows:

⟨disk space statement⟩ ::= SPACE (⟨file part⟩, ⟨number of records⟩)

⟨number of records⟩ ::= ⟨arithmetic expression⟩

Examples:

```
SPACE (FILEID, 5)
SPACE (FILEID, -5)
SPACE (FILEID, CNTR)
```

SEMANTICS.

The SPACE statement provides the means of adjusting the value of the record pointer. When the SPACE statement is executed, the record pointer is adjusted by the value of the arithmetic expression.

DISK REWIND STATEMENT.

SYNTAX.

The syntax for ⟨disk rewind statement⟩ is as follows:

⟨disk rewind statement⟩ :: REWIND (⟨file part⟩)

Example:

```
REWIND (FILEID)
```

SEMANTICS.
The REWIND statement causes the record pointer to be set to the address of the first record in the file.

DISK CLOSE STATEMENT.
SYNTAX.
The syntax for ⟨disk close statement⟩ is as follows:

⟨disk close statement⟩ ::= CLOSE (⟨file part⟩) | CLOSE (⟨file part⟩, RELEASE) | CLOSE (⟨file part⟩, SAVE) | CLOSE (⟨file part⟩, *) | CLOSE (⟨file part⟩, PURGE)

Examples:

```
CLOSE (FILEID)
CLOSE (FILEID, RELEASE)
CLOSE (FILEID, SAVE)
CLOSE (FILEID, *)
CLOSE (FILEID, PURGE)
```

SEMANTICS.
A CLOSE statement causes the buffer areas reserved for the file to be returned. Also, if the file is a temporary file, the disk space for the file is returned.

If a CLOSE with PURGE statement is executed on a permanent file, that file is removed from the disk directory and the disk space is returned.

DISK LOCK STATEMENT.
SYNTAX.
The syntax for ⟨disk lock statement⟩ is as follows:

⟨disk lock statement⟩ ::= LOCK (⟨file part⟩ |
LOCK (⟨file part⟩, RELEASE) |
LOCK (⟨file part⟩, SAVE)

Examples:

```
LOCK (FILEID)
LOCK (FILEID, RELEASE)
LOCK (FILEID, SAVE)
```

SEMANTICS.
A LOCK statement causes a temporary file to be made permanent. All of the LOCK statements cause the same action on the file. When it is executed, an entry is made into the disk directory for the file; and the buffer areas reserved for the file are returned.

SEARCH STATEMENT.
SYNTAX.
The syntax for the ⟨search statement⟩ is as follows:

⟨search statement⟩ ::= SEARCH (⟨file part⟩, ⟨array row⟩)

⟨file part⟩ ::= ⟨file identifier⟩ | ⟨switch file designator⟩

Examples:

```
SEARCH (DISKFILE, A[*])
SEARCH (DISKFILESWITCH [I], A[*])
SEARCH (DISKFILE, B[J, *])
```

SEMANTICS.
The SEARCH statement provides a programmer with the means to determine the existence of a disk file which is accessible under the File Security System. The SEARCH statement causes the MCP to perform a disk directory search for the specified file. Values are assigned to the elements of the designated array row depending on the results of the directory search.

If the specified file is present and the requester is a legitimate

user of the file, the MCP will set the designated array row as follows:

WORD	CONTENTS
0	7 if primary user 3 if secondary user 2 is tertiary user
1	Multi-file identification
2	File identification
3	Record length
4	Block length
5	End-of-file pointer
6	Open counter

If the specified file is not present in the disk directory, the MCP will set words 0, 3, 4, 5, and 6 of the designated array row all to negative one (-1).

If the specified file is present but the requester is not a legitimate user of the file, the MCP will set words 0, 3, 4, 5, and 6 of the designated array row to zero (0).

The designated array row must be at least seven (7) words in length. If the array row is less than seven words, the object program will be terminated with an invalid index.

FILL STATEMENT.

SYNTAX.

The syntax for ⟨fill statement⟩ is as follows:

⟨fill statement⟩ ::= FILL ⟨array identifier⟩ [⟨row designator⟩] WITH ⟨value list⟩

⟨row designator⟩ ::= * | ⟨row⟩ , *

⟨row⟩ ::= ⟨arithmetic expression⟩ | ⟨row⟩, ⟨arithmetic expression⟩

⟨value list⟩ ::= ⟨initial value⟩ | ⟨value list⟩ , ⟨initial value⟩

⟨initial value⟩ ::= ⟨number⟩ | ⟨string⟩ | ⟨unsigned integer⟩ (⟨value list⟩)

Example:

```
FILL MATRIX[*] WITH 458.54, +546, -1354.54@6, 16@-12
```

SEMANTICS.
The FILL statement fills an array row with specified values.

ROW DESIGNATOR. The row designator indicates which row is to be filled by designating a specific value for each subscript position of the array row except the right-most position. The symbol * must appear in the right-most subscript position of the row designator.

If the values of a row are other than integer, it is rounded to an integer in accordance with the rules applicable to assignment statements (see page 6-6).

VALUE LIST. Each initial value may have one of three forms (number, string, or octal number), and a value list may contain any mixture of these forms. The concept of type does not apply to initial values, and transfer functions are not invoked because the array is filled as indicated.

A number is converted to its octal equivalent, then stored.

A string causes the six-bit code for each character in the string, other than the two string bracket characters at the ends, to be stored. The string may contain as many as eight characters. If fewer than eight characters are in the string, leading zeros are supplied.

An octal number will be stored as such, and must not exceed 16 digits.

The number of initial values in the value list may differ from the number of elements in the row being filled. If the number of values is less than the number of elements, the elements with the largest subscript values retain their former values. If the number is greater than the number of elements, the right-most values in the value list are not used.

RESTRICTIONS.
The maximum number of words allowed in a single FILL statement is 1023. A defined identifier (see page 7-9) must not be used in a FILL statement. There must be no space between OCT and the octal number which follows.

SORT STATEMENT.
SYNTAX.
The syntax for ⟨sort statement⟩ is as follows:

⟨sort statement⟩ ::= SORT (⟨output option⟩, ⟨input option⟩, ⟨number of tapes⟩, ⟨hivalue procedure⟩, ⟨compare procedure⟩, ⟨record length⟩ ⟨size specifications⟩)

⟨size specifications⟩ ::= ⟨empty⟩ | ⟨core size⟩ | ⟨core size⟩ ⟨disk size⟩

⟨core size⟩ ::= , ⟨arithmetic expression⟩

⟨disk size⟩ ::= , ⟨arithmetic expression⟩

⟨record length⟩ ::= ⟨arithmetic expression⟩

⟨compare procedure⟩ ::= ⟨identifier⟩

⟨hivalue procedure⟩ ::= ⟨identifier⟩

⟨number of tapes⟩ ::= ⟨arithmetic expression⟩

⟨input option⟩ ::= ⟨file part⟩ | ⟨input procedure⟩

⟨input procedure⟩ ::= ⟨identifier⟩

⟨output option⟩ ::= ⟨file part⟩ | ⟨output procedure⟩

⟨output procedure⟩ ::= ⟨identifier⟩

Examples:

```
SORT (OUTPRCD, INPRCD, 3, HIVAL, COMP, 3)
SORT (OUTFID, INFID, 0, HI, CMP, 2)
```

SEMANTICS.
The SORT statement provides the means whereby data, as specified by the ⟨input option⟩, is reordered and returned to the program, as specified by the ⟨output option⟩. The sequence of reordering the data is determined by the ⟨compare procedure⟩.

The size specifications allow the programmer to specify the amount of main memory and the amount of disk storage that may be used.

The core size, if present, specifies the number of words of main memory that may be used. If unspecified, a value of 1200 is assumed.

The disk size, if present, specifies the amount of disk storage in words that may be used. If unspecified, a value of 600,000 words of disk storage is assumed (this is equivalent to 0.5 disk file module).

The record length represents the length in words of the largest item that will be presented to the SORT statement. If the value of the arithmetic expression is not a positive integer, the largest integer which is less than the absolute value of the expression will be used (i.e., a record length of 12 would be used if an expression had a value of -12.995). If the value of the arithmetic expression is zero (0), the program will loop indefinitely.

The compare procedure is called by the SORT to determine which of two records should be used next in the sorting process. It must be a BOOLEAN procedure with exactly two (2) parameters. Both of

the parameters must be arrays. The Boolean value which is returned via the procedure identifier should be TRUE if the array given as the first parameter is to appear in the output before the array given as the second parameter. As an example, the following procedure could be used for sorting in ascending sequence:

```
BOOLEAN PROCEDURE CMP (A, B);
ARRAY A, B [0];
CMP ← A[0] ≤ B[0];
```

In the example, CMP would be TRUE if array A is equal to or less than array B, and CMP would be FALSE if array A is greater than array B. This would result in the lower valued array being passed to the output first.

The hivalue procedure is called by the SORT to create a unique record for its own internal use. The record created is not returned as sorted output. This created record must be such that it will cause the compare procedure to determine that it should appear after all valid input items being sorted. This procedure must be untyped and must have an array as its only parameter. This procedure is a hivalue procedure if sorting in ascending sequence, and essentially a low-value procedure if sorting in a descending sequence. The following is an example of a hivalue procedure that could be used by the compare procedure above.

```
PROCEDURE HV (A);
ARRAY A[0];
FILL A[*] WITH OCT777777777777777;
```

The number of tapes specifies the number of tape files that may be used, if necessary, in the sorting process. If the value of the arithmetic expression is less than three (3), no tapes will be used. If five (5) or more tapes are specified, five tapes may be used if it is necessary; otherwise, the specified number of tapes will be used, if necessary.

If an input file is used as the input option, the records in that

file will be used as input to the SORT. This file will be LOCKed after all of the records on the file have been read by the SORT.

If an input procedure is used as the input option, the procedure is called on to furnish input records to the SORT. This input procedure must be a BOOLEAN PROCEDURE, with an array as its only parameter. This procedure, on each call, will:

a. Either insert the next record to be sorted into its array parameter.

b. Or assign a TRUE value to the procedure identifier.

When a TRUE is returned by the input procedure, the SORT will not use the contents of the array parameter and will not call on the input procedure again during the SORT. An example of an input procedure that will sort N elements of the array Q follows:

```
BOOLEAN PROCEDURE INPROC (A);
ARRAY A[0];
IF NOT (INPROC← (N←N-1) < 0) THEN A[0] ← Q[N];
```

If an output file is specified as the output option, the SORT will write the sorted output on this file. Upon completion of the SORT, the file will be LOCKed.

If an output procedure is specified as the output option, the SORT will call on this procedure once for each sorted record and once to allow end-of-output action. This procedure must be untyped and must use two parameters. The first parameter must be Boolean and the second parameter must be an array. The Boolean parameter will be FALSE until the last record has been returned from the SORT. When the first parameter is FALSE, the second parameter will contain a sorted record. When all records have been returned, the first parameter will be TRUE and the second parameter must not be accessed. An example of an output procedure follows:

```
PROCEDURE OUTPROC (B, A);
```

```
VALUE B;
BOOLEAN B;
ARRAY A[0];
IF B THEN CLOSE (FILEID, RELEASE) ELSE WRITE (FILEID,
 RECSIZE, A[*]);
```

PROGRAM EXAMPLE.

The following is an example of a program to perform a tag sort of a disk file, with printed output.

```
SAMPLE TAG SORT PROGRAM BEGIN
FILE IN DISK DISK RANDOM "INPUT" "TOSORT"(2,15,30);
FILE OUT P 6(2,15);
BOOLEAN BOO;
ARRAY Q[0:14];
INTEGER N;
BOOLEAN PROCEDURE IP(A); ARRAY A[0];
   BEGIN LABEL EOF,XIT;
      READ(DISK[N],15,Q[*])[EOF];
      A[0]←Q[0]; A[1]←N; N←N+1;
      GO TO XIT;
EOF:  BOO←TRUE;
XIT:  IP←BOO;
   END IP;
BOOLEAN PROCEDURE CMP(A,B); ARRAY A,B[0]; CMP←A[0]≤B[0];
PROCEDURE HV(A); ARRAY A[0]; A[0]←549755813887;
PROCEDURE OP(B,A); VALUE B; BOOLEAN B; ARRAY A[0];
   IF B THEN CLOSE(P) ELSE
   BEGIN FORMAT F(I8,"                                                ",
                  "                                                ",
                  "                                    ");
      READ(DISK[A[1]],F,N);
      WRITE(P,F,A[0]);
   END OP;
COMMENT START OF PROGRAM;
BOO←FALSE;
```

```
N←0;
SORT(OP,IP,0,HV,CMP,2);
END OF PROGRAM.
```

MERGE STATEMENT.

SYNTAX.

The syntax for ⟨merge statement⟩ is as follows:

⟨merge statement⟩ ::= MERGE (⟨output option⟩, ⟨hivalue procedure⟩, ⟨compare procedure⟩, ⟨record length⟩, ⟨merge file list⟩)

⟨merge file list⟩ ::= ⟨merge file⟩, ⟨merge file⟩ | ⟨merge file⟩, ⟨merge file list⟩

⟨merge file⟩ ::= ⟨file identifier⟩ | ⟨switch file designator⟩

Examples:

```
MERGE (FA, HV, CMP, 10, SWF[I], FC, FILESW[I]);
```

SEMANTICS.

The MERGE statement causes data in all of the files specified by the merge file list to be combined and returned. The compare procedure determines the manner in which the data is combined. The output option specifies the way in which the data is returned from the merge.

The merge file list must contain two files but may contain as many as seven merge files as input to the merge.

SECTION 7
DECLARATIONS

GENERAL.

SYNTAX.

The syntax for ⟨declaration⟩ is as follows:

⟨declaration⟩ ::= ⟨type declaration⟩ | ⟨array declaration⟩ |
⟨pointer declaration⟩ | ⟨switch declaration⟩ |
⟨define declaration⟩ | ⟨label declaration⟩ |
⟨procedure declaration⟩ | ⟨I/O declaration⟩ |
⟨forward reference declaration⟩ | ⟨dump declaration⟩ |
⟨monitor declaration⟩ | ⟨fault declaration⟩ |
⟨file declaration⟩ | ⟨switch file declaration⟩ |
⟨format declaration⟩ | ⟨switch format declaration⟩ |
⟨list declaration⟩ | ⟨switch list declaration⟩

SEMANTICS.

Declarations define certain properties of entities and relate these entities with identifiers.

The entities dealt with in Compatible ALGOL are:

a. Variables.
b. Labels.
c. Procedures.
d. Strings.

Every identifier has a scope. The scope of the identifier is usually the block in which it is declared. The exceptions are:

a. Formal symbols in a define declaration.
b. Formal parameters in a procedure declaration.

An identifier is said to be local to the block in which it is declared. That is, the entity represented by the identifier inside the block is not recognized by that identifier outside the block. Conversely, any entity represented by the identifier outside the block is not recognized by that identifier inside the block.

An identifier is said to be global to a block if:

a. It is not declared in the block, and
b. It is declared in an exterior block.

When the block is entered, all identifiers declared for the block assume the significance implied by the nature of the declarations.

At the time of exit from the block, all identifiers which are declared for the block lose their significance and reassume any previous significance which they may have had.

If the quantity represented by an identifier was declared OWN, the value(s) associated with the quantity are not lost upon block exit and are available upon reentry into the block.

An identifier may not be declared to represent more than one entity within a given block.

TYPE DECLARATIONS.

SYNTAX.

The syntax for ⟨type declaration⟩ is as follows:

⟨type declaration⟩ ::= ⟨local or own type⟩ ⟨type list⟩

⟨local or own type⟩ ::= ⟨type⟩ | OWN ⟨type⟩

⟨type⟩ ::= REAL | INTEGER | BOOLEAN | ALPHA

⟨type list⟩ ::= ⟨identifier⟩ | ⟨type list⟩ , ⟨identifier⟩

Examples:

```
INTEGER A, B, C
ALPHA NAME, CODE, AREA
OWN REAL, Q,R,T
```

SEMANTICS.

A type declaration defines the type of value of each identifier in the type list.

LOCAL OR OWN. The local or OWN portion of the type declaration indicates whether the value associated with a simple variable is to be retained upon exit from the block in which it is declared. A variable which has been declared as OWN retains its value upon exit from the block, and, at the time of reentry into that block, is defined as to its value. The values of variables not declared OWN are undefined upon reentry into the block, and these variables must be initialized again.

TYPE. Four declarators are defined for type declarations; their meanings are shown below.

a. REAL (single precision positive and negative values, including zero).

b. INTEGER (positive and negative integral values, including zero).

c. BOOLEAN (logical value).

d. ALPHA (character values, treated as REAL except for monitoring purposes).

LABEL DECLARATIONS.

SYNTAX.

The syntax for ⟨label declaration⟩ is as follows:

⟨label declaration⟩ ::= LABEL ⟨label list⟩

⟨label list⟩ ::= ⟨identifier⟩ | ⟨label list⟩ , ⟨identifier⟩

Examples:

```
LABEL START
LABEL ENTER, EXIT, START, STOP
```

SEMANTICS.

As in the case of all identifiers, a label must be declared before

it is used. A label identifier must appear in a label declaration in the head of the block in which it is used to label a statement.

A label declaration defines each identifier in its label list as a label identifier.

If any statement in a procedure body is labeled, the declaration of this label must appear within the procedure body.

ARRAY DECLARATION.

SYNTAX.

The syntax for ⟨array declaration⟩ is as follows:

⟨array declaration⟩ ::= ⟨array kind⟩ ARRAY ⟨array list⟩

⟨array kind⟩ ::= ⟨empty⟩ | ⟨local or own type⟩ | SAVE ⟨local or own type⟩

⟨array list⟩ ::= ⟨array segment⟩ | ⟨array list⟩ , ⟨array segment⟩

⟨array segment⟩ ::= ⟨identifier⟩ [⟨bound pair list⟩] | ⟨identifier⟩, ⟨array segment⟩

⟨bound pair list⟩ ::= ⟨bound pair⟩ | ⟨bound pair list⟩ , ⟨bound pair⟩

⟨bound pair⟩ ::= ⟨lower bound⟩ : ⟨upper bound⟩

⟨lower bound⟩ ::= ⟨arithmetic expression⟩

⟨upper bound⟩ ::= ⟨arithmetic expression⟩

Examples:

ARRAY Declarations:

```
INTEGER ARRAY MATRIX [1:IF B2 THEN B + K ELSE B + I]
OWN REAL ARRAY GROUP [0:9]
SAVE OWN BOOLEAN ARRAY GATE [1:10, 3:9]
```

ARRAY Lists:

```
MATRIX [0:9]
MATRIX, GROUP [0:9, 3:9]
```

ARRAY Segments:

```
MATRIX [0:9]
MATRIX, GROUP [0:9]
```

Bound Pair Lists:

```
9:9
0:9, 3:9
A + 2:B + 4
IF B1 THEN A + K ELSE A + I:IF B2 THEN B + K ELSE B + I
```

SEMANTICS.
An ARRAY declaration declares one or more identifiers to represent arrays of subscripted variables, and gives the number of dimensions of the arrays, the bounds of the subscripts, and the types of the variables.

SAVE ARRAYS. The declarator SAVE causes absolute storage allocation for an array to remain fixed. SAVE prevents overlaying of the storage areas allocated for an array.

LOCAL OR OWN. An array may be declared as OWN with the same effect as that given for simple variables (see page 7-2, type declarations).

In the case of dynamic OWN arrays, i.e., those arrays whose elements behave as OWN declared variables and whose subscript bounds may change with each entrance to the block in which the array is declared, the array is remapped in memory automatically.

However, this remapping may cause the loss of some elements of the original array. Only those elements whose subscripts are the same as the subscripts of the new array are copied over to this new array. The rest of the elements of the old array are lost.

TYPE. Arrays which are declared in the same array declaration must be the same type. If an array is declared OWN, the type must be declared. If the array is not OWN, type may be omitted, in which case the type will become REAL by default. If an array with variable bounds is declared OWN, the values of the corresponding subscripted variables are defined for only those variables which have subscripts within the most recently calculated bounds.

BOUND PAIR LIST. The bound pair list defines the dimensions of the array and the number of elements in each dimension; the subscript bounds for an array are given in the first bound pair list following the array identifier (see page 3-3, evaluation of subscripts).

The bound pair list gives the lower and upper bounds of all subscripts taken in order from left to right. The expressions are evaluated once, from left to right, upon entrance into the block.

Expressions used in forming bound pairs can depend only on variables and procedures which are nonlocal to the block for which the ARRAY declaration is valid. Arrays declared in the outermost block must therefore use constant bounds.

The number of dimensions in the array equals the number of bound pairs in a bound pair list.

Upper bounds must not be smaller than the corresponding lower bounds. No dimension may contain more than 1023 elements.

POINTER DECLARATIONS.

SYNTAX.

The syntax for ⟨pointer declaration⟩ is as follows:

⟨pointer declaration⟩ ::= POINTER ⟨pointer list⟩

⟨pointer list⟩ ::= ⟨identifier⟩ | ⟨pointer list⟩ ,
⟨identifier⟩

SEMANTICS.

A pointer represents the address of a character position in a one-dimensional array or array row. That is to say it points to a character position.

The pointer declaration establishes each identifier in the pointer list as a pointer identifier.

Associated with each pointer identifier is a pointer level. This level is defined to be the lexicographic (addressing) level at which the pointer identifier is declared. This represents the innermost (or greatest) addressing level to which the pointer identifier may point. No pointer assignment or pointer update operation may cause a pointer identifier to point into an array which is declared at a greater lexicographic level.

The rules for obtaining the lexicographic level of a declaration are:

a. Assign a lexicographic level of 2 for the outermost block of the program. Repeat the following steps until the subject declaration level is reached.

b. Add 1 for each block head BEGIN.

c. Subtract 1 for each block compound tail END.

In other words, the lexicographic level for a given declaration is equal to the block nesting count, starting with 2 for the outer program block.

SWITCH DECLARATIONS.

SYNTAX.

The syntax for ⟨switch declaration⟩ is as follows:

⟨switch declaration⟩ ::= SWITCH ⟨identifier⟩ ⟨replacement operator⟩ ⟨switch list⟩

⟨switch list⟩ ::= ⟨designational expression⟩ | ⟨switch list⟩ ,
⟨designational expression⟩

Examples:

```
SWITCH CHOOSEPATH ← L1, L2, L3, L4, SW1[3], LAB
SWITCH SELECT := START, ERRORI, CHOOSEPATH [I + 2]
```

SEMANTICS.
A SWITCH declaration defines an identifier to represent a set of designational expressions. These values are the designational expressions in the switch list. Associated with each designational expression in the switch list (from left to right) is an ordinal number from 1 to N (where N is the number of designational expressions). This integer indicates the position of the designational expression in the switch list. The value of the switch designator corresponding to a given value of the subscript expression (see pages 4-17 through 4-19, designational expressions) determines which designational expression is selected from the switch list. The designational expression thus selected supplies a label in the program to which control is transferred.

EVALUATION OF EXPRESSIONS IN THE SWITCH LIST. An expression in the switch list is evaluated each time it is selected using the current values of the variables from which it is composed.

INFLUENCE OF SCOPE.
If a quantity appears in a designational expression of a switch list and a switch designator selects the above-mentioned designational expression outside the scope of this quantity, the quantity which would otherwise be inaccessible to the switch designator will be used in the evaluation of the selected designational expression.

Examples:

```
BEGIN
    BOOLEAN B;
```

```
        LABEL L1, L2, L3, L4, L5;
        SWITCH SW ← L1, L2, L3, IF B THEN L4 ELSE L5;
        S;

    BEGIN
        INTEGER B;
        S;
        GO TO SW [4];

    END;
        S;

    END
```

DEFINE DECLARATIONS AND INVOCATIONS.

SYNTAX.

The syntax for ⟨define declaration⟩ is as follows:

⟨define declaration⟩ ::= DEFINE ⟨definition list⟩

⟨definition list⟩ ::= ⟨definition⟩ | ⟨definition list⟩ , ⟨definition⟩

⟨definition⟩ ::= ⟨defined identifier⟩ ⟨formal symbol part⟩ = ⟨text⟩ #

⟨defined identifier⟩ ::= ⟨identifier⟩

⟨formal symbol part⟩ ::= ⟨empty⟩ | (⟨formal symbol list⟩)

⟨formal symbol list⟩ ::= ⟨formal symbol⟩ | ⟨formal symbol list⟩ , ⟨formal symbol⟩

⟨formal symbol⟩ ::= ⟨identifier⟩

⟨text⟩ ::= {any sequence of valid symbols not including free #}

The syntax for ⟨invocation⟩ is as follows:

⟨invocation⟩ ::= ⟨defined identifier⟩ ⟨actual text part⟩

⟨actual text part⟩ ::= ⟨empty⟩ | (⟨closed text list⟩) |
[⟨closed text list⟩]

⟨closed text list⟩ ::= ⟨closed text⟩ | ⟨closed text list⟩ ,
⟨closed text⟩

⟨closed text⟩ ::= {an actual text not containing unmatched
bracketing symbols or unbracketed commas}

Examples:

Define Declaration:

```
DEFINE FORI = FOR I←1 STEP 1 UNTIL#,ADDUP = AxB+C/D#
```

Definition List:

```
MOVER = ← #
SPLIT = GO TO #, LOOK (LOOK1, LOOK2) = IF Q = "." THEN
  LOOK1 ← TRUE ELSE LOOK1 ← LOOK2 ← FALSE #
```

Definition:

```
LOOP1(LOOP11,LOOP12,LOOP13) = FOR LOOP11 ← LOOP12
  STEP 1 UNTIL LOOP13#
```

Formal Symbol List:

```
IDENTIFIERONE, TWO
ONLY
```

Text:

```
(
PROCEDURE
ANYID
IF A THEN GO TO SOUTH ELSE BEGIN X-ZxQ;GO TO NORTH END EG;
```

Invocation:

```
FLOO
GUARANTY(X-Y+1)
```

Actual Text Part:

```
(ERGO)
(X-1;GO TO L;)
[U+V, M-N]
```

SEMANTICS.

The DEFINE declaration assigns the meaning of the defined identifiers. An invocation causes the replacement of the defined identifier being invoked by the text which is associated with the identifier.

If the definition of a defined identifier included any formal symbols, any appearance of these symbols in the text of the definition (but not in a string or comment) will be replaced by the corresponding actual texts. Formal symbol identifiers must be constructed by appending integer digits to the defined identifier. The integer digits must correspond to the location of the formal symbol in the list (i.e., D(D1,D2,D3,...)).

The word COMMENT is recognized in a text. It and all characters up to and including the next semicolon are deleted from the text. No text may include an incomplete comment.

In a closed text list, the closed texts are separated by commas, and the closed text list is terminated by a right parenthesis or bracket.

In a closed text, a comma may appear only between matching bracketing symbols. No unmatched bracketing may appear.

The scope of a formal symbol is the text of the definition in which the formal symbol appears.

Bracketing symbols are [ ], ( ), and the group consisting of:

```
DEFINE =  # ;
```

At declaration time, a definition is of no consequence; it has meaning only in relation to the context in which its related defined identifier appears. For this reason, undeclared identifiers may appear in definitions; all identifiers must have been declared, however, when the defined identifier is used.

During compilation, syntax errors (if any) in a definition are noted following the use of the defined identifier.

NESTING OF DEFINITIONS. Definitions can be nested; that is, defined identifiers may be used in definitions. For instance, in the example below, the definition for D3 is equivalent to the definition for DD. In the example, the definition +A+A is considered nested one level in the first declaration. In the second declaration, the definition +A+A is considered nested two levels, and so forth.

Example:

```
DEFINE D1 = +A+A#
DEFINE D2 = D1 D1 #
DEFINE D3 = D2 D2 #
DEFINE DD = +A+A +A+A +A+A +A+A #
```

RESTRICTIONS.
A definition cannot be nested more than eight levels. Defined identifiers may not be used in a FORMAT or SWITCH FORMAT declaration. No more than nine ⟨formal symbol⟩s may be used in a ⟨formal symbol list⟩. If a definition ends with the word END, its defined identifier may be followed in the program only by a semicolon or the words ELSE, END, or UNTIL. The maximum number of characters (excluding the COMMENTs and superfluous blanks*) that may appear in a single definition may range from 1971 to 2035, depending upon the number of characters in the defined identifier, as follows:

IDENTIFIER SIZE	MAXIMUM
1-5	2034
6-13	2027

* Blanks are superfluous except in strings or when used as delimiters.

IDENTIFIER SIZE	MAXIMUM
14-21	2019
22-29	2011
30-37	2003
38-45	1995
46-53	1987
54-61	1979
62-63	1971

FORWARD REFERENCE DECLARATION.

SYNTAX.

The syntax for ⟨forward reference declaration⟩ is as follows:

⟨forward reference declaration⟩ ::= ⟨forward procedure declaration⟩ | ⟨forward switch declaration⟩

⟨forward procedure declaration⟩ ::= ⟨procedure type⟩ PROCEDURE ⟨procedure heading⟩ FORWARD

⟨forward switch declaration⟩ ::= SWITCH ⟨identifier⟩ FORWARD

Examples:

```
SWITCH SELECT FORWARD
INTEGER PROCEDURE SUM (A,B,C); VALUE A,B,C; INTEGER A,B,C;
  FORWARD
```

SEMANTICS.

Before a procedure of a switch can be called in a program, it must have been declared previously. A contradiction arises in two special cases, namely:

a. When a procedure calls another procedure, which in turn references the first procedure.

b. When a switch references another switch, which in turn references the first switch.

In such cases, the first PROCEDURE declaration must contain at least one reference to the second, as yet undeclared at this point; a similar situation would occur in the case of switches used in this way.

To enable the programmer to use such recursive references, the FORWARD construct has been introduced. This is, in effect, a temporary declaration and does not eliminate the need for the normal PROCEDURE and SWITCH declarations which must follow in the program.

I/O DECLARATIONS.

SYNTAX.

The syntax for ⟨I/O declaration⟩ is as follows:

⟨I/O declaration⟩ ::= ⟨file declaration⟩ | ⟨format declaration⟩
| ⟨list declaration⟩ | ⟨switch format declaration⟩
⟨switch file declaration⟩ | ⟨switch list declaration⟩

SEMANTICS.

I/O declarations describe the environment in which input to and output from a program must be handled.

FILE DECLARATIONS.

SYNTAX.

The syntax for ⟨file declaration⟩ is as follows:

⟨file declaration⟩ ::= ⟨file lock part⟩ ⟨mode part⟩ FILE
⟨in-out part⟩ ⟨file identifier⟩ ⟨label equation
part⟩ (⟨buffer part⟩ ⟨save factor⟩)

⟨file lock part⟩ ::= ⟨empty⟩ | SAVE

⟨mode part⟩ ::= ⟨empty⟩ | ALPHA

⟨in-out part⟩ ::= IN | OUT | ⟨empty⟩

⟨file identifier⟩ ::= ⟨identifier⟩

⟨label equation part⟩ ::= ⟨output media part⟩ ⟨disk file description⟩ ⟨label part⟩

⟨output media part⟩ ::= ⟨output media digit⟩ | ⟨empty⟩ | DISK ⟨disk access technique⟩ | REMOTE

⟨output media digit⟩ ::= ⟨arithmetic expression⟩ | *

⟨label part⟩ ::= ⟨file identification part⟩ | ⟨multi-file identification part⟩ ⟨file identification part⟩ | ⟨empty⟩

⟨disk file description⟩ ::= ⟨empty⟩ | [⟨number of areas⟩ : ⟨size of areas⟩]

⟨number of areas⟩ ::= ⟨arithmetic expression⟩

⟨size of areas⟩ ::= ⟨arithmetic expression⟩

⟨disk access technique⟩ ::= SERIAL | RANDOM | UPDATE | ⟨empty⟩

⟨file identification part⟩ ::= "{7 or less string characters}"

⟨multi-file identification part⟩ ::= "{7 or less string characters}"/

⟨buffer part⟩ ::= ⟨number of buffers, ⟨record specifications⟩

⟨number of buffers⟩ ::= ⟨unsigned integer⟩

⟨record specifications⟩ ::= ⟨unblocked specification⟩ | ⟨blocking specifications⟩

⟨unblocked specification⟩ ::= ⟨fixed physical record size⟩

⟨blocking specifications⟩ ::= ⟨fixed logical record size⟩, ⟨fixed physical record size⟩ | ⟨fixed physical record .size⟩, ⟨fixed logical record size⟩

⟨fixed logical record size⟩ ::= ⟨arithmetic expression⟩

⟨fixed physical record size⟩ ::= ⟨arithmetic expression⟩

⟨save factor⟩ ::= , SAVE ⟨arithmetic expression⟩ | ⟨emtpy⟩

Examples:

```
FILE IN REED (1, 10)
FILE OUT RITE (2, 15)
FILE OUT RITE 1 (2, 15)
FILE OUT CARDS (2, 10)
FILE OUT CARDS 0 (2, 10)
FILE IN TAPE (2, 300, 40)
ALPHA FILE OUT TAPEOUT 2 (2, 400, 45)
SAVE FILE TAPE10 (2, 40)
FILE FILEID "IDENTI" (2, 350, 25)
SAVE ALPHA FILE OUT FILEID 2 "MULTIFI"/"IDENTIF" (2, 470,
  35, SAVE 25)
ALPHA FILE OUT F18 (1, 10)
ALPHA FILE IN DATACOM 14 (2, 29)
ALPHA FILE OUT REPLY REMOTE (5, 5)
FILE IN RIED DISK SERIAL (2, 30)
FILE IN RANRIED DISK RANDOM (1, 60, 180)
FILE RANRW DISK RANDOM [3:6000] (1, 30, 120)
FILE OUT NEW DISK SERIAL [4:2000] "A123456" (3, 12, 180)
FILE UPD DISK UPDATE [N:S] "PREFIX"/"FILEID" (A, B, C)
SAVE FILE ID DISK SERIAL [3:3000 "PART"/"REC" (3, 30, 120,
  SAVE 30)
```

SEMANTICS.

The FILE declaration associates a file identifier with the specifications which govern the handling of that file.

Upon exit from the block in which a file is declared, the file is closed and related I/O units are released to the system. Tape units, if any, are rewound.

The file lock part causes the implied execution of a LOCK statement upon the file when exiting the block in whose head the file declaration is made.

The mode part may be included in the declaration of a file using magnetic tape and data communications; in all other cases, it should be empty. For magnetic tape files, ALPHA is used to specify that records recorded with even parity are to be written on an output file or read from an input file. Records recorded with odd parity on tape files are assumed if the mode part is empty.

If the mode part specifies ALPHA on a data communications file, then the I/O channel will perform a BCL-to-internal translation on each READ statement and an internal-to-BCL translation on each WRITE statement. A data transmission control unit is required to ensure that automatic terminal code-to-BCL or BCL-to-terminal code translation takes place. The absence of a control unit would require programmatic translation; therefore, the mode part would be left empty to inhibit automatic I/O translation.

The in/out part may contain IN or OUT, or may be empty.

In the case of tape files which are both used for output and input in the same program, the in/out part must be empty.

The in/out part designates the type of action to be taken when the buffer is released if the buffer had been opened by other than a READ, SPACE, or WRITE statement. If no direction is stated, it will be interpreted as IN.

All file identifiers in a program should be unique. The file identifier is used in the program; and in Program Parameter cards, it references the declared file.

The label equation part has the same function as a Label Equation card and may be used in lieu of the card. If a label equation part and Label Equation card are both used, the card takes precedence.

The output media part specifies the output medium. With the exception of the SPO and data communications, the output media is ignored on <u>input</u> files and should be left empty. The digits used

in the output media part are shown in table 6-2. An output media part of 11 must be used on both input and output files referencing the SPO. An output media part of 14 or REMOTE must be used on both input and output files referencing the data communications unit.

If the output media part is left empty, a 2 is assumed for output files.

The label part serves to designate the identifier in the label of a particular file which differs from the declared file identifier. It also indicates use of a multi-file reel. Data communications files do not have labels; but if they are used, they have no effect on program execution.

The buffer part specifies the number of buffer areas desired and the size (number of words) needed for each buffer area. When the file is referencing the SPO, the input message is assumed to be 80 characters in length. Consequently, all SPO file buffer sizes must always be at least 10 words long. The buffer size of a data communications file should be declared to be 10 words long.

The information in one punched card requires a buffer of 10 words. A buffer of 15 or 17 words is required for one line of print on the 120 or 132 position line printers, respectively.

If more than one buffer is specified and storage is inadequate to accommodate the number designated, the program cannot be executed. For data communications input files, only one buffer will be used, regardless of the value of ⟨number of buffers⟩.

Blocked records may be read or written when using magnetic tape or disk files. This is specified by the ⟨record specifications⟩ of the file declaration. The ⟨fixed logical record size⟩ specifies the number of words for each record, and the ⟨fixed physical record size⟩ specifies the number of words in the entire block. The block size depends on the type of blocking used and should be

determined as described in the following paragraph.

When using magnetic tape files, two types of blocking may be used:

a. If the ⟨record specifications⟩ is of the form ⟨fixed logical record size⟩, ⟨fixed physical record size⟩, then the block size will be a multiple of the record size. For example, a file declaration such as FILE OUT TAPE1 (2, 55, 550) would create a tape where there are 55 words to each record and 10 records per block, for a total block size of 550 words.

b. If the ⟨record specifications⟩ is of the form ⟨fixed physical record size⟩, ⟨fixed logical record size⟩, then the block size must be large enough to include link words. For example, to create a tape with the same blocking factor as the above example, the file declaration would be FILE OUT TAPE1 (2, 561, 55). The ⟨fixed physical record size⟩ must be a multiple of the logical record size plus the number of logical records plus one or 10 x 55 + 10 + 1 = 561. The additional 11 words are link words created by the MCP.

When the file declaration references a disk file, the blocking can only be of the form ⟨fixed logical record size⟩, ⟨fixed physical record size⟩. Each physical record will start at the beginning of a disk segment and may contain a maximum of 63 segments.

The SAVE factor is applicable to labeled magnetic tape output files and disk files that are entered into the disk directory. When a SAVE factor is used on tape files, the value of the arithmetic expression is added to the current date and included in the tape label as the purge date. When a SAVE factor is used on a disk file, the value of the arithmetic expression is added to the current date every day that the file is accessed, creating a dynamic purge date. A SAVE factor may be specified on a data communications file but has no effect.

The disk access technique used with disk files specifies the buffering action to be used with the file. Which technique to use is dependent on the primary purpose for accessing the file. The six basic purposes for accessing a file on disk are to:

a. Serially read records.
b. Serially write records.
c. Randomly read records.
d. Randomly write records.
e. Serially update records.
f. Randomly update records.

The file should be declared SERIAL if the primary purpose is either a or b above. The file should be declared RANDOM if the primary purpose is either c, d, or f above. If e is the primary purpose, the file should be declared UPDATE.

When a disk file is declared SERIAL, the following actions take place:

a. As READ statements are performed, reading is buffered. The buffers are filled with records of consecutively higher addresses than the record last accessed.

b. If the file is declared unblocked and a WRITE statement is performed, there is never a need for an implicit READ before writing, and writing is buffered.

c. If the file is declared blocked, if necessary, an implicit READ will be made before a WRITE statement is performed. This action is required since the entire physical record which contains the logical record must be written.

When a disk file is declared RANDOM, the following action takes place:

a. READ operations are buffered only through the use of a READ SEEK statement.

b. If the file is declared unblocked and a WRITE statement is performed, an implicit READ is not required and writing is buffered.

c. If the file is declared blocked and a WRITE is performed, the action taken is the same as for a serial disk file.

READ and WRITE statements which reference a random file must contain a record address.

When a disk file is declared UPDATE, buffer handling is designed to provide optimum handling of I/O statements that cause a record to be read but not released, and then updated and written. Each time a WRITE is performed, the buffer used for the output record is written and immediately refilled with the next record to be buffered in from disk. The buffers of the file are filled with records of consecutively higher addresses than the last record read and/or written.

The disk file description is used when a file on disk is being created. It consists of the ⟨number of areas⟩ and the ⟨size of areas⟩, each defined below.

a. The number of areas can have any value from 1 through 20. This specifies the maximum number of areas on the disk that the file may occupy.

b. The size of the areas specifies the size of each area that the file on disk may occupy. This size is in terms of the number of logical records that the area is to contain.

The total area that the file could occupy on disk is the number of areas times the size of each area. When more than one area is declared, the next area is not allocated until the preceding

area has been filled with the number of logical records specified by the size of the area.

RESTRICTIONS.
A program may contain more than one FILE declaration involving the same file identifier; however, no such file after the first may be accessed with a Label Equation card.

A file identifier may designate a file on a multi-file magnetic tape. More than one such file may be used in a program; however, no more than one file on a given multi-file tape may be open at any time.

A variable number of words may be contained in one magnetic tape block, but the number may not exceed 1023.

A disk file description should not be used with files declared IN.

If a file which exists on the disk is specified by a disk file declaration, the disk file description must be empty.

SWITCH FILE DECLARATIONS.
SYNTAX.
The syntax for ⟨switch file declaration⟩ is as follows:

⟨switch file declaration⟩ ::= SWITCH FILE ⟨switch file identifier⟩ ⟨replacement operator⟩ ⟨switch file list⟩

⟨switch file identifier⟩ ::= ⟨identifier⟩

⟨switch file list⟩ ::= ⟨file identifier⟩ | ⟨switch file list⟩, ⟨file identifier⟩

Examples:

```
SWITCH FILE SWHTAPE ← TAPE1, TAPE2, TAPE3
SWITCH FILE SWHUNIT:=CARDOUT, TAPEOUT, PRINT
```

SEMANTICS.

The SWITCH FILE declaration associates a switch file identifier with a number of files, as designated by the file identifiers in the switch file list.

Associated with each of the file identifiers in the switch file list is an integer reference. The references are 0, 1, 2, ..., obtained by counting the identifiers from left to right. This integer indicates the position of the file identifier in the list. The file identifiers are referenced, according to position, by switch file designators.

If the switch file designator yields a value which is outside the range of the switch file list, the file so referenced is undefined. Each file identifier used in a switch file list must have appeared previously in a prevailing FILE declaration and each file is governed according to the FILE declaration in which it was declared.

FORMAT DECLARATIONS.

SYNTAX.

The syntax for ⟨format declaration⟩ is as follows:

⟨format declaration⟩ ::= FORMAT ⟨input or output⟩ ⟨format part⟩

⟨input or output⟩ ::= IN | OUT | ⟨empty⟩

⟨format part⟩ ::= ⟨format identifier⟩ (⟨editing specifications⟩) | ⟨format part⟩, ⟨format identifier⟩ (⟨editing specifications⟩)

⟨format identifier⟩ ::= ⟨identifier⟩

⟨editing specifications⟩ ::= ⟨editing segment⟩ | ⟨editing specifications⟩ / | / ⟨editing specifications⟩ | ⟨editing specifications⟩ / ⟨editing segment⟩

⟨editing segment⟩ ::= ⟨editing phrase⟩ | ⟨repeat part⟩
(⟨editing specifications⟩) | ⟨editing segment⟩,
⟨editing phrase⟩ | ⟨editing segment⟩, ⟨repeat part⟩
(⟨editing specifications⟩)

⟨editing phrase⟩ ::= ⟨repeat part⟩ ⟨editing phrase type⟩
⟨field part⟩ | ⟨string⟩

⟨repeat part⟩ ::= ⟨empty⟩ | ⟨unsigned integer⟩ | *

⟨editing phrase type⟩ ::= A | D | E | F | I | L | O | R |
S | V | X

⟨field part⟩ ::= ⟨empty⟩ | ⟨field width⟩ | ⟨field width⟩
⟨decimal places⟩

⟨field width⟩ ::= ⟨unsigned integer⟩ | *

⟨decimal places⟩ ::= ⟨unsigned integer⟩ | *

Examples:

```
FORMAT IN EDIT (X4, 2I6, 5E9.2, 3F5.1, X4)
FORMAT IN F1 (A6,5(X3,2E10.2,2F6.1),3I7),F2(A6,D,A6)
FORMAT OUT FORM1 (X56, "HEADING",X57),FORM2 (X10,4A6/X7,
  5A6/X2,5A6)
FORMAT OUT F3 (10230)[1]
FORMAT OUT F4(F5.2, X2, R3.1, S-2)
FORMAT FMT1 (*I*)
FORMAT FMT2 (*V*.*)
```

SEMANTICS.

The FORMAT declaration associates a set of editing specifications with a format identifier. The following discussion of FORMAT declarations is divided into two parts: those used for input and those used for output.

1. The last character before the right parenthesis is the letter O, not zero.

INPUT EDITING SPECIFICATIONS. Input data can be introduced to the system by various media such as punched cards or magnetic tape. Once the information is in the system, however, it may be considered a string of bits, regardless of the input equipment used.

For editing purposes, this string can be processed in one or two ways: either as a set of six-bit characters (see appendix B, internal character codes), or an eight-character word. The input editing specifications, through the editing phrases, designate where and in what form the initial values of variables are to be found in this string.

INPUT EDITING PHRASES. The editing phrases, except the D and O types, designate six-bit character processing. They describe a portion of the input data in which the initial value of one variable is to be found. Editing phrases type D and O cause the input string to be processed as full eight-character words.

A phrase such as rAw has the same effect as Aw, Aw..., Aw(r times), where r is the repeat part and w the field width. The field width may specify from one to 63 characters. If the repeat part of an editing phrase is empty, it is given a value of 1.

Characteristics of the input editing phrase types are summarized in table 7-1.

The definition of each input editing phrase type in table 7-1 is given below.

a. A - initializes a variable to the characters found in the field described by the field width. If the field width is greater than six, the right-most six characters are taken as the value to be assigned to the variable. If the field width is less than six, zeros are appended to the left of the characters in the field to make a total of six characters.

Table 7-1

Characteristics of Types of Input Editing Phrases

Editing Phrase Type	Editing Phrase Example	Processed As	Type of Variable Being Initialized	Example of Field Contents
A	A6	6-bit characters	ALPHA	TOTALS
D	D	Full word	None	Any operand
E	E9.2	6-bit characters	REAL	+0.18@-03
F	F7.1	6-bit characters	REAL	-3892.5
I	I6	6-bit characters	INTEGER	+76329
L	L5	6-bit characters	BOOLEAN	FALSE
O	O	Full word	Any	Any operand
R	R11.4	6-bit characters	REAL	+2123123@+4
S	S-2	6-bit characters	REAL	None
X	X7	6-bit characters	None	Any 7 characters

b. D - causes one full word of eight characters in the input data string to be ignored. The field part should be empty.

c. E - initializes a variable to the number found in the field described by the field width. The field width must be at least seven greater than the number of decimal places specified since the input data is required to be of the following form:

$$\pm n.dd---d@\pm ee$$

The sign of the number must appear first. A digit and a decimal point must follow the sign. One or more digits may follow the decimal point. The number of digits following the decimal point must equal the number of decimal places indicated by the editing

phrase. Following the digits must be the symbol @, the sign of the exponent, and a two-digit exponent. The sign of the number may be indicated by +, -, or a single space which is interpreted as positive. The number must be right-justified in the designated field.

d. F - initializes a variable to the number found in the field described by the field width. The input data must be in one of the following forms:

$\pm$nn---n.	$\pm$nn---n.dd-d	nn---n.dd---d
nn---n.	$\pm$.dd---d	.dd---d

The sign of the number is optional. If there is a sign, it must appear first; if there is no sign, the number is assumed to be positive. A decimal point must be present; zero or more digits may precede it. There must be as many digits after the decimal point as specified by the editing phrase. The number must be right-justified in the designated field.

e. I - initializes a variable to the integer found in the field described by the field width. The sign of the number is optional; the applicable rules are the same as in the case of editing phrase F.

The number itself may consist of one or more digits which must be right-justified in the designated field.

f. L - initializes a variable to the logical value found in the field described in the field width. There are two possible values, TRUE and FALSE; the programmer may truncate these input words as shown in table 7-2.

Table 7-2

Boolean Values for Various Field Widths in Input Editing Phrase

Editing Phrase	Boolean Value	
	TRUE	FALSE
L1	T or b	F
L2	TR or bT	FA
L3	TRU or bTR	FAL
L4	TRUE or bTRU	FALS
L5	TRUEb or bTRUE	FALSE
Ln, where n > 5	Skip n-5 then same as L5	

g. O - initializes a variable to the contents of an eight-character word taken from the input string. The field part is ignored and should be left empty.

h. R - initializes a variable to the contents of an input field which may be written according to the specifications of the I, F, or E editing phrase. A decimal point as implied in the editing phrase is sufficient; its location is considered to be as many digit-positions to the left, from the right-most position of the field, as indicated by d in the editing phrase. An actual decimal point in the input takes precedence over the implied decimal point. If there is an actual decimal point in the input, the input data may appear anywhere within the field. No explicit sign is required in either the characteristic or the mantissa; allowed exponents range from -68 to +68. If the input field is a field of blanks, a -0 (minus zero) is generated. The d indicator of the editing phrase is ignored if the input consists only of an exponent part. The symbol & may be used in place of +, and E in place

of @. An error condition transfers control to the parity action label, if one is present; otherwise, the program will be terminated.

i. S - the integer number in the editing phrase itself is used as a power of 10 to multiply all values associated with subsequent R editing phrases. More than one S phrase may appear in a format, each taking precedence over the one before.

j. V - causes an access to the list during the program execution to determine the ⟨editing phrase⟩ type. The value obtained from the list should be one of the characters A, D, E, F, I, L, O, R, S, or X.

k. X - causes the number of characters indicated by the field width to be ignored.

If the input editing phrase is a string, the string in the FORMAT declaration is replaced by the corresponding input string. The number of characters transferred from the input string is equal to the number of characters in the FORMAT declaration which are enclosed between the string bracket characters. If the editing phrase is not D or O, the field part must not be empty.

If the ⟨repeat part⟩, ⟨field width⟩, or ⟨decimal places⟩ of an ⟨editing phrase⟩ is an asterisk (*), the value of the next list element during execution of the program will be used to complete the definition of the ⟨editing phrase⟩. If the value of the list element corresponding to the repeat part is less than or equal to O, the editing phrase will be skipped. If the repeat part preceding a left parenthesis is an asterisk, the number of repetitions is determined by the value of the corresponding list element as follows:

a. If the value is greater than O, then repeat the number of times of the value.

b. If the value is equal to 0, then repeat indefinitely.

c. If the value is less than 0, then skip to the corresponding right parenthesis.

Examples of the above and the V editing phrase are shown below.

```
.
.
FORMAT FMT1 (*I*);
FORMAT FMT2 (*V*.*);
.
.
READ (INPUT, FMT1, 2, 4, A, B);
.
.
WRITE (LINE, FMT2, 3, "F", 6, 4, X, Y, Z);
.
.
```

The READ causes FMT1 to be executed as 2I4, while the WRITE causes FMT2 to be executed as 3F6.4.

When a READ statement uses a free-field part, no FORMAT declaration is required to provide the editing specifications for data. Editing specifications, in this case, are determined by the format of the data. Such data must be formatted as described on page 6-19.

OUTPUT EDITING SPECIFICATIONS. Output can be performed by the system through various media such as magnetic tape and the line printer. The information in the system, ready for output but not yet transferred to the output equipment, may be considered a string of bits, regardless of the output equipment to be used. For editing purposes, this string can be built in one of two ways: either from a set of six-bit characters (see appendix B), or from a set of eight-character full words. The output editing specifications, by means of the editing phrases, designate where and in what forms the values of expressions are to be placed in this string.

OUTPUT EDITING PHRASES. The editing phrases, except D and O types, designate six-character processing. They describe a portion of the

output data string into which output information is to be placed. This information may be one of three kinds:

a. The value of an expression.

b. The characters of the editing phrase itself (when the editing phrase is a string).

c. The insert characters 0 (zero) and single space.

Editing phrase types D and O designate that the output string is to be built from full words. The field width may specify a length of one to 63 characters. The expression rAw has the same effect as Aw, Aw,..., Aw (r times), where r is the repeat part and w is the field width. If the repeat part of an editing phrase is empty, it is given a value of 1. Characteristics of the output editing phrase types are summarized in table 7-3.

The definition of each output editing phrase is given below.

a. A - places the value of one expression (six characters) in the field width. If the field width is greater than six, the six characters are placed at the right end of the field and leading blanks are inserted to fill out the field. If the field width is less than six, the right-most characters of the expression value are placed in the field.

b. D - places one full word of all zeros in the output data string.

c. E - places the value of one expression in the field described by the field width. This value has the following form when placed in the output data string:

$$^{b}_{-}n.dd---d@^{+}_{-}ee$$

Table 7-3

Characteristics of Types of Output Editing Phrases

Editing Phrase Type	Editing Phrase Example	Processed As	Type of Evaluated Expression	Example of Field Contents
A	A6	6-bit characters	ALPHA	RESULT
D	D	Full word	None	One full word of zeros
E	E11.4	6-bit characters	REAL	-1.2500@+02
F	F8.3	6-bit characters	REAL	6735.125
I	I6	6-bit characters	INTEGER	bb1416
L	L5	6-bit characters	BOOLEAN	bTRUE
O	O	Full word	Any	Any operand
R	R11.4	6-bit characters	REAL	b2.1231@+09
S	S-2	6-bit characters	REAL	None in field; result: (10*(-2)) x R (subsequent)
X	X8	6-bit characters	None	8 blanks

The sign of the number is represented by a single space if positive and a minus sign if negative ($\overset{b}{-}$ = blank or minus). If the field width is more than seven greater than the number of decimal places specified, leading single spaces are used to complete the field. Then the sign of the number, the first significant digit, and a decimal point are inserted. The value of the expression is rounded to the number of decimal places specified by the editing phrase. If the number of significant digits in the expression value is less than the number of decimal places specified, the digits are left-justified with trailing zeros. To complete the field, the symbol @, the sign of the exponent, and the appropriate two-digit

exponent are inserted. The sign of the exponent is indicated by either + or -.

d. F - places the value of one expression in the field described by the field width. This value has the following form when placed in the output string:

$$^{b}_{-}nn---n.dd--d$$

The expression value is rounded to the number of designated decimal places. If the number is smaller than the field specified, it is placed in the field right-justified. If the number of digits equals the number of places specified and if the number is:

1) Positive, it will be placed in the field without a sign.

2) Negative, the entire field will be filled with asterisks (*).

If the number is greater than the field specified, the entire field will be filled with asterisks. The sign is treated as in editing phrase E.

e. I - places the value of one expression in the field described by the field width. The expression value is rounded to an integer and placed right-justified in the field, preceded by leading single-spaces, if any are required. If the number is greater than the maximum allowable integer, the entire field will be filled with asterisks. The sign is treated as in editing phrase F.

f. L - places the value of one Boolean expression in the field designated by the field width. Table 7-4 shows the effect of various values of field width.

Table 7-4

Boolean Values for Various Field Widths in Output Editing Phrase

Field Width	Boolean Value	
	TRUE	FALSE
L1	T	F
L2	TR	FA
L3	TRU	FAL
L4	TRUE	FALS
L5	TRUEb	FALSE
Ln, where n > 5	Skip n-5 then same as L5	

g. O - places the value of one expression, in full word form, in the output string.

h. R - places the value of one expression in the field described by the field width. The output will be either an F-type or an E-type field, depending upon the magnitude of the expression. Assuming that:

E = exponent number,

sign = 0 for +, 1 for -,

w = total field width,

d = number of decimal places to the right of decimal point, and

I = number of decimal digits to the left of decimal point, then:

1) The output will be in F-format if the absolute value of the number is equal to or greater than 1 but less than the maximum allowable integer, and

$$w \geq I + d + 1 + sign$$

or if the absolute value of the number is less than 1, and

$$w \geq d + 1 + sign$$

and either

$$ABS(E) \leq d$$

or

$$w < d + 6 + sign$$

2) The output will be in E-format if the conditions for F-format are not met, and

$$w \geq d + 6 + sign$$

3) If none of the above conditions are fulfilled, the field will be filled with asterisks.

i. S - the values associated with the subsequent R format phrases will be multiplied by such powers of 10 as designated by the integer in the S format phrase itself. More than one S phrase may appear in a format, each taking precedence over the one before.

j. V - causes an access to the list during program execution to determine the ⟨editing phrase⟩ type. The value obtained from the list should be one of the characters A, D, E, F, I, L, O, R, S, or X.

k. X - places a number of single spaces, as indicated by the field width, in the output string.

An output editing phrase may itself be a string; this editing phrase is defined as placing itself, except for the delimiting string bracket characters, in the output string.

If the ⟨repeat part⟩, ⟨field width⟩, or ⟨decimal places⟩ of an ⟨editing phrase⟩ is an asterisk (*), the value of the next list element during execution of the program will be used to complete the definition of the ⟨editing phrase⟩. If the value of the list element corresponding to the repeat part is less than or equal to 0, the editing phrase will be skipped. If the repeat part preceding a left parenthesis is an asterisk, the number of repetitions is determined by the value of the corresponding list element as follows:

a. If the value is greater than 0, then repeat the number of times of the value.

b. If the value is equal to 0, then repeat indefinitely.

c. If the value is less than 0, then skip to the corresponding right parenthesis.

Examples of the above and the V editing phrase are shown below.

```
.
.
FORMAT FMT1 (*I*);
FORMAT FMT2 (*V*.*);
.
.
READ (INPUT, FMT1, 2, 4, A, B);
.
.
WRITE (LINE, FMT2, 3, "F", 6, 4, X, Y, Z);
.
.
```

The READ causes FMT1 to be executed as 2I4, while the WRITE causes FMT2 to be executed as 3F6.4.

RESTRICTION. In editing phrases O and D the field part must be empty; in all other cases it must not be empty.

THE MEANING OF THE SYMBOL /. The / (slash) used in editing specifications causes output from, and clearing of, the buffer. The buffer is cleared by filling it with single spaces. The right-most

parenthesis of the editing specification performs the function of one slash. When the line printer is used, consecutive slashes cause vertical spacing of the printer by printing blank lines. It should be taken into account, however, that the first slash will cause the actual contents of the buffer to be printed.

SWITCH FORMAT DECLARATIONS.

SYNTAX.

The syntax for ⟨switch format declaration⟩ is as follows:

⟨switch format declaration⟩ ::= SWITCH FORMAT ⟨switch format identifier⟩ ⟨replacement operator⟩ ⟨switch format list⟩

⟨switch format identifier⟩ ::= ⟨identifier⟩

⟨switch format list⟩ ::= (⟨editing specifications⟩) | ⟨switch format list⟩, (⟨editing specifications⟩)

Examples:

```
SWITCH FORMAT SF ← (A6, 3I4, I2, X60),(I4,X2,2I4,3I2),
                   (X78,I2),(X2);
SWITCH FORMAT SWHFT ← (X78,I2),(4A6,I2),(10A6,I2);
```

SEMANTICS.

The SWITCH FORMAT declaration associates a switch format identifier with the editing specifications in the switch format list.

Associated with each of the editing specification parts is an integer reference starting from 0, obtained by counting the editing specifications from left to right. This integer reference indicates the position of the editing specification part in the list. The editing specifications are referenced according to position, by switch format designators.

If a switch format designator yields a value which is outside the range of the switch format list, the format so referenced is undefined.

LIST DECLARATIONS.

SYNTAX.

The syntax for ⟨list declaration⟩ is as follows:

⟨list declaration⟩ ::= LIST ⟨list part⟩

⟨list part⟩ ::= ⟨list identifier⟩ ((⟨list⟩)) | ⟨list part⟩, ⟨list identifier⟩ ((⟨list⟩))

⟨list identifier⟩ ::= ⟨identifier⟩

⟨list⟩ ::= ⟨list segment⟩ | ⟨list⟩, ⟨list segment⟩

⟨list segment⟩ ::= ⟨expression part⟩ | ⟨for clause⟩ ⟨list segment⟩ | ⟨for clause⟩ [⟨expression list⟩]

⟨expression part⟩ ::= ⟨arithmetic expression⟩ | ⟨Boolean expression⟩

⟨expression list⟩ ::= ⟨list segment⟩ | ⟨expression list⟩, ⟨list segment⟩

Examples:

```
LIST L1 (X,Y,A[J], FOR I ← P STEP 1 UNTIL 5 DO B [I])

LIST ANSWERS (P + Q,Z,SQRT (R)), RESULTS (X1,X2,X3,X4/2)

LIST LIST3 (FOR I ← 0 STEP 1 UNTIL 10 DO FOR J ← 0 STEP 1
            UNTIL 15 DO A [I,J])

LIST L4 (B AND C, NOT AB1, IF X = 0 THEN R1 ELSE R2)

LIST RESULTS (FOR I ← 1 STEP 1 UNTIL N DO [A[I], FOR J ← 1
              STEP 1 UNTIL K DO[B[I,J], C[J]]])
```

SEMANTICS.

A LIST declaration serves to associate a set of expressions (arithmetic or Boolean) with a list identifier. A list identifier may be used in a READ statement (pages 6-16 through 6-23) for specifying

the variables to be initialized and the order in which the initializing is to be done. Since input may not be made to any construct other than a variable, a list identifier used in a READ statement must refer to a LIST declaration which includes variables only. The variables in a LIST declaration must have been previously declared as to type.

The list identifier may be used in a WRITE statement (pages 6-24 through 6-26) for specifying values to be included in an output operation. These values are placed in the output string in the order of their appearance in the LIST declaration. Variables in a LIST declaration may be either local or nonlocal to the block in which the LIST declaration appears.

SWITCH LIST DECLARATIONS.

SYNTAX.

The syntax for ⟨switch list declaration⟩ is as follows:

⟨switch list declaration⟩ ::= SWITCH LIST ⟨switch list identifier⟩ ⟨replacement operator⟩ ⟨switch list list⟩

⟨switch list identifier⟩ ::= ⟨identifier⟩

⟨switch list list⟩ ::= ⟨list identifier⟩ | ⟨switch list designator⟩ | ⟨list identifier⟩, ⟨switch list list⟩ | ⟨switch list designator⟩, ⟨switch list list⟩

Examples:

```
SWITCH LIST LX1 ← L1, L2, L3
SWITCH LIST LX2 ← L1, LX1 [1], L3
```

SEMANTICS.

A SWITCH LIST declaration associates a switch list identifier with a number of list identifiers. Associated with each of the list identifiers is an integer reference which is obtained by counting the list identifiers from left to right starting with 0. This

integer indicates the position of the list identifier in the switch list. These list identifiers are referenced my means of switch list designators.

If a switch list designator yields a value which is outside the range of the switch list, the list so referenced is undefined. Each list used in the switch list must have been previously declared.

MONITOR DECLARATIONS.

SYNTAX.

The syntax for ⟨monitor declaration⟩ is as follows:

⟨monitor declaration⟩ ::= MONITOR ⟨monitor part⟩

⟨monitor part⟩ ::= ⟨file identifier⟩ (⟨monitor list⟩) |
⟨monitor part⟩, ⟨file identifier⟩ (⟨monitor list⟩)

⟨monitor list⟩ ::= ⟨monitor list element⟩ | ⟨monitor list⟩,
⟨monitor list element⟩

⟨monitor list element⟩ ::= ⟨simple variable⟩ | ⟨subscripted variable⟩ | ⟨array identifier⟩ | ⟨switch identifier⟩ | ⟨procedure identifier⟩ | ⟨label⟩

Example:

```
MONITOR ANSWER (A,Q[I,J], GROUP1, START,SELECT,INTEGRATE)
```

SEMANTICS.

The diagnostic declaration MONITOR declares certain quantities to be placed under surveillance during the execution of the program. Each time an identifier included in the monitor list is used in one of the ways described below, the identifier and its current value are written on the file indicated in the MONITOR declaration.

MONITOR LIST ELEMENTS. When a simple variable in the monitor list is used as a left part in an assignment statement, the following information is written on the designated file:

⟨simple variable⟩ = {value of variable}

When a subscripted variable in the monitor list is encountered during the execution of the program as the left-most element in a left part list, the following information is written on the designated file:

⟨array identifier⟩[{value of subscript expression}] =
{value of variable}

When only an array identifier is given in the monitor list, and a subscripted variable of that array is encountered as the left-most element in a left part list, the following information is written on the designated file:

⟨array identifier⟩[{value of subscript expression}] =
{value of variable}

When a switch designator is encountered with a switch identifier which is in the monitor list, the following information is written on the designated file:

⟨switch identifier⟩

When a procedure identifier in the monitor list is used as a function designator during the execution of a program, the following information is written on the designated file:

⟨procedure identifier⟩ = {value of function designator}

Each time a label which is in the monitor list is encountered in the program, the label is written on the designated file.

RESTRICTIONS.

Only the first seven characters of any identifier are written. All pertinent subscripts, however, are written. Only one subscripted variable from an array may be monitored at one time. If a monitor list, or several monitor lists, contain more than one subscripted

variable which are elements of the same array, only the last of these is monitored.

DUMP DECLARATIONS.

SYNTAX.

The syntax for ⟨dump declaration⟩ is as follows:

⟨dump declaration⟩ ::= DUMP ⟨dump part⟩

⟨dump part⟩ ::= ⟨file identifier⟩ (⟨dump list⟩)
⟨label⟩:⟨dump indicator⟩ | ⟨dump part⟩,
⟨file identifier⟩ (⟨dump list⟩) ⟨label⟩:
⟨dump indicator⟩

⟨dump list⟩ ::= ⟨dump list element⟩ | ⟨dump list⟩, ⟨dump list element⟩

⟨dump list element⟩ ::= ⟨simple variable⟩ | ⟨subscripted variable⟩ | ⟨label⟩ | ⟨array identifier⟩

⟨dump indicator⟩ ::= ⟨unsigned integer⟩ | ⟨simple variable⟩

Example:

```
DUMP INPUTDATA (A,Q[I,J],GROUP1,START) ENTER:4,
     OUTPUTDATA (A,GROUP1) EXIT:X
```

SEMANTICS.

The DUMP declaration declares certain quantities to be placed under surveillance during the execution of the program. Diagnositc information requested by means of the DUMP declaration is written on the designated file when a label in the dump part has been passed the number of times equal to the associated dump indicator.

Since the dump indicator can be a simple variable, dump information can be obtained more than once during each execution of the block containing the DUMP declaration. The number of times the controlling statement is executed applies only to one pass through the

DUMP declaration block. The number is not cumulative from one pass to the next.

DUMP LIST ELEMENTS. A simple variable in the dump list causes the current value of that variable to be supplied in the following form:

⟨simple variable⟩ = {value of variable}

A subscripted variable in the dump list causes the current value of that variable to be supplied in the following form:

⟨array identifier⟩ [{value of subscript expression}] =
{value of variable}

An array identifier in the dump list causes the current values of all elements in that array to be supplied in the following form:

⟨array identifier⟩ = {value of first six elements}
{value of second six elements}
.
.
.
{value of last elements}

The order in which the array elements are written is as follows. All subscripts are first set to their declared lower bounds and the corresponding value is printed out. The right-most subscript is then counted up, and the corresponding value is printed; this procedure continues until the subscript reaches its declared upper bound. After this printout, the right-most subscript is again set to its declared lower bound, the next left subscript is counted up, and the process recycles until all subscripts have reached their declared upper bounds.

RESTRICTION.
Only the first seven characters of any identifier are written. All pertinent subscripts, however, are written.

FAULT DECLARATIONS.

SYNTAX.

The syntax for ⟨fault declaration⟩ is as follows:

⟨fault declaration⟩ ::= MONITOR ⟨fault list⟩

⟨fault list⟩ ::= ⟨fault type⟩ | ⟨fault list⟩ , ⟨fault type⟩ |
⟨fault list⟩, ⟨fault equate⟩

⟨fault type⟩ ::= EXPOVR|INTOVR|INDEX|FLAG|ZERO

⟨fault equate⟩ ::= ⟨fault type⟩ ← ⟨identifier⟩

Example:

MONITOR INTOVR, ZERO, FLAG ← PENNANT

SEMANTICS.

The fault declaration allows the programmer to indicate to the Compiler that he wishes to specify, via a fault statement, action to be taken upon the occurrence of one of the errors included in the fault list.

The fault list may include from one to five fault type identifiers.

Each fault type identifier is associated with a specific program error, as indicated in table 6-1, page 6-30.

In any block in which a fault type identifier does not appear in a fault declaration, it may be declared as any other type of quantity.

A fault equate construct assigns the identifier on the right of the assignment operator to the fault type on the left. The identifier may then be used in a fault statement, and the fault name (ZERO, FLAG, etc.) may be used as any other identifier.

SECTION 8
PROCEDURE DECLARATIONS

GENERAL.

SYNTAX.

The syntax for ⟨procedure declaration⟩ is as follows:

⟨procedure declaration⟩ ::= ⟨procedure type⟩ PROCEDURE ⟨procedure heading⟩ ⟨procedure body⟩

⟨procedure heading⟩ ::= ⟨identifier⟩ ⟨formal parameter part⟩

⟨formal parameter part⟩ ::= ⟨empty⟩ | (⟨formal parameter list⟩); ⟨value part⟩ ⟨specification part⟩

⟨formal parameter list⟩ ::= ⟨formal parameter⟩ | ⟨formal parameter list⟩ ⟨parameter delimiter⟩ ⟨formal parameter⟩

⟨value part⟩ ::= ⟨empty⟩ | VALUE ⟨identifier list⟩;

⟨formal parameter⟩ ::= ⟨identifier⟩

⟨identifier list⟩ ::= ⟨identifier⟩ | ⟨identifier list⟩ , ⟨identifier⟩

⟨specification part⟩ ::= ⟨specification⟩; | ⟨specification part⟩ ; ⟨specification⟩

⟨specification⟩ ::= ⟨specifier⟩ ⟨identifier list⟩ | ⟨array specification⟩

⟨specifier⟩ ::= LABEL | SWITCH | ⟨type⟩ | FILE | LIST | FORMAT | SWITCH FORMAT | SWITCH FILE | SWITCH LIST | POINTER

⟨procedure type⟩ ::= ⟨empty⟩ | ⟨type⟩

⟨array specification⟩ ::= ⟨array type⟩ ARRAY ⟨array specifier list⟩

⟨array type⟩ ::= ⟨empty⟩ | ⟨type⟩

⟨array specifier list⟩ ::= ⟨array specifier⟩ | ⟨array specifier list⟩ , ⟨array specifier⟩

⟨array specifier⟩ ::= ⟨array identifier list⟩ [⟨lower bound list⟩]

⟨array identifier list⟩ ::= ⟨identifier list⟩

⟨lower bound list⟩ ::= ⟨specified lower bound⟩ | ⟨lower bound list⟩ , ⟨specified lower bound⟩

⟨specified lower bound⟩ ::= * | ⟨integer⟩

⟨procedure body⟩ ::= ⟨unlabeled conditional statement⟩ | ⟨unlabeled unconditional statement⟩

Example:

```
PROCEDURE ROOT (A, B, C, N, X1, X2, X3);
  VALUE N;
  INTEGER N; ARRAY A, B, C, X1, X2[1]; ALPHA ARRAY X3[1];
  BEGIN
    INTEGER I; REAL DISC; LABEL START;
    START:  FOR I ← 1 STEP 1 UNTIL N DO
      BEGIN DISC ← B[I] * 2 - 4 x A[I] x C[I];
        IF DISC < 0 THEN X3 [I] ← "IMAG" ELSE
          BEGIN X1[I] ← (-B[I] + SQRT (DISC))/(2 x A[I]);
                X2[I] ← (-B[I] - SQRT (DISC))/(2 x A[I]);
                X3[I] ← "REAL"
          END
        END
    END ROOT
```

SEMANTICS.

A PROCEDURE declaration defines the procedure identifier as the name of a procedure. Whenever the identifier followed by the

appropriate parameters appears in the program, it produces a call upon the procedure (see page 6-3, procedure statement).

A procedure declared with a non-empty procedure type cannot be called as a procedure statement, but may be used only as a function designator.

Every formal parameter must appear in the specification part.

The value part specifies which formal parameters are to be called by value. When a formal parameter is called by value, the formal parameter is set to the value of the corresponding actual parameter; thereafter, the formal parameter is handled as a variable that is local to the procedure body. That is, any change of value of the variable will not ramify outside the procedure body.

Only expressions may be given as actual parameters to be called by value. These expressions are evaluated once, left-to-right, in the order in which they occur in the actual parameter list.

Formal parameters not in the value part are called by name. This means that wherever an actual parameter, called by name, appears in the procedure body, the actual parameter is replaced by the formal parameter.

A specified lower bound form of integer denotes that the corresponding dimension of the actual parameter has a declared lower bound equal to this value.

A specified lower bound form of * indicates that the corresponding dimension of the actual parameter has a declared lower bound that may vary in value.

Procedures may be called recursively.

Procedures which start with a type declarator cannot be called by procedure statements, but must be used as function designators.

A PROCEDURE declaration is composed of two parts: the procedure heading and procedure body.

PROCEDURE HEADING.

The procedure heading contains the identifier for the procedure, the list of formal parameters, and information pertaining to the formal parameters.

Whenever the procedure is activated, formal parameters in the procedure body will be assigned the value of, or be replaced by, actual parameters. The formal parameter part contains a listing of all formal parameters used in the procedure body.

The VALUE part specifies which formal parameters are to be called by value. Formal parameters called by value are called in the order in which they appear in the formal parameter list. Formal parameters not in the VALUE part are called by name. The value part of a procedure heading should contain only the identifiers of formal parameters which are specified as simple variables. If identifiers of arrays are included, they are ignored.

The specification part indicates certain characteristics of the formal parameters, that is, the kinds of identifiers they represent. Every formal parameter must appear in the specification part.

In the case of formal parameters used as array identifiers, information about the lower bounds must be given. A lower bound specified by an integer indicates that any corresponding actual parameter has a declared lower bound equal to this value. A specified lower bound of * indicates that the declared lower bound of the corresponding actual parameter may vary in value from one call on the procedure to the next. When a specifier of the form

```
ARRAY A, B, C, ..., X, Y, Z [*];
```

is used in a procedure heading, it is assumed that the lower bound for each actual parameter will be the same, and its value will be

determined by the value found for the lower bound of the actual array row corresponding to Z.

PROCEDURE BODY.
The procedure body is a statement that is to be executed when the procedure is called. This statement may be any of those listed in the syntax of statements (see section 6, statements), and therefore may be a procedure statement calling upon itself. Procedures may thus be called recursively.

SCOPE OF IDENTIFIERS OTHER THAN FORMAL PARAMETERS.
Identifiers in the procedure body which are not formal parameters are either local or global to the body, depending on whether they are declared within the body or outside the body. Those which are global to the body may be local to the block which contains the PROCEDURE declaration in its head.

Any quantity that is non-local to a procedure is inaccessible to that procedure if that quantity is local to some other procedure and is not declared to be OWN.

SPECIAL RULES OF TYPED PROCEDURES.
Certain procedures are called by means of function designators. In such cases, the PROCEDURE declaration must start with a type declarator.

The procedure body of a typed declaration must contain, and cause to be executed, an assignment statement with the procedure identifier in the left part list.

RESTRICTIONS.
A procedure body itself must not be labeled. A GO TO statement appearing in a typed procedure may not lead outside that procedure. Furthermore, in using a procedure statement within a typed procedure, any procedure called for execution in this manner must not contain a GO TO statement leading outside the typed procedure. If any statement in a procedure body is labeled, the declaration of that label must appear in the appropriate block head within the procedure body.

APPENDIX A
RESERVED WORDS

Some reserved words in Compatible ALGOL may be used as identifiers in certain constructs. Hence, the following list of reserved words is divided into three types as follows:

Type 1 - reserved throughout Compatible ALGOL.

Type 2 - standard function designators. These may be used for any purpose for which they have been declared; if not declared, they will be interpreted as function designators of the standard functions.

Type 3 - may be used as identifiers, except in those constructs where they appear in the syntax.

Type 1

ALPHA	EQV	LIST	SAVE
AND	FALSE	LOCK	SPACE
ARRAY	FILE	MOD	STEP
BEGIN	FILL	MONITOR	STREAM
BOOLEAN	FOR	NOT	SWITCH
CLOSE	FORMAT	OR	THEN
COMMENT	FORWARD	OUT	TO
DEFINE	GO	OWN	TRUE
DIV	IF	PROCEDURE	UNTIL
DO	IMP	READ	VALUE
DOUBLE	IN	REAL	WHILE
DUMP	INTEGER	RELEASE	WITH
ELSE	LABEL	REWIND	WRITE
END			

Type 2

ABS	ENTIER	MIN	STOP
ARCTAN	EXP	SIGN	TIME
CASE	LN	SIN	
COS	MAX	SQRT	

Type 3

BREAK	INTOVR	PURGE	TIMES
DBL	LB	RANDOM	UPDATE
DISK	LEQ	RB	WAIT
EQL	LSS	REMOTE	WHEN
EXPOVR	MERGE	REVERSE	ZERO
FLAG	NEQ	SEARCH	ZIP
GEQ	NO	SEEK	
GTR	PAGE	SERIAL	
INDEX	PUNCH	SORT	

APPENDIX B

INTERNAL CHARACTER CODES
(In Order of Collating Sequence)

Character	6-bit Code	Character	6-bit Code
blank	11 0000	H	01 1000
.	01 1010	I	01 1001
[	01 1011	x	10 0000
(	01 1101	J	10 0001
<	01 1110	K	10 0010
←	01 1111	L	10 0011
&	01 1100	M	10 0100
$	10 1010	N	10 0101
*	10 1011	O	10 0110
)	10 1101	P	10 0111
;	10 1110	Q	10 1000
≤	10 1111	R	10 1001
-	10 1100	≠	11 1100
/	11 0001	S	11 0010
,	11 1010	T	11 0011
%	11 1011	U	11 0100
=	11 1101	V	11 0101
]	11 1110	W	11 0110
"	11 1111	X	11 0111
#	00 1010	Y	11 1000
@	00 1011	Z	11 1001
:	00 1101	0	00 0000
>	00 1110	1	00 0001
≥	00 1111	2	00 0010
+	01 0000	3	00 0011
A	01 0001	4	00 0100
B	01 0010	5	00 0100
C	01 0011	6	00 0110
D	01 0100	7	00 0111
E	01 0101	8	00 1000
F	01 0110	9	00 1001
G	01 0111	?	00 1100

APPENDIX C

COMPILER ERROR MESSAGES

ERROR NUMBER	ROUTINE	ERROR MESSAGE
000	BLOCK	DECLARATION NOT FOLLOWED BY SEMICOLON.
001	BLOCK	IDENTIFIER DECLARED TWICE IN SAME BLOCK.
002	PROCEDUREDEC	SPECIFICATION PART CONTAINS IDENTIFIER NOT APPEARING IN FORMAL PARAMETER PART.
003	BLOCK	NON-IDENTIFIER APPEARS IN IDENTIFIER LIST OF DECLARATION.
005	PROCEDUREDEC	PROCEDURE DECLARATION PRECEDED BY ILLEGAL DECLARATOR.
006	PROCEDUREDEC	PROCEDURE IDENTIFIER USED BEFORE IN SAME BLOCK (NOT FORWARD).
007	PROCEDUREDEC	PROCEDURE IDENTIFIER NOT FOLLOWED BY (OR SEMICOLON IN PROCEDURE DECLARATION.
008	PROCEDUREDEC	FORMAL PARAMETER LIST NOT FOLLOWED BY).
009	PROCEDUREDEC	FORMAL PARAMETER PART NOT FOLLOWED BY SEMICOLON.
010	PROCEDUREDEC	VALUE PART CONTAINS IDENTIFIER WHICH DID NOT APPEAR IN FORMAL PARAPART.
011	PROCEDUREDEC	VALUE PART NOT ENDED BY SEMICOLON.
012	PROCEDUREDEC	MISSING OR ILLEGAL SPECIFICATION PART.
013	PROCEDUREDEC	OWN USED IN ARRAY SPECIFICATION.
014	PROCEDUREDEC	SAVE USED IN ARRAY SPECIFICATION.
015	ARRAYDEC	ARRAY CALL-BY-VALUE NOT IMPLEMENTED.
016	ARRAYDEC	ARRAY ID IN DECLARATION NOT FOLLOWED BY [.
017	ARRAYDEC	LOWER BOUND IN ARRAY DEC NOT FOLLOWED BY :.
018	ARRAYDEC	BOUND PAIR LIST NOT FOLLOWED BY].
019	ARRAYSPEC	ILLEGAL LOWER BOUND DESIGNATOR IN ARRAY SPECIFICATION.
020	BLOCK	OWN APPEARS IMMEDIATELY BEFORE IDENTIFIER (NO TYPE).
021	BLOCK	SAVE APPEARS IMMEDIATELY BEFORE IDENTIFIER (NO TYPE).

ERROR NUMBER	ROUTINE	ERROR MESSAGE
023	BLOCK	DECLARATOR PRECEDED ILLEGALLY BY ANOTHER DECLARATOR.
024	PROCEDUREDEC	LABEL CANNOT BE PASSED TO FUNCTION.
025	BLOCK	DECLARATOR OR SPECIFIER ILLEGALLY PRECEDED BY OWN OR SAVE OR SOME OTHER DECLARATOR.
026	FILEDEC	MISSING (IN FILE DEC.
027	FILEDEC	MISSING RECORD SIZE.
028	FILEDEC	ILLEGAL BUFFER PART OR SAVE FACTOR IN FILE DEC.
029	FILEDEC	MISSING) IN FILE DEC.
030	IODEC	MISSING COLON IN DISK DESCRIPTION.
031	LISTDEC	MISSING (IN LISTDEC.
032	FORMATDEC	MISSING (IN FORMAT DEC.
033	SWITCHDEC	SWITCH DEC DOES NOT HAVE ← OR FORWARD AFTER IDENTIFIER.
034	SWITCHFILEDEC	MISSING ← AFTER FILED.
035	SWITCHFILEDEC	NON FILE ID APPEARING IN DECLARATION OF SWITCHFILE.
036	SUPERFORMATDEC	FORMAT ID NOT FOLLOWED BY ←.
037	SUPERFORMATDEC	MISSING (AT START OF FORMATPHRASE.
038	SUPERFORMATDEC	FORMAT SEGMENT >1023 WORDS.
039	BLOCK	NUMBER OF NESTED BLOCKS IS GREATER THAN 31.
040	IODEC	PROGRAM PARAMETER BLOCK SIZE EXCEEDED.
041	HANDLESWLIST	MISSING ← AFTER SWITCH LIST ID.
042	HANDLESWLIST	ILLEGAL LIST ID APPEARING IN SWITCH LIST
043	IODEC	MISSING] AFTER DISK IN FILEDEC.
044	IODEC	MISSING [AFTER DISK IN FILEDEC.
045	DEFINEDEC	MISSING "=" AFTER DEFINE ID.
046	ARRAE	NON-LITERAL ARRAY BOUND NOT GLOBAL TO ARRAY DECL.
047	TABLE	ITEM FOLLOWING @ NOT A NUMBER.
048	PROCEDUREDEC	NUMBER OF PARAMETERS DIFFERS FROM FWD DECL.
049	PROCEDUREDEC	CLASS OF PARAMETER DIFFERS FROM FWD DECL.

ERROR NUMBER	ROUTINE	ERROR MESSAGE
050	PROCEDUREDEC	VALUE PART DIFFERS FROM FWD DECL.
059	ARRAYDEC	MISSING ← IN FAULT STATEMENT.
061	FAULTDEC	INVALID FAULT TYPE: MUST BE FLAG, EXPOVR, ZERO, INTOVR, OR INDEX.
062	SCANSTMT OR REPLACESTMT	LEVEL OF POINTER EXPRESSION EXCEEDS LEVEL OF UPDATE POINTER IDENTIFIER.
063	SCANSTMT OR REPLACESTMT	UPDATE POINTER MAY NOT BE CALL-BY-NAME FORMAL PARAMETER.
070	CASESTMT	MISSING "BEGIN".
071	CASESTMT	MISSING END.
072	SCANSTMT OR REPLACESTMT	POINTER IDENTIFIER REQUIRED.
073	SCANSTMT OR REPLACESTMT	SIMPLE ARITHMETIC VARIABLE REQ.
074	SCANSTMT OR REPLACESTMT	RELATIONAL OP OR IN EXPECTED.
075	SCANSTMT OR REPLACESTMT	CONDITION MUST START WITH WHILE OR UNTIL.
076	REPLACESTMT	BY MISSING AFTER DESTINATION POINTER.
077	REPLACESTMT	SOURCE MUST BE POINTER OR ARITHMETIC EXP.
078	SCANSTMT OR REPLACESTMT	ALPHA REQUIRED AFTER IN.
079	PRIMARY	ILLEGAL EXPRESSION TYPE.
080	PRIMARY	MISSING COMMA.
090	PARSE	MISSING LEFT BRACKET.
091	PARSE	MISSING COLON.
092	PARSE	ILLEGAL BIT NUMBER.
093	PARSE	FIELD SIZE MUST BE LITERAL.
094	PARSE	MISSING RIGHT BRACKET.
095	PARSE	ILLEGAL FIELD SIZE.
100	ANYWHERE	UNDECLARED IDENTIFIER.
101	CHECKER	AN ATTEMPT HAS BEEN MADE TO ADDRESS AN IDENTIFIER WHICH IS LOCAL TO ONE PROCEDURE AND GLOBAL TO ANOTHER. IF THE QUANTITY IS A PROCEDURE NAME OR AN OWN VARIABLE, THIS RESTRICTION IS RELAXED.

ERROR NUMBER	ROUTINE	ERROR MESSAGE
102	AEXP	CONDITIONAL EXPRESSION IS NOT OF ARITHMETIC TYPEH.
103	PRIMARY	PRIMARY MAY NOT START WITH A QUANTITY OF THIS TYPE.
104	ANYWHERE	MISSING RIGHT PARENTHESIS.
105	ANYWHERE	MISSING LEFT PARENTHESIS.
106	PRIMARY	PRIMARY MAY NOT START WITH DECLARATOR.
107	BEXP	THE EXPRESSION IS NOT OF BOOLEAN TYPE.
108	EXPRSS	A RELATION MAY NOT HAVE CONDITIONAL EXPRESSIONS AS THE ARITHMETIC EXPRESSIONS.
109	BOOSEC, SIMPBOO, AND BOOCOMP	THE PRIMARY IS NOT BOOLEAN.
110	BOOCOMP	A NON-BOOLEAN OPERATOR OCCURS IN A BOOLEAN EXPRESSION.
111	BOOPRIM	NO EXPRESSION (ARITHMETIC, BOOLEAN, OR DESIGNATIONAL) MAY START WITH A QUANTITY OF THIS TYPE.
112	BOOPRIM	NO EXPRESSION (ARITHMETIC, BOOLEAN, OR DESIGNATIONAL) MAY START WITH A DECLARATOR.
113		EITHER THE SYNTAX OR THE RANGE OF THE LITERALS FOR A CONCATENATE OPERATOR IS INCORRECT.
114	DOTSYNTAX	EITHER THE SYNTAX OR THE RANGE OF THE LITERALS FOR A PARTIAL WORD DESIGNATOR IS INCORRECT.
115	DEXP	THE EXPRESSION IS NOT OF DESIGNATIONAL TYPE.
116	IFCLAUSE	MISSING THEN.
117	BANA	MISSING LEFT BRACKET.
118	BANA	MISSING RIGHT BRACKET.
119	COMPOUNDTAIL	MISSING SEMICOLON OR END.
120	COMPOUNDTAIL	MISSING END.
121	ACTUALPARAPART	INDEXED FILES MAY NOT BE PASSED.
123	ACTUALPARAPART	THE ACTUAL AND FORMAL PARAMETERS DO NOT AGREE AS TO TYPE.
124	ACTUALPARAPART	ACTUAL AND FORMAL ARRAYS DO NOT HAVE SAME NUMBER OF DIMENSIONS.

ERROR NUMBER	ROUTINE	ERROR MESSAGE
126	ACTUALPARAPART	NO ACTUAL PARAMETER MAY START WITH A QUANTITY OF THIS TYPE.
128	ACTUALPARAPART	EITHER ACTUAL AND FORMAL PARAMETERS DO NOT AGREE AS TO NUMBER, OR EXTRA RIGHT PARENTHESIS.
129	ACTUALPARAPART	ILLEGAL PARAMETER DELIMITER.
130	RELSESTMT	NO FILE NAME.
131	DOSTMT	MISSING UNTIL.
132	WHILESTMT	MISSING DO.
133	LABELR	MISSING COLON.
134	LABELR	THE LABEL WAS NOT DECLARED IN THIS BLOCK.
135	LABELR	THE LABEL HAS ALREADY OCCURRED.
136	FORMATPHRASE	IMPROPER FORMAT EDITING PHRASE.
137	FORMATPHRASE	A FORMAT EDITING PHRASE DOES NOT HAVE AN INTEGER WHERE AN INTEGER IS REQUIRED.
138	FORMATPHRASE	THE WIDTH IS TOO SMALL IN E OR F EDITING PHRASE.
139	TABLE	DEFINE IS NESTED MORE THAN EIGHT DEEP.
140	NEXTENT	AN INTEGER IN A FORMAT IS GREATER THAN 1023.
141	SCANNER	INTEGER OR IDENTIFIER HAS MORE THAN 63 CHARACTERS.
142	DEFINEGEN	A DEFINE CONTAINS MORE THAN 2047 CHARACTERS (BLANK SUPPRESSED).
143	COMPOUNDTAIL	EXTRA END.
144	STMT	NO STATEMENT MAY START WITH THIS TYPE IDENTIFIER.
145	STMT	NO STATEMENT MAY START WITH THIS TYPE QUANTITY.
146	STMT	NO STATEMENT MAY START WITH A DECLARATOR - MAY BE A MISSING END OF A PROCEDURE OR A MISPLACED DECLARATION.
147	SWITCHGEN	MORE THAN 256 EXPRESSIONS IN A SWITCH DECLARATION.
148	GETSPACE	MORE THAN 1023 PROGRAM REFERENCE TABLE CELLS ARE REQUIRED FOR THIS PROGRAM.
149	GETSPACE	MORE THAN 255 STACK CELLS ARE REQUIRED FOR THIS PROCEDURE.

ERROR NUMBER	ROUTINE	ERROR MESSAGE
150	THRUSTMT	MISSING DO IN THRU CLAUSE.
151	FORSTMT	INDEX VARIABLE MAY NOT BE BOOLEAN.
152	FORSTMT	MISSING LEFT ARROW FOLLOWING INDEX VARIABLE.
153	FORSTMT	MISSING UNTIL OR WHILE IN STEP ELEMENT.
154	FORSTMT	MISSING DO IN FOR CLAUSE.
155	IFEXP	MISSING ELSE.
156	LISTELEMENT	A DESIGNATIONAL EXPRESSION MAY NOT BE A LIST ELEMENT.
157	LISTELEMENT	A ROW DESIGNATOR MAY NOT BE A LIST ELEMENT.
158	LISTELEMENT	MISSING RIGHT BRACKET IN GROUP ELEMENTS.
159	PROCSTMT	ILLEGAL USE OF PROCEDURE OR FUNCTION IDENTIFIER.
160	PURGE	DECLARED LABEL DOES NOT OCCUR.
161	PURGE	DECLARED FORWARD PROCEDURE DOES NOT OCCUR.
162	PURGE	DECLARED SWITCH FORWARD DOES NOT OCCUR.
163	FORMATPHRASE	THE WIDTH OF A FIELD IS MORE THAN 63.
164	UNKNOWNSTMT	MISSING COMMA IN ZIP OR WAIT STATEMENT.
165	IMPFUN	MISSING COMMA IN DELAY PARAMETER LIST.
166	PEXP	THE EXPRESSION IS NOT OF POINTER TYPE.
167	PTRPRIMARY	POINTER PRIMARY MAY NOT START WITH A QUANTITY OF THIS TYPE.
168	VARIABLE	POINTER MAY NOT HAVE PARTIAL WORD SYNTAX.
169	ARRAE	POINTER ARRAYS NOT PERMITTED.
170	SWAPSTMT	MISSING COMMA.
171	SWAPSTMT	PARAMETERS MUST BE 2-DIMENSIONAL ARRAYS.
200	EMIT	SEGMENT TOO LARGE (> 4093 SYLLABLES).
201	SIMPLE VARIABLE	PARTIAL WORD DESIGNATOR NOT LEFT-MOST IN A LEFT PART LIST.
202	SIMPLE VARIABLE	MISSING, OR ←.
203	SUBSCRIPTED VARIABLE	WRONG NUMBER OF SUBSCRIPTS IN A ROW DESIGNATOR.
204	SUBSCRIPTED VARIABLE	MISSING] IN A ROW DESIGNATOR.

ERROR NUMBER	ROUTINE	ERROR MESSAGE
205	SUBSCRIPTED VARIABLE	A ROW DESIGNATOR APPEARS OUTSIDE OF AN ACTUAL PARAMETER LIST OR FILL STATEMENT.
206	SUBSCRIPTED VARIABLE	MISSING].
207	SUBSCRIPTED VARIABLE	MISSING [.
208	SUBSRCIPTED VARIABLE	WRONG NUMBER OF SUBSCRIPTS.
209	SUBSCRIPTED VARIABLE	PARTIAL WORD DESIGNATOR NOT LEFT-MOST IN A LEFT PART LIST.
210	SUBSCRIPTED VARIABLE	MISSING , OR ←.
211	VARIABLE	PROCEDURE ID USED OUTSIDE OF SCOPE IN LEFT PART.
212	VARIABLE	SUB-ARRAY DESIGNATOR PERMITTED AS ACTUAL PARAMETER ONLY.
213	MAKEPOINTER	POINTER REQUIRES ARRAY ROW, SUBSCRIPTED VARIABLE, OR ONE-DIMENSIONAL ARRAY ID.
214	STRINGRELATION	POINTER RELATION MUST BE = OR ≠ ONLY.
215	MAKEPOINTER	CHARACTER SIZE MUST BE LITERAL 6 or 8.
216	VARIABLE	LEVEL OR POINTER EXPRESSION EXCEEDS LEVEL OR LEFT-PART POINTER IDENTIFIER.
217	VARIABLE	LEFT-PART POINTER MAY NOT BE CALL-BY-NAME FORMAL PARAMETER.
218	STRINGRELATION	POINTER UPDATE NOT PERMITTED WITH POINTER RELATION.
219	BOOPRIM	RELATIONAL OPERATOR EXPECTED WHEN POINTER UPDATE CONSTRUCT USED.
268	EMITC	A REPEAT INDEX ≥ 64 WAS SPECIFIED OR TOO MANY FORMAL PARAMETERS, LOCALS, AND LABELS.
269	TABLE	A CONSTANT IS SPECIFIED WHICH IS TOO LARGE OR TOO SMALL.
281	DBLSTMT	MISSING (.
282	DBLSTMT	TOO MANY OPERATORS.
283	DBLSTMT	TOO MANY OPERANDS.
284	DBLSTMT	MISSING ,.
285	DBLSTMT	MISSING).

ERROR NUMBER	ROUTINE	ERROR MESSAGE
300	FILLSTMT	THE IDENTIFIER FOLLOWING THE WORD FILL IS NOT AN ARRAY IDENTIFIER.
301	FILLSTMT	MISSING WITH IN FILL STATEMENT.
302	FILLSTMT	IMPROPER FILL ELEMENT.
303	FILLSTMT	NON OCTAL CHARACTER IN OCTAL FILL. THE THREE LOW ORDER BITS ARE CONVERTED AND COMPILATION CONTINUES.
304	FILLSTMT	IMPROPER ROW DESIGNATOR.
305	FILLSTMT	NUMBER OF DATA WORDS EXCEEDS 1023.
350	CHECKCOMMA	MISSING OR ILLEGAL PARAMETER DELIMITER IN SORT OR MERGE STATEMENT.
351	OUTPROCHECK	ILLEGAL TYPE FOR SORT OR MERGE OUTPUT PROC.
352	OUTPROCHECK	OUTPUT PROCEDURE IN SORT OR MERGE STMT DOES NOT HAVE EXACTLY TWO PARAMETERS.
353	OUTPROCHECK	FIRST PARAMETER OF OUTPUT PROCEDURE MUST BE BOOLEAN.
354	OUTPROCHECK	SECOND PARAM OF OUTPUT PROCEDURE MUST BE ONE-DIM ARRAY.
355	SORTSTMT	MISSING (.
356	HVCHECK	ILLEGAL TYPE FOR SORT OR MERGE HIGHVALUE PRO.
357	HVCHECK	HIVALUE PROCEDURE DOES NOT HAVE EXACTLY ONE PARAMETER.
358	HVCHECK	HIVALUE PROCEDURE PARAM NOT ONE-DIM ARRAY.
359	EQLESCHECK	SORT OR MERGE COMPARE PROCEDURE NOT BOOLEAN.
360	EQLESCHECK	COMPARE PROCEDURE DOES NOT HAVE EXACTLY TWO PARAMETERS.
361	EQLESCHECK	COMPARE PROCEDURE FIRST PARAM NOT 1-D ARRAY.
362	EQLESCHECK	COMPARE PROCEDURE SECOND PARAM NOT 1-D ARRAY.
363	INPROCHECK	SORT STMT INPUT PROCEDURE NOT BOOLEAN.
364	INPROCHECK	INPUT PROCEDURE DOES NOT HAVE EXACTLY ONE PARAMETER.
365	INPROCHECK	INPUT PROCEDURE PARAMETER NOT ONE-D ARRAY.
366	SORTSTMT	MISSING).

ERROR NUMBER	ROUTINE	ERROR MESSAGE
367	MERGESTMT	MISSING (.
368	MERGESTMT	MORE THAN 7 or LESS THAN 2 FILES TO MERGE.
369	MERGESTMT	MISSING).
400	MERRIMAC	MISSING FILE ID IN MONITOR DEC.
401	MERRIMAC	MISSING LEFT PARENTHESIS IN MONITOR DEC.
402	MERRIMAC	IMPROPER SUBSCRIPT FOR MONITOR LIST ELEMENT.
403	MERRIMAC	IMPROPER SUBSCRIPT EXPRESSION DELIMITER IN MONITOR LIST ELEMENT.
404	MERRIMAC	IMPROPER NUMBER OF SUBSCRIPTS IN MONITOR LIST ELEMENT.
405	MERRIMAC	LABEL OR SWITCH MONITORED AT IMPROPER LEVEL.
406	MERRIMAC	IMPROPER MONITOR LIST ELEMENT.
407	MERRIMAC	MISSING RIGHT PARENTHESIS IN MONITOR DECLARATION.
408	MERRIMAC	IMPROPER MONITOR DECLARATION DELIMITER.
409	DMUP	MISSING FILE IDENTIFIER IN DUMP DECLARATION.
410	DMUP	MISSING LEFT PARENTHESIS IN DUMP DECLARATION.
411	DMUP	SUBSCRIPTED VARIABLE IN DUMP LIST HAS WRONG NUMBER OF SUBSCRIPTS.
412	DMUP	SUBSCRIPTED VARIABLE IN DUMP LIST HAS WRONG NUMBER OF SUBSCRIPTS.
413	DMUP	IMPROPER ARRAY DUMP LIST ELEMENT.
414	DMUP	ILLEGAL DUMP LIST ELEMENT.
415	DMUP	MORE THAN 100 LABELS APPEAR AS DUMP LIST ELEMENTS IN ONE DUMP DECLARATION.
416	DMUP	ILLEGAL DUMP LIST ELEMENT DELIMITER.
417	DMUP	MISSING OR NON-LOCAL LABEL IN DUMP DECLARATION.
418	DMUP	MISSING COLON IN DUMP DECLARATION.
419	DMUP	IMPROPER DUMP DECLARATION DELIMITER.
420	READSTMT	MISSING LEFT PARENTHESIS IN READ STATEMENT.
421	READSTMT	MISSING LEFT PARENTHESIS IN READ REVERSE STATEMENT.

ERROR NUMBER	ROUTINE	ERROR MESSAGE
422	READSTMT	MISSING FILE IN READ STATEMENT.
424	READSTMT	IMPROPER FILE DELIMITER IN READ STATEMENT.
425	READSTMT	IMPROPER FORMAT DELIMITER IN READ STATEMENT.
426	READSTMT	IMPROPER DELIMITER FOR SECOND PARAMETER IN READ STATEMENT.
427	READSTMT	IMPROPER ROW DESIGNATOR IN READ STATEMENT.
428	READSTMT	IMPROPER ROW DESIGNATOR DELIMITER IN READ STATEMENT.
429	READSTMT	MISSING ROW DESIGNATOR IN READ STATEMENT.
430	READSTMT	IMPROPER DELIMITER PRECEDING THE LIST IN A READ STATEMENT.
431	FCRSCAN	IMPROPER SYNTAX.
433	HANDLETHETAILEND OFAREADORSPACESTA TEMENT	MISSING RIGHT BRACKET IN READ OR SPACE STATEMENT.
434	SPACESTMT	MISSING LEFT PARENTHESIS IN SPACE STATEMENT.
435	SPACESTMT	IMPROPER FILE IDENTIFITER IN SPACE STATEMENT.
436	SPACESTMT	MISSING COMMA IN SPACE STATEMENT.
437	SPACESTMT	MISSING RIGHT PARENTHESIS IN SPACE STATEMENT.
438	WRITESTMT	MISSING LEFT PARENTHESIS IN A WRITE STATEMENT.
439	WRITESTMT	IMPROPER FILE IDENTIFIER IN A WRITE STATEMENT.
440	WRITESTMT	IMPROPER DELIMITER FOR FIRST PARAMETER IN A WRITE STATEMENT.
441	WRITESTMT	MISSING RIGHT BRACKET IN CARRIAGE CONTROL PART OF A WRITE STATEMENT.
442	WRITESTMT	ILLEGAL CARRIAGE CONTROL DELIMITER IN A WRITE STATEMENT.
443	WRITESTMT	IMPROPER SECOND PARAMETER DELIMITER IN WRITE STATEMENT.
444	WRITESTMT	IMPROPER ROW DESIGNATOR IN A WRITE STATEMENT.

ERROR NUMBER	ROUTINE	ERROR MESSAGE
445	WRITESTMT	MISSING RIGHT PARENTHESIS AFTER A ROW DESIGNATOR IN A WRITE STATEMENT.
446	WRITESTMT	IMPROPER DELIMITER PRECEDING A LIST IN A WRITE STATEMENT.
448	WRITESTMT	IMPROPER LIST DELIMITER IN A WRITE STATEMENT.
449	READSTMT	IMPROPER LIST DELIMITER IN A READ STATEMENT.
450	LOCKSTMT	MISSING LEFT PARENTHESIS IN A LOCK STATEMENT.
451	LOCKSTMT	IMPROPER FILE PART IN A LOCK STATEMENT.
452	LOCKSTMT	MISSING COMMA IN A LOCK STATEMENT.
453	LOCKSTMT	IMPROPER UNIT DISPOSITION PART IN A LOCK STATEMENT.
454	LOCKSTMT	MISSING RIGHT PARENTHESIS IN A CLOSE STATEMENT.
455	CLOSESTMT	MISSING LEFT PARENTHESIS IN A CLOSE STATEMENT.
456	CLOSESTMT	IMPROPER FILE PART IN A CLOSE STATEMENT.
457	CLOSESTMT	MISSING COMMA IN A CLOSE STATEMENT.
458	CLOSESTMT	IMPROPER UNIT DISPOSITION PART IN A CLOSE STATEMENT.
459	CLOSESTMT	MISSING RIGHT PARENTHESIS IN A CLOSE STATEMENT.
460	RWNDSTMT	MISSING LEFT PARENTHESIS IN A REWIND STATEMENT.
461	RWNDSTMT	IMPROPER FILE PART IN A REWIND STATEMENT.
462	RWNDSTMT	MISSING RIGHT PARENTHESIS IN A REWIND STATEMENT.
463	BLOCK	A MONITOR DECLARATION APPEARS IN THE SPECIFICATION PART OF A PROCEDURE.
464	BLOCK	A DUMP DECLARATION APPEARS IN THE SPECIFICATION PART OF A PROCEDURE.
465	DMUP	DUMP INDICATOR MUST BE UNSIGNED INTEGER OR SIMPLE VARIABLE.
500	SEARCHLIB	ILLEGAL LIBRARY IDENTIFIER.
501	SEARCHLIB	LIBRARY IDENTIFIER NOT CONTAINED IN DIRECTORY.
502	SEARCHLIB	ILLEGAL LIBRARY START POINT.

ERROR NUMBER	ROUTINE	ERROR MESSAGE
503	SEARCHLIB	SEPARATOR REQUIRED BETWEEN START POINT AND LENGTH.
504	SEARCHLIB	ILLEGAL LIBRARY LENGTH.
505	SEARCHLIB	MISSING BRACKET.
507	SEARCHLIB	TAPE POSITIONING ERROR.
508	ANYWHERE	CONSTRUCT NOT ALLOWED IN TIME SHARING SYSTEM.
509	IODEC	NON-LITERAL FILE VALUE NOT GLOBAL TO FILE DECL.

INDEX

METALINGUISTIC VARIABLES

The syntactical definition of each Compatible ALGOL metalinguistic variable will be found on the pages shown below.

cut along ed line

BURROUGHS CORPORATION
DATA PROCESSING PUBLICATIONS
REMARKS FORM

TITLE: _____ FORM: _____

_____ DATE: _____

_____

CHECK TYPE OF SUGGESTION:

☐ ADDITION ☐ DELETION ☐ REVISION ☐ ERROR

GENERAL COMMENTS AND/OR SUGGESTIONS FOR IMPROVEMENT OF PUBLICATION:

FROM: NAME _____ DATE _____

TITLE _____

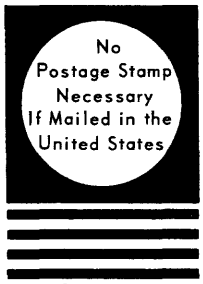
COMPANY _____

ADDRESS _____

_____

STAPLE

FOLD DOWN SECOND FOLD DOWN



BUSINESS REPLY MAIL
First Class Permit No. 817, Detroit, Mich. 48232

Burroughs Corporation
6071 Second Avenue
Detroit, Michigan 48232

attn: Sales Technical Services
Systems Documentation

FOLD UP FIRST FOLD UP


Wherever There's
Business There's
Burroughs