Burroughs

B 5500

Information
Processing Systems

EXTENDED ALGOL

REFERENCE MANUAL

Burroughs

B 5500

INFORMATION PROCESSING SYSTEMS

EXTENDED ALGOL

REFERENCE MANUAL

Business Machines Group
Sales Technical Services
Systems Documentation

Burroughs Corporation
Detroit, Michigan 48232

Copyright © 1966, 1964, 1962
BURROUGHS CORPORATION
11596952 11739491

PREFACE

One of the programing languages utilized by the Burroughs B 5500 Electronic Information Processing System is Extended ALGOL. In addition to implementing virtually all of ALGOL 60, Extended ALGOL provides for communication between the processor and input/output devices, enables editing of data, and facilitates program debugging. Within the framework of an Extended ALGOL program, the programmer can thus exercise close control over data transmission and manipulation to any desired degree.

This manual is a detailed reference source for Extended ALGOL. It describes all the structures contained in the language through the use of syntactical descriptions, pertinent examples, and semantics. Although the material contained herein is not intended as a teaching aid, serious and careful study should provide the reader with a thorough understanding of Extended ALGOL.

Except where spaces are specifically prohibited or mandatory, as described in the following text, the use of blanks is optional. For this reason, the spacing within many examples has been deliberately varied to illustrate both optimum program readability and optimum packing of information on punched cards.

When a reserved word is actually used in a given construct, it appears in capital letters; when it is merely descriptive, however, it is in lowercase letters. For example, a "LIST declaration" contains the reserved word LIST, but a "list part" does not.

The reader is assumed to have had some experience in systems programing. For those unfamiliar with ALGOL 60, the Burroughs B 5500 Electronic Information Processing System, or both, the following publications are suggested:

1. Burroughs B 5500 Information Processing System Reference Manual (1021326)

2. Burroughs B 5500 System Operation Manual (1024916)

3. Thurnau, D. H., et al., ALGOL Programing - A Basic Approach.

4. Naur, P., et al., Revised Report on the Algorithmic Language ALGOL 60 (Communications of the Association for Computing Machinery, Vol. 6, No. 1, Jan., 1963).

5. McCracken, Daniel D., An Introduction to ALGOL Programming (New York, New York: John Wiley and Sons, 1962).

In many cases, portions of reference 4 have been reproduced in this manual with little change in order to adhere as closely as possible to the formal definition of ALGOL 60.

TABLE OF CONTENTS

NOTE

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

SECTION	TITLE	PAGE
	INTRODUCTION	xiii
1	STRUCTURE OF THE LANGUAGE	1-1
	General	1-1
	Conventions Used in the Description of the Language	1-2
	Character Set	1-4
2	BASIC COMPONENTS: BASIC SYMBOLS, IDENTIFIERS, NUMBERS, AND STRINGS	2-1
	General	2-1
	Letters	2-1
	Digits	2-1
	Logical Values	2-2
	Delimiters	2-2
	Spacing	2-3
	The Use of Comments	2-4
	Identifiers	2-5
	Numbers	2-6
	Size Limitations of Numbers	2-7
	Strings	2-7
	Use of Strings	2-8
	Letter String	2-8
	Constituents and Scopes	2-9
	Values and Types	2-9
3	GENERAL COMPONENTS	3-1
	General	3-1

TABLE OF CONTENTS (cont)

SECTION	TITLE	PAGE
3 (cont)	Variables	3-1
	Simple Variables	3-2
	Subscripted Variables	3-2
	Number of Subscripts	3-3
	Evaluation of Subscripts	3-3
	Partial Word Designators	3-3
	Values Allowed for Field	3-4
	Switch File Designator	3-4
	Switch Format Designator	3-5
	Switch List Designator	3-6
	Function Designators	3-7
	Standard Functions	3-8
	The TIME Functions	3-9
	Type Transfer Functions	3-9
	ENTIER	3-10
	REAL	3-10
	BOOLEAN	3-10
	Interrogate Function	3-10
	STATUS	3-10
4	EXPRESSIONS	4-1
	General	4-1
	Arithmetic Expressions	4-1
	Simple Arithmetic Expressions	4-3
	Primaries	4-3
	Conditional Arithmetic Expressions	4-4
	Operators and Types	4-5
	Arithmetic Operators	4-6
	Arithmetic Expression Types	4-6
	Precedence of Operators	4-7
	Numerical Limitations and Significant Digits	4-8
	Boolean Expressions	4-8
	Simple Boolean Expressions	4-10

TABLE OF CONTENTS (cont)

SECTION	TITLE	PAGE
4 (cont)	Conditional Boolean Expressions . .	4-11
	Types	4-12
	Relational and Logical Operators. . . .	4-12
	Relational Operators.	4-12
	Logical Operators	4-13
	Precedence of Operators	4-13
	Designational Expressions	4-14
	Simple Designational Expressions. .	4-15
	Conditional Designational Expressions	4-15
	The Subscript Expression of a Switch Designator	4-16
	Concatenate Expression.	4-16
5	PROGRAMS, BLOCKS, AND COMPOUND STATEMENTS . . .	5-1
	General	5-1
	Nested Blocks	5-2
	Disjoint Blocks	5-2
6	STATEMENTS.	6-1
	General	6-1
	Unconditional Statements.	6-1
	Assignment Statements	6-2
	Types	6-4
	GO TO Statements.	6-5
	Dummy Statements.	6-5
	Fill Statements	6-6
	Row Designator.	6-6
	Value List.	6-7
	Library Call Statements	6-7
	DOUBLE Statements	6-8
	Procedure Statements.	6-10
	Value Assignment (Call by Value). .	6-12
	Name Replacement (Call by Name) . .	6-12
	Stream Procedure Call Statement	6-15

TABLE OF CONTENTS (cont)

SECTION	TITLE	PAGE
6 (cont)	Stream Value Parameters	6-16
	Stream Name Parameters	6-16
	Input/Output Statements	6-18
	READ Statements	6-18
	Free-Field Data	6-21
	Logical Values	6-24
	RELEASE Statements	6-25
	SPACE Statements	6-26
	WRITE Statements	6-27
	REWIND Statements	6-29
	LOCK Statements	6-30
	CLOSE Statements	6-30
	Break-Out Statements	6-32
	WHEN Statement	6-32
	WAIT Statement	6-33
	Fault Statement	6-34
	ZIP Statement	6-36
	Label Equation Statement	6-37
	SORT Statement and MERGE Statement	6-40
	Edit and Move Statement	6-41
	Disk I/O Statement	6-42
	Disk READ Statement	6-42
	Disk WRITE Statement	6-44
	Disk READ SEEK Statement	6-45
	Disk SPACE Statement	6-47
	Disk REWIND Statement	6-48
	Disk CLOSE Statement	6-48
	Disk LOCK Statement	6-49
	Data Communications I/O Statement	6-49
	Status Word	6-50
	Data Communications READ Statement	6-52
	Data Communications READ LOCK Statement	6-54
	Data Communications READ SEEK Statement	6-55

TABLE OF CONTENTS (cont)

SECTION	TITLE	PAGE
6 (cont)	Data Communications WRITE Statement	6-57
	Data Communications WRITE LOCK Statement	6-59
	Interrogate Function	6-59
	Data Communications CLOSE Statement	6-61
	Data Communications REWIND Statement	6-61
	CASE Statement	6-62
	SEARCH Statement	6-62
7	CONDITIONAL STATEMENTS	7-1
	General	7-1
	IF Statement	7-2
	IF . . . ELSE Statement	7-2
	Nested IF Statements	7-2
	Entering a Conditional Statement	7-4
8	ITERATIVE STATEMENTS	8-1
	General	8-1
	FOR Statement	8-1
	The For-List	8-2
	Arithmetic Expression Element	8-3
	STEP-UNTIL Element	8-3
	WHILE Element	8-4
	STEP-WHILE Element	8-5
	Value of Controlled Variable on Exit from FOR Statement	8-5
	DO Statements	8-6
	WHILE Statements	8-6
9	DECLARATIONS	9-1
	General	9-1
	Type Declarations	9-2
	Local or OWN	9-3
	Type	9-3
	ARRAY Declarations	9-3
	SAVE Arrays	9-5

TABLE OF CONTENTS (cont)

SECTION	TITLE	PAGE
9 (cont)	Local or OWN	9-5
	Type	9-5
	Bound Pair List	9-5
	SWITCH Declarations	9-6
	Evaluation of Expressions in the Switch List	9-6
	Influence of Scope	9-7
	DEFINE Declarations	9-7
	Nesting of Definitions	9-8
	LABEL Declarations	9-9
	FILE Declarations	9-10
	SWITCH FILE Declarations	9-18
	FORMAT Declarations	9-19
	Input Editing Specifications	9-20
	Input Editing Phrases	9-20
	Output Editing Specifications	9-25
	Output Editing Phrases	9-26
	The Meaning of the Symbol	9-31
	SWITCH FORMAT Declarations	9-32
	LIST Declarations	9-33
	SWITCH LIST Declarations	9-34
	FORWARD Reference Declarations	9-35
	MONITOR Declarations	9-36
	Monitor List Elements	9-36
	DUMP Declarations	9-38
	FAULT Declarations	9-40
10	PROCEDURE DECLARATIONS	10-1
	General	10-1
	Procedure Heading	10-3
	Procedure Body	10-4
	Scope of Identifiers Other Than Formal Parameters	10-4
	Special Rules of Typed Procedures	10-4

TABLE OF CONTENTS (cont)

SECTION	TITLE	PAGE
11	STREAM PROCEDURE DECLARATIONS	11-1
	General	11-1
	Formal Parameters and Value Part	11-2
	Stream Declarations	11-2
	Compound Stream Tail	11-3
	Automatic Index Adjustment	11-4
	Stream Statements	11-5
	Unconditional Stream Statements	11-5
	Set Address Statements	11-6
	Store Address Statements	11-6
	Skip Address Statements	11-7
	Recall Address Statements	11-8
	Destination String Statements	11-8
	Transfer Words	11-10
	Transfer Characters	11-10
	Input Convert	11-10
	Output Convert	11-10
	Transfer and Add	11-11
	Transfer Character Portions	11-11
	Literal Characters	11-12
	Literal Bits	11-12
	Repetitive Indicator	11-12
	Blank Replacement	11-12
	Stream GO TO Statements	11-13
	SKIP Bit Statements	11-13
	Stream TALLY Statements	11-14
	Stream Nest Statements	11-14
	Stream RELEASE Statements	11-15
	Compound Stream Statements	11-16
	Stream Dummy Statements	11-16
	Conditional Stream Statements	11-17
	Source With Literal	11-18
	Source With Destination	11-18

TABLE OF CONTENTS (cont)

SECTION	TITLE	PAGE
11 (cont)	Source Bit	11-18
	TOGGLE	11-18
	Source For Alpha	11-18
12	SORT STATEMENT AND MERGE STATEMENT	12-1
	SORT Statement	12-1
	Program Example	12-4
	MERGE Statement	12-5
APPENDIX A -	RESERVED WORDS	A-1
APPENDIX B -	INTERNAL CHARACTER CODES	B-1
INDEX		one

LIST OF ILLUSTRATIONS

FIGURE	TITLE	PAGE
6-1	Format for Control Deck on Disk	6-38

LIST OF TABLES

TABLE	TITLE	PAGE
3-1	Results of Different TIME (AE) Parameters	3-9
4-1	Represented Values of Primaries in Arithmetic Expression	4-3
4-2	Meaning of *	4-6
4-3	Types of Values Resulting from an Arithmetic Operation	4-7
4-4	Values Represented by Primaries in a Boolean Expression	4-11
4-5	Operation of Logical Operators	4-13
6-1	Program Errors for Fault Types	6-35
6-2	Values for Output Media Digit	6-40
9-1	Characteristics of Types of Input Editing Phrases	9-21
9-2	Boolean Values for Various Field Widths in Input Editing Phrase	9-23
9-3	Characteristics of Types of Output Editing Phrases	9-27
9-4	Boolean Values for Various Field Widths in Output Editing Phrase	9-29

INTRODUCTION

Extended ALGOL, one of the languages used for programing the Burroughs B 5500 Electronic Information Processing System, is based on the definitive "Revised Report on the Algorithmic Language ALGOL 60" (Communications of the ACM, Vol. 6, No. 1; January, 1963). Extended ALGOL implements virtually all of ALGOL 60, and adds certain extensions which are necessary to handle situations peculiar to computer operations: input/output operations, partial-word operations, character manipulation, and diagnostic facilities. The extensions which have been added were designed with the philosophy used in the design of ALGOL 60.

SECTION 1

STRUCTURE OF THE LANGUAGE

GENERAL.

ALGOL 60 deals with the formation of rules for calculation of a value or values by means of a computer. Burroughs Extended ALGOL also includes the means required by a programmer to communicate with the computing equipment.

Extended ALGOL employs a vocabulary of reserved words and symbols. These reserved words and symbols may not be used in a program for any purpose other than that 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 concatenate.

The results produced by the evaluation of arithmetic, Boolean, and concatenate 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 either a compound statement or a block; each provides a method for grouping related statements. If a declaration of identifiers appears immediately after the word BEGIN, the statement group is called a block. A statement group may contain subordinate statement groups. A program is a grouping of statements, usually a block. (To be completely precise, a program may also be a compound statement.)

CONVENTIONS USED IN THE DESCRIPTION OF THE LANGUAGE.

The syntax of the language is described through the use of metalinguistic symbols. These symbols have the following meanings:

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," and separates the metalinguistic variable on the left of the formula from its 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.

The above 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. The entire set of such formulas defines the constructs of Extended ALGOL.

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.

Spaces have been used between language elements for readability in this document, but in general, spaces may be used or omitted except as prescribed herein. See pages 2-3 and 2-4 in particular.

In order to indicate specifically the differences between Extended ALGOL and ALGOL 60, each metalinguistic formula is preceded by an underlined number. These numbers have the following meanings:

a. 1 Same as ALGOL 60 except for character set.*
b. 2 Different from ALGOL 60.
c. 3 In addition to ALGOL 60 (all or in part).

To illustrate the use of syntax, the following example is offered:

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

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

The metalinguistic formula defines a recursive relationship by which a construct called an identifier may be formed. Evaluation

*Formulas preceded by the number 1 represent the material presented in "Report on the Algorithmic Language, ALGOL 60" (Communications of the Association for Computing Machinery, Vol. 3, No. 5; May, 1960) as modified by the changes which were made during the Rome meeting of the ALGOL Committee (April 2-3, 1962).

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.

The number 1 indicates the departure of the defined construct from the definitions of ALGOL 60, as noted above.

CHARACTER SET.

SYNTAX.

The syntax for ⟨character⟩ is as follows:

3 ⟨character⟩ ::= ⟨string character⟩ | ⟨string bracket character⟩ | ⟨illegitimate character⟩

3 ⟨string character⟩ ::= ⟨visible string character⟩ | ⟨single space⟩

3 ⟨visible string character⟩ ::= .|[|(|<|←|&|$|*|)|;|≤|-|/|
,|%|=|]|#|@|:|>|≥|+|A|B|C|
D|E|F|G|H|I|x|J|K|L|M|N|O|
P|Q|R|≠|S|T|U|V|W|X|Y|Z|0|
1|2|3|4|5|6|7|8|9

3 ⟨single space⟩ ::= {a single unit of horizontal spacing which is blank}

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

3 ⟨string bracket character⟩ ::= "

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

3 ⟨illegitimate character⟩ ::= ?

NOTE

The illegitimate character ? is not used in writing Extended ALGOL programs. It serves to represent any illegitimate card code detected during a card read operation. It is shown here merely to complete the illustration of the character set.

SEMANTICS.

The above character set has been defined; therefore, the

definition of Extended ALGOL will reflect the use of this character set. The visible string characters, the string bracket character, the single space, and the illegitimate character provide a total of 64 characters.

SECTION 2

BASIC COMPONENTS: BASIC SYMBOLS, IDENTIFIERS, NUMBERS, AND STRINGS

GENERAL.

SYNTAX.

The syntax for ⟨basic symbols⟩ is as follows:

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

SEMANTICS.

The entire Extended ALGOL language is formed from the above basic symbols.

LETTERS.

SYNTAX.

The syntax for ⟨letter⟩ is as follows:

```
1  ⟨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
```

SEMANTICS.

The alphabet defined for Extended ALGOL is restricted to the uppercase letters of the English alphabet. The lowercase letters are specifically disallowed. Individual letters do not have individual meaning but serve to form identifiers and strings (see page 2-5, Identifiers, and pages 2-7 and 2-8, Strings).

DIGITS.

SYNTAX.

The syntax for ⟨digit⟩ is as follows:

```
1  ⟨digit⟩ ::= 0|1|2|3|4|5|6|7|8|9
```

SEMANTICS.

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

LOGICAL VALUES.

SYNTAX.

The syntax for ⟨logical value⟩ is as follows:

```
1  ⟨logical value⟩ ::= TRUE|FALSE
```

SEMANTICS.

Logical values are the only values defined for Boolean quantities (see pages 9-2 and 9-3, Type Declarations).

DELIMITERS.

SYNTAX.

The syntax for ⟨delimiter⟩ is as follows:

```
1  ⟨delimiter⟩ ::= ⟨operator⟩ | ⟨separator⟩ | ⟨bracket⟩ |
                   ⟨declarator⟩ | ⟨specificator⟩
1  ⟨operator⟩ ::= ⟨arithmetic operator⟩ | ⟨relational operator⟩
                  | ⟨logical operator⟩ | ⟨sequential operator⟩
3  ⟨arithmetic operator⟩ ::= +|-|x|/|DIV|*|MOD|TIMES
1  ⟨relational operator⟩ ::= <|≤|=|≥|>|≠|LSS|LEQ|EQL|GEQ|
                             GTR|NEQ
3  ⟨logical operator⟩ ::= EQV|IMP|OR|AND|NOT
3  ⟨sequential operator⟩ ::=  GO|TO|IF|THEN|ELSE|FOR|DO|READ|
                              WRITE|DOUBLE|RELEASE|DS|TOGGLE|
                              JUMP|SKIP|DB|DI|SET|LOCK|ZIP|CI|
                              SC|DC|RESET|SB|SI|TALLY|REWIND|
                              CLOSE|SPACE|FILL|PAGE|DBL|NO|
                              BREAK
3  ⟨separator⟩ ::= ,|.|@|:|;|←|&|⟨single space⟩|STEP|UNTIL|
                   WHILE|COMMENT|LOC|WDS|ADD|SUB|LIT|CHR|NUM|
                   ZON|DEC|OCT|WITH|:=
3  ⟨bracket⟩ ::= (|)|[|]|"|BEGIN|END|#|LB|RB
3  ⟨declarator⟩ ::= OWN|BOOLEAN|INTEGER|REAL|ARRAY|SWITCH|LABEL|
                    LOCAL|FORWARD|SAVE|PROCEDURE|STREAM|LIST|
                    FORMAT|IN|OUT|MONITOR|DUMP|FILE|ALPHA|
                    DEFINE|REVERSE
3  ⟨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 1-4, alternate representations of certain delimiters are provided as follows:

LSS	$<$
LEQ	$\leq$
EQL	$=$
GEQ	$\geq$
GTR	$>$
NEQ	$\neq$
TIMES	x
LB	[
RB	]
:=	$\leftarrow$

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 below. 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 the ALGOL 60 Reference Language, 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 Extended

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 declarations. 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 reserved words END, ELSE, UNTIL}	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.

IDENTIFIERS.

SYNTAX.

The syntax for ⟨identifier⟩ is as follows:

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

Examples:

```
I
ID
A5
G76D3
ARITHMETICMEAN
```

SEMANTICS.

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

RESTRICTIONS.

Reserved words of Extended ALGOL may not be used as identifiers.

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.

NUMBERS.

SYNTAX.

The syntax for ⟨number⟩ is as follows:

1 ⟨number⟩ ::= ⟨unsigned number⟩ | +⟨unsigned number⟩ | -⟨unsigned number⟩

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

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

1 ⟨exponent part⟩ ::= @ ⟨integer⟩

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

1 ⟨integer⟩ ::= ⟨unsigned integer⟩ | +⟨unsigned integer⟩ | -⟨unsigned integer⟩

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

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.

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 first series of examples under Numbers (page 2-6) shows the lower and upper limits of the absolute values of numbers, of both INTEGER and REAL types, which are allowed in Extended ALGOL. (See also page 4-8, Numerical Limitations and Significant Digits.)

RESTRICTION.

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

STRINGS.

SYNTAX.

The syntax for ⟨string⟩ is as follows:

2 ⟨string⟩ ::= "⟨proper string⟩" | "⟨string bracket character⟩"

2 ⟨proper string⟩ ::= ⟨string character⟩ | ⟨proper string⟩ ⟨string character⟩

Examples:

String:

```
"ALGOL"
" " "
"THE FOLLOWING TABLE OF RESULTS WAS BASED ON FORMULA:
 A = B*C"
```

Proper String:

```
#
#A@FG
ALGOL 60
```

SEMANTICS.

Strings are of two forms:

a. A proper string delimited on both ends with the string bracket character.

b. " " "

USE OF STRINGS.

Strings can be used to form arithmetic expressions (see pages 4-1 through 4-8, Arithmetic Expressions), FORMAT declarations (see pages 9-19 through 9-32, FORMAT Declarations), FILL statements (pages 6-6 and 6-7) and destination string statements (pages 11-8 through 11-12, Destination String Statement).

RESTRICTION.

A string may not exceed 63 characters in length.

<u>LETTER STRING</u>.

SYNTAX.

The syntax for ⟨letter string⟩ is as follows:

<u>3</u> ⟨letter string⟩ ::= ⟨letter⟩ | ⟨letter string⟩ ⟨letter⟩ | ⟨space⟩ | ⟨letter string⟩ ⟨space⟩

<u>Examples</u>:

A
ABCDEF
ALGOL

SEMANTICS.

The letter string may incorporate a space as an integral part of its construct, and any spaces appearing between the delimiters of a letter string will be so interpreted.

A letter string may be used in a parameter delimiter for explanatory purposes in a parameter list (pages 6-10 through 6-15, Procedure Statements and section 10, PROCEDURE Declarations).

RESTRICTION.

A letter string may not exceed 63 characters in length.

CONSTITUENTS AND SCOPES.

The following kinds of quantities are declared in Extended ALGOL: simple variables, arrays, labels, switches, and procedures. In addition, certain other constituents are declared: files, formats, definitions, lists, forward references, and diagnostics.

The scope of any quantity or other constituent is the block in which the quantity or constituent is declared. All the above quantities and other constituents must be declared before they are referenced in any manner.

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 components⟩ is as follows:

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

SEMANTICS.

Combinations of basic components are used to create general components, which in turn are combined in building 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:

1 ⟨variable⟩ ::= ⟨simple variable⟩ | ⟨subscripted variable⟩
1 ⟨simple variable⟩ ::= ⟨variable identifier⟩
1 ⟨variable identifier⟩ ::= ⟨identifier⟩
1 ⟨subscripted variable⟩ ::= ⟨array identifier⟩ [⟨subscript list⟩]
1 ⟨array identifier⟩ ::= ⟨identifier⟩
1 ⟨subscript list⟩ ::= ⟨subscript expression⟩ | ⟨subscript list⟩, ⟨subscript expression⟩
1 ⟨subscript expression⟩ ::= ⟨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 pages 6-2 through 6-4, Assignment Statements). There are two forms of variables: simple and subscripted.

SIMPLE VARIABLES.

A simple variable is defined as being composed of a single variable identifier. The type of value that a simple variable may represent is defined by its type declaration (see pages 9-2 and 9-3, Type Declarations).

SUBSCRIPTED VARIABLES.

A subscripted variable represents a value which is a member of a set of values described by an array. A subscripted variable is composed of an array identifier and a subscript list. The array identifier specifies a particular array (see pages 9-3 through

9-6, ARRAY Declarations). 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.

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 by the following automatically-invoked transfer operation (see pages 3-9 and 3-10, Type Transfer Functions):

subscript value = ENTIER (value of subscript expression + 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.

PARTIAL WORD DESIGNATORS.

SYNTAX.

The syntax for ⟨partial word designator⟩ is as follows:

3 ⟨partial word designator⟩ ::= ⟨partial word operand⟩ . [⟨field description⟩]

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

3 ⟨field description⟩ ::= ⟨left bit of field⟩ : ⟨bits in field⟩

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

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

Examples:

Partial Word Designators:

X.[3:6]
Z(A).[1:1]
A[1,3].[9:39]
(Q + 3.543).[2:1]

Field Descriptions:

3:6
9:39
1:1
2:1
42:6

SEMANTICS.

The function of a partial word designator is to allow operations upon portions of the numerical or character representations assigned to certain quantities, rather than upon the entire representation or word.

VALUES ALLOWED FOR FIELD.

The value of a partial word operand is contained in a word which is 48 bits in length. The addressable bits in this word are numbered from left to right, from 1 to 47. (Bit 0 cannot be addressed.) Therefore, neither the value of the left bit of the field nor the value of the bits in the field may exceed 47. In addition, the sum of the left bit of the field and the bits in the field must not be greater than 48 (e.g., [46:2] specifies bit 46 and 47).

SWITCH FILE DESIGNATOR.

SYNTAX.

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

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

3 ⟨switch file identifier⟩ ::= ⟨identifier⟩

Examples:

SWHFI[I]
SWIFI[IF X > N THEN 0 ELSE 1]
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 (page 3-3). If the value of the expression is outside the scope of the switch file list, the file so referenced is undefined.

SWITCH FORMAT DESIGNATOR.

SYNTAX.

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

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

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 (see page 3-3).

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

SWITCH LIST DESIGNATOR.

SYNTAX.

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

```
3 ⟨switch list designator⟩ ::= ⟨switch list identifier⟩
                                [⟨subscript expression⟩]
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 position 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 (see page 3-3).

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:

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

1 ⟨procedure identifier⟩ ::= ⟨identifier⟩

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

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

2 ⟨actual parameter⟩ ::= ⟨expression⟩ | ⟨array row⟩ | ⟨array identifier⟩ | ⟨procedure identifier⟩ | ⟨file identifier⟩ | ⟨format identifier⟩ | ⟨list identifier⟩ | ⟨switch identifier⟩ | ⟨switch file identifier⟩ | ⟨switch format identifier⟩ | ⟨switch list identifier⟩ | ⟨switch file designator⟩ | ⟨switch format designator⟩ | ⟨switch list designator⟩

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

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 defines a single value. This value is produced by application of a given set of rules defined by a special form of a PROCEDURE declaration (see section 10, PROCEDURE Declarations). This set of rules is applied to the actual parameters of the function designator, thereby producing a single value.

A function designator may be used, depending upon its type, in either arithemtic or Boolean expressions (see pages 4-1 through 4-8. Arithmetic Expressions, and pages 4-8 through 4-14, BOOLEAN Expressions).

STANDARD FUNCTIONS.

The standard (or "intrinsic") functions supplied for Extended ALGOL are listed below, with appropriate definitions. Given that AE is an arithmetic expression, then:

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 logarithm of the value of AE.
EXP (AE)	Produces the exponential function of the value of AE, i.e., e^{AE}.

These functions are understood to operate indifferently on arguments both of type REAL and type INTEGER. They all yield values

of type REAL, except for SIGN (AE) which produces a value 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.

THE 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 (AE) Parameters

Parameter	Result	Type
TIME (0)	Current date (e.g., 64323 (year and day))	ALPHA
TIME (1)	Start time plus elapsed time since last start time, in sixtieths of a second	INTEGER
TIME (2)	Elapsed processor time, in sixtieths of a second	INTEGER
TIME (3)	Elapsed I/O time, in sixtieths of a second	INTEGER
TIME (4)	Value of 6-bit machine timer	INTEGER

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

TYPE TRANSFER FUNCTIONS.
In addition to the set of standard functions provided for Extended ALGOL, a set of type transfer functions is also provided.

These type transfer functions are listed below, with their definitions following.

```
ENTIER   (AE)
REAL     (BE)
BOOLEAN  (AE)
```

ENTIER. The function ENTIER yields a value of type INTEGER. This function is understood to transfer an expression of type REAL to an expression of type INTEGER, and produces the value which is the largest integer not greater than the value of the arithmetic expression.

REAL. The function REAL (BE) yields a value of type REAL. The use of this function does not alter the internal system representation of the value, but allows arithmetic operations to be carried out on quantities which have been declared type BOOLEAN.

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

BOOLEAN. The function BOOLEAN (AE) yields a value of type BOOLEAN. The use of this function does not alter the internal system representation of the value, but allows BOOLEAN operations to be carried out on arithmetic quantities.

The functions REAL and BOOLEAN, used in conjunction, allow for handling masking operations since the logical operators (page 4-13) operate on the entire word in the system.

INTERROGATE FUNCTION.

STATUS. The function STATUS (AE,AE) causes the MCP to perform different actions for a specified data communications terminal unit and buffer. The STATUS function is described in detail beginning on page 6-59.

SECTION 4

EXPRESSIONS

GENERAL.

SYNTAX.

The syntax for ⟨expression⟩ is as follows:

3 ⟨expression⟩ ::= ⟨arithmetic expression⟩ | ⟨Boolean expression⟩ | ⟨designational expression⟩ | ⟨concatenate expression⟩

SEMANTICS

Expressions, which are basic to any algorithmic process, are rules 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:

1 ⟨arithmetic expression⟩ ::= ⟨simple arithmetic expression⟩ | ⟨if clause⟩ ⟨arithmetic expression⟩ ELSE ⟨arithmetic expression⟩

1 ⟨simple arithmetic expression⟩ ::= ⟨term⟩ | ⟨adding operator⟩ ⟨term⟩ | ⟨simple arithmetic expression⟩ ⟨adding operator⟩ ⟨term⟩

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

1 ⟨term⟩ ::= ⟨factor⟩ | ⟨term⟩ ⟨multiplying operator⟩ ⟨factor⟩

1 ⟨factor⟩ ::= ⟨primary⟩ | ⟨factor⟩*⟨primary⟩

3 ⟨primary⟩ ::= ⟨unsigned number⟩ | ⟨variable⟩ | ⟨function designator⟩ | (⟨arithmetic expression⟩) | ⟨partial word designator⟩ | ⟨string⟩ | ⟨concatenate expression⟩ | ⟨assignment statement⟩

1 ⟨adding operator⟩ ::= + | -

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

Examples:

Arithmetic Expressions:

```
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.[9:39]
"ALPHA"
```

SEMANTICS.

An arithmetic expression is a rule for computing a numerical value. Arithmetic expressions may be divided 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 pages 4-6 and 4-7, Operators and Types.

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

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

Table 4-1 (cont)

Represented Values of Primaries in Arithmetic Expression

Name of Primary	Value Represented
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

RESTRICTION. If a primary is a string, it should generally not exceed six characters in length. It is permissible to use a seven-character string, but a seven-character string must not be used in comparisons or arithmetic operations, unless the left-most character of the string is a digit not greater than seven.

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

The Boolean expression is evaluated (see pages 4-8 through 4-14, 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.

RESTRICTION. If the primary is an assignment statement, partial word designators are not allowed in the left part list.

OPERATORS AND TYPES.

The constituent variables of an arithmetic expression must be of type INTEGER, REAL, or ALPHA. Note, however, that variables of type BOOLEAN may occur in an IF clause of an arithmetic expression. (See pages 9-2 and 9-3, Type Declarations.) Definitions of the various arithmetic operators are given in the paragraphs below.

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

Y DIV Z = SIGN (Y/Z) x ENTIER (ABS (Y/Z))

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

Y MOD Z = Y - [Z x (SIGN (Y/Z) x ENTIER (ABS (Y/Z)))]

The operator * denotes exponentiation. Its meaning depends on the types and values of the 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 x Y x...x Y (Z times).

Note 2: Y * Z = the reciprocal of Y x Y x...x Y (Z times).

Note 3: Y * Z = EXP(Z x 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. In arithmetic operations, operands of type ALPHA are treated as if they were of type REAL. All cases are shown in table 4-3.

Table 4-3

Types of Values Resulting from an Arithmetic Operation

OPERAND ON LEFT	OPERAND ON RIGHT	+, -, x	/	DIV	MOD	*
Integer	Integer	Integer	Real	Integer	Real	Note A
Integer	Real	Real	Real	Integer	Real	Real
Real	Integer	Real	Real	Integer	Real	Real
Real	Real	Real	Real	Integer	Real	Real

Note A: If the operand on the right is less than zero, Real; otherwise, Integer.

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: x / 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.

The expression between a left parenthesis and the matching right parenthesis is evaluated by itself and this value is used in subsequent calculations. Consequently, the desired order of execution of operations within an expression can always be arranged by appropriate positioning of parentheses.

NUMERICAL LIMITATIONS AND SIGNIFICANT DIGITS.

Normally the result of an arithmetic operation involving the operators +, -, and x is of type INTEGER if both operands are of type INTEGER (see pages 4-5 and 4-6, Operators and Types). If the value of the result exceeds 549755813887, however, it will become of type REAL (left-justified) to ensure that least significant rather than most significant digits are lost. Therefore, the maximum absolute value of type INTEGER (right-justified) that an arithmetic operation may yield is 549755813887.

Since the system utilizes an octal number system, the range of absolute real values that an arithmetic operation may yield can best be expressed as:

from (8 * 13 - 1) x 8 * 63 to (8 *12) x 8 * (-63), and zero

or approximately

from 4.314@68 to 8.758@-47, and zero.

BOOLEAN EXPRESSIONS.

SYNTAX.

The syntax for ⟨Boolean expression⟩ is as follows:

2 ⟨Boolean expression⟩ ::= ⟨simple Boolean⟩ | ⟨if clause⟩ ⟨Boolean expression⟩ ELSE ⟨Boolean expression⟩

1 ⟨simple Boolean⟩ ::= ⟨implication⟩ | ⟨simple Boolean⟩ EQV ⟨implication⟩

1 ⟨implication⟩ ::= ⟨Boolean term⟩ | ⟨implication⟩ IMP ⟨Boolean term⟩

1 ⟨Boolean term⟩ ::= ⟨Boolean factor⟩ | ⟨Boolean term⟩ OR ⟨Boolean factor⟩

1 ⟨Boolean factor⟩ ::= ⟨Boolean secondary⟩ | ⟨Boolean factor⟩ AND ⟨Boolean secondary⟩

1 ⟨Boolean secondary⟩ ::= ⟨Boolean primary⟩ | NOT ⟨Boolean primary⟩

3 ⟨Boolean primary⟩ ::= ⟨logical value⟩ | ⟨variable⟩ | ⟨function designator⟩ | ⟨relation⟩ | (⟨Boolean expression⟩) | ⟨partial word designator⟩ | ⟨concatenate expression⟩ | (⟨assignment statement⟩)

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

1 ⟨relational operator⟩ ::= < | ≤ | = | ≥ | > | ≠

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]

Simple Boolean Expressions:

TRUE
DIODE
NOT A ≠ C IMP GATE[1,2]

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 Secondaries:

TRUE
NOT A ≠ C

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])

SEMANTICS.
A Boolean expression is a rule for computing 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* 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 pages 4-13 and 4-14).

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.

*The logical operators are analyzed on page 4-13.

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.

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 4-10, 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.

The quantities which are used to form Boolean expressions must have been declared as type BOOLEAN (see pages 9-2 and 9-3, Type Declarations, and page 10-4, 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 pages 3-9 and 3-10, Type Transfer Functions).

RELATIONAL AND LOGICAL OPERATORS.

Two types of operators are defined for Boolean expressions: relational and logical. These operators are discussed in the following paragraphs.

RELATIONAL OPERATORS. The relational operators denote the following relations:

a. $<$ (is less than).
b. $\leq$ (is less than or equal to).
c. $=$ (is equal to).
d. $\geq$ (is greater than or equal to).
e. $>$ (is greater than).
f. $\neq$ (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 NOT (negation), AND (logical product), OR (logical sum), IMP (implication), and EQV (logical equivalence) is described in table 4-5.

Table 4-5

Operation of Logical Operators

B1	False	False	True	True
B2	False	True	False	True
NOT B1	True	True	False	False
B1 AND B2	False	False	False	True
B1 OR B2	False	True	True	True
B1 IMP B2	True	True	False	True
B1 EQV B2	True	False	False	True

PRECEDENCE OF OPERATORS.

The sequence of operations within a simple Boolean expression is generally from left to right, with the additional rules shown below. The following specific rules of precedence are defined:

a. First: Arithmetic expressions, according to the rules given on pages 4-7 and 4-8.

b. Second: $< \leq = \geq > \neq$

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.

RESTRICTION.

If the primary is an assignment statement, partial word designators are not allowed in the left part list.

DESIGNATIONAL EXPRESSIONS.

SYNTAX.

The syntax for ⟨designational expression⟩ is as follows:

1 ⟨designational expression⟩ ::= ⟨simple designational expression⟩ | ⟨if clause⟩ ⟨designational expression⟩ ELSE ⟨designational expression⟩

1 ⟨simple designational expression⟩ ::= ⟨label⟩ | ⟨switch designator⟩ | (⟨designational expression⟩)

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

1 ⟨switch identifier⟩ ::= ⟨identifier⟩

2 ⟨label⟩ ::= ⟨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 is a rule for obtaining a label of a statement (see Section 6, Statements). 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 self-evident. 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.

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 pages 4-8 through 4-14, Boolean Expressions). If a logical value of TRUE results, the designational expression following the ~~IF~~ THEN 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.

THE SUBSCRIPT EXPRESSION OF A SWITCH DESIGNATOR.
The value of the switch designator is defined by positive integer values 1, 2, 3, ..., n, where n is the number of entries in the switch list. If the value of the subscript expression is of a type other than integer, it is rounded to an integer in accordance with the rules applicable to the evaluation of subscripts (see page 3-3, Evaluation of Subscripts). If the value of the expression is outside the scope of the switch list, the switch designator is undefined, and program control continues in sequence.

CONCATENATE EXPRESSION.

SYNTAX.
The syntax for ⟨concatenate expression⟩ is as follows:

3 ⟨concatenate expression⟩ ::= ⟨left base⟩ ⟨link part⟩
3 ⟨left base⟩ ::= ⟨general primary⟩ | ⟨concatenate expression⟩
3 ⟨general primary⟩ ::= ⟨primary⟩ | ⟨Boolean primary⟩
3 ⟨link part⟩ ::= ⟨concatenate operator⟩ ⟨right base⟩ ⟨link description⟩
3 ⟨concatenate operator⟩ ::= &
3 ⟨right base⟩ ::= ⟨general primary⟩
3 ⟨link description⟩ ::= [⟨left bit of left base⟩ : ⟨left bit of right base⟩ : ⟨number of bits in link⟩]
3 ⟨left bit of left base⟩ ::= ⟨unsigned integer⟩
3 ⟨left bit of right base⟩ ::= ⟨unsigned integer⟩
3 ⟨number of bits in link⟩ ::= ⟨unsigned integer⟩

Examples:

E & D [36:42:6] & C [30:42:6] & B [24:42:6] & A [18:42:6]
S & (R + T) [42:42:6]

```
SQRT (C) & 1 [1:47:1]
X & Y [1:1:1] & Z [2:2:46]
M & N [4:4:6]
```

SEMANTICS.

The concatenate expression provides an efficient method of forming a primary, or Boolean primary, from selected bits of two or more primaries, or Boolean primaries, respectively.

A concatenate expression can utilize any number of concatenate operators; the expression is evaluated from left to right. Each concatenate operator causes a concatenated result to be formed; this concatenated result may be the final result of the expression, or a left base.

A concatenated result is formed by obtaining the value of the left base and then replacing a portion of it with a link made up of bits from the right base. The link is placed in the left base, starting at the bit specified by the left bit of left base. The link is obtained from the right base, starting with the bit designated by the left bit of right base. The number of bits in the link is designated by the value of the number of bits in the link.

RESTRICTIONS.

The integers used for designating the number of bits in the link, the left bit of the left base, and the left bit of the right base may range from 1 through 47. The sum of the left bit of the left base and the number of bits in the link, or the left bit of the right base and the number of bits in the link, must not exceed 48.

SECTION 5

PROGRAMS, BLOCKS, AND COMPOUND STATEMENTS

GENERAL.

SYNTAX.

The syntax for ⟨program⟩ is as follows:

2 ⟨program⟩ ::= ⟨block⟩.⟨space⟩ | ⟨compound statement⟩.⟨space⟩
1 ⟨block⟩ ::= ⟨unlabeled block⟩ | ⟨label⟩ : ⟨block⟩
1 ⟨unlabeled block⟩ ::= ⟨block head⟩ ; ⟨compound tail⟩
1 ⟨block head⟩ ::= BEGIN ⟨declaration⟩ | ⟨block head⟩ ;
⟨declaration⟩
1 ⟨compound tail⟩ ::= ⟨statement⟩ END | ⟨statement⟩ ;
⟨compound tail⟩
1 ⟨compound statement⟩ ::= ⟨unlabeled compound statement⟩ |
⟨label⟩ : ⟨compound statement⟩
1 ⟨unlabeled 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), it should be kept in mind that 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, constitute 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 9, Declarations), be specified to be local to the block in question. Such a declaration means that:

a. The entity represented by this identifier inside the block has no existence outside the block.

b. Any entity represented by the same identifier outside the block is completely inaccessible inside the block.

An identifier occurring within an inner block and not declared within that block will be nonlocal to it; that is, the identifier will represent the same entity inside the block and in the level or levels immediately outside it, up to and including the level in which the identifier is declared.

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

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

SEMANTICS.

The basic constituents of an Extended ALGOL Program are statements. Statements may be divided into three major groups: unconditional, conditional, and iterative statements. Unconditional statements are much like imperative sentences in the English language whereby a particular action is directly specified. A conditional statement may be compared to a conditional sentence since the function of the conditional statement is to ask a question and, depending upon the answer, select an appropriate course of action in the program. The iterative statement is used to describe a repetitive process.

Statements are normally executed in the order in which they are written. However, the sequence of operations may be changed by a conditional statement, or by an unconditional statement which explicitly defines its labeled successor.

NOTE

Only unconditional statements are further discussed in this section. Conditional statements and iterative statements are discussed in Sections 7 and 8 respectively.

UNCONDITIONAL STATEMENTS.

SYNTAX.

The syntax for ⟨unconditional statement⟩ is as follows:

1 ⟨unconditional statement⟩ ::= ⟨compound statement⟩ | ⟨block⟩ |
⟨basic statement⟩

1 ⟨basic statement⟩ ::= ⟨unlabeled basic statement⟩ | ⟨label⟩ :
⟨basic statement⟩

3 ⟨unlabeled basic statement⟩ ::= ⟨assignment statement⟩ |
⟨go to statement⟩ | ⟨dummy
statement⟩ | ⟨fill statement⟩ |
⟨library call statement⟩ |
⟨double statement⟩ |
⟨procedure statement⟩ |
⟨stream procedure call
statement⟩ | ⟨I/O statement⟩ |
⟨break-out statement⟩ |
⟨when statement⟩ | ⟨wait
statement⟩ | ⟨fault statement⟩ |
⟨zip statement⟩ |
⟨label equation statement⟩ |
⟨sort statement⟩ | ⟨merge
statement⟩ | ⟨edit and move
statement⟩ | ⟨disk I/O
statement⟩ | ⟨data communication
I/O statement⟩ | ⟨case
statement⟩ | ⟨ search statement⟩

SEMANTICS.

This group of statements includes (besides such basic constructs as the assignment, GO TO, and procedure statements) all the numerous kinds of input/output statements.

In the following paragraphs, each statement listed above will be discussed separately.

ASSIGNMENT STATEMENTS.

SYNTAX.

The syntax for ⟨assignment statement⟩ is as follows:

1 ⟨assignment statement⟩ ::= ⟨left part list⟩ ⟨arithmetic
expression⟩ | ⟨left part list⟩
⟨Boolean expression⟩

3 ⟨left part list⟩ ::= ⟨left part⟩ | ⟨left part list⟩ ⟨left part⟩ | ⟨partial word designator⟩ ←

1 ⟨left part⟩ ::= ⟨variable⟩ ← | ⟨procedure identifier⟩ ←

Examples:

Assignment Statements:

```
A ← A + 1
Q.[30:1] ← P ≥ R
P ← "RESULT"
A ← B ← C ← D ← 1
X.[47:1] ← X ← Z ← 0
```

Left Part List:

```
A ←
Q.[30:1] ←
X ← Y ← Z
```

Left Parts:

```
A ←
PROCID ←
```

SEMANTICS.

The assignment statement causes the value represented by an expression to be assigned to the variable appearing on the left of each assignment symbol. As shown in the last two examples above, one value may be assigned to two or more variables through the use of two or more assignment symbols. The operation of the assignment statement proceeds in three steps, as follows:

a. The subscript expressions of the left part variables are evaluated from left to right.

b. The expression following the right-most assignment symbol is evaluated.

c. The value of the expression is assigned to all the left part variables, with subscript expressions, if any, having values as determined in the first step.

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 pages 9-2 and 9-3, 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.

GO TO STATEMENTS.

SYNTAX.

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

1 ⟨go to statement⟩ ::= GO TO ⟨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 provides an unconditional transfer to the point in the program defined by the designational expression. When the designational expression is a label, the statement causes a transfer to the point in the program indicated by the label. In the case of a more complex designational expression, the path taken depends upon the label produced by the expression (see pages 4-14 through 4-16, Designational Expressions).

Labels must be declared in, and therefore are local to, the innermost block in which they appear as a statement label. A GO TO statement cannot lead from outside a block to a point inside that block; each block must be entered at the block head so that the associated declarations can be invoked.

The normal consecutive sequence of statement execution is unaltered in the case of an undefined switch designator (see page 3-3, Evaluation of Subscripts).

DUMMY STATEMENTS.

SYNTAX.

The syntax for ⟨dummy statement⟩ is as follows:

1 ⟨dummy statement⟩ ::= ⟨empty⟩

Examples:

```
L1:
EXIT:
```

SEMANTICS.

A dummy statement executes no operation. It may serve to place a label.

FILL STATEMENTS.

SYNTAX.

The syntax for ⟨fill statement⟩ is as follows:

```
3 ⟨fill statement⟩ ::= FILL ⟨array identifier⟩ [⟨row designator⟩]
                         WITH ⟨value list⟩
1 ⟨array identifier⟩ ::= ⟨identifier⟩
3 ⟨row designator⟩ ::= * | ⟨row⟩,*
3 ⟨row⟩ ::= ⟨arithmetic expression⟩ | ⟨row⟩, ⟨arithmetic
              expression⟩
3 ⟨value list⟩ ::= ⟨initial value⟩ | ⟨value list⟩, ⟨initial
                     value⟩
3 ⟨initial value⟩ ::= ⟨number⟩ | ⟨string⟩ | OCT ⟨octal number⟩
3 ⟨octal number⟩ ::= ⟨octal digit⟩ | ⟨octal number⟩ ⟨octal digit⟩
3 ⟨octal digit⟩ ::= 0|1|2|3|4|5|6|7
```

Examples:

```
FILL MATRIX[*] WITH 458.54, +546, - 1354.54@6, 16@-12
FILL GROUP[1,*] WITH .25, "ALGOL", " " ", OCT14, "365"
```

SEMANTICS.

The FILL statement causes one row of an array to be filled with a list of 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. The symbol * must appear in the right-most subscript position of the row designator.

If the value of a row designator is other than integer, it is rounded to an integer in accordance with the rules applicable to assignment statements (see page 6-4, Types).

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 1022. A defined identifier (see pages 9-7 through 9-9) must not be used in a FILL statement. There must be no space between OCT and the octal number which follows.

LIBRARY CALL STATEMENTS.

SYNTAX.
The syntax for ⟨library call statement⟩ is as follows:

3 ⟨library call statement⟩ ::= ZIP (⟨program designator⟩,
⟨library designator⟩)
3 ⟨program designator⟩ ::= ⟨arithmetic expression⟩
3 ⟨library designator⟩ ::= ⟨arithmetic expression⟩

Examples:

ZIP ("PROGIDT", "MCPROG")
ZIP (A, "MCPROG")
ZIP (ARA [I],B)

SEMANTICS.
Execution of a library call statement causes the program indicated by the program designator to be called out from the library tape indicated by the library designator. Immediately after causing the specified program to be called out, the calling program continues to be processed. The called program and calling program may then be multiprocessed.

RESTRICTION.
The values provided by the program designator and the library designator are interpreted as alpha variables. Therefore, these designators must be strings or arithmetic expressions which yield alpha values. Alpha values of less than seven characters are right-justified in a field of zeros.

DOUBLE STATEMENTS.
SYNTAX.
The syntax for ⟨double statement⟩ is as follows:

3 ⟨double statement⟩ ::= DOUBLE (⟨double expression⟩, ←,
⟨most-significant variable⟩,
⟨least-significant variable⟩)
3 ⟨double expression⟩ ::= ⟨double primary⟩ | ⟨double expression⟩,
⟨double primary⟩, ⟨double operator⟩ |
⟨double primary⟩, ⟨double expression⟩,
⟨double operator⟩
3 ⟨double primary⟩ ::= , ⟨double constant⟩ | ⟨most-significant
portion⟩, ⟨least-significant portion⟩

3 ⟨double operator⟩ ::= + | - | x | /
3 ⟨double constant⟩ ::= ⟨number⟩
3 ⟨most-significant variable⟩ ::= ⟨variable⟩
3 ⟨least-significant variable⟩ ::= ⟨variable⟩
3 ⟨most-significant portion⟩ ::= ⟨arithmetic expression⟩
3 ⟨least-significant portion⟩ ::= ⟨arithmetic expression⟩

Examples:

Storing single-length variable into array:

```
DOUBLE (X, 0, ←, MATRIX [0], MATRIX [1])
```

Double-length equivalent of RESULT ← (X - Y x Z) x.33333333333 is:

```
DOUBLE (HX, LX, HY, LY, HZ, LZ, x, -, ,
.33333333333333333333, x, ←, HRESULT, LRESULT)
```

Matrix Multiplication:

```
FOR I ← 0 STEP 1 UNTIL M DO
FOR J ← 0 STEP 1 UNTIL N DO
BEGIN
THIGH ← TLOW ← 0;
FOR K ← 0 STEP 1 UNTIL R DO
DOUBLE (A[I, 2 x K], A[I, 2 x K + 1], B[K, 2 x J],
       B[K, 2 x J + 1], x, THIGH, TLOW, +, ←,
       THIGH, TLOW);
DOUBLE (THIGH, TLOW, ←, C[I, 2 x J], C[I, 2 x J + 1])
END
```

SEMANTICS.

The DOUBLE statement assigns the double-length result of the double expression to the variables following the assignment operator, i.e., to the right-hand part of the statement.

Double-length values have the same range as real numbers. The difference is in the number of significant digits. Double-length

values may have a maximum of 26 significant octal digits.

Double constants are decimal numbers which are converted to their equivalent double-length octal value.

A double expression is a suffix Polish notation, i.e., an algebraic notation, which -- in contrast to the parentheses notation of common algebra -- omits the use of parentheses, brackets, and braces, using only operands and operators, arranged in sequence in such a manner that operations are executed in order of priority.

The evaluation of a double expression proceeds as follows. The occurrence of a double primary causes the double primary to be evaluated and the double-length value retained in the order in which the double primaries occur. The occurrence of a double operator causes the indicated arithmetic operation to be executed on the last two double primaries and the result to be saved. The evaluation continues in this fashion until the expression string is exhausted, leaving a double-length value as the result.

PROCEDURE STATEMENTS.

SYNTAX.

The syntax for ⟨procedure statement⟩ is as follows:

1 ⟨procedure statement⟩ ::= ⟨procedure identifier⟩ ⟨actual parameter part⟩
1 ⟨procedure identifier⟩ ::= ⟨identifier⟩
1 ⟨actual parameter part⟩ ::= ⟨empty⟩ | (⟨actual parameter list⟩)
1 ⟨actual parameter list⟩ ::= ⟨actual parameter⟩ | ⟨actual parameter list⟩ ⟨parameter delimiter⟩ ⟨actual parameter⟩

2 ⟨actual parameter⟩ ::= ⟨expression⟩ | ⟨array row⟩ | ⟨array identifier⟩ | ⟨procedure identifier⟩ | ⟨file identifier⟩ | ⟨format identifier⟩ | ⟨list identifier⟩ | ⟨switch identifier⟩ |

⟨switch file identifier⟩ |
⟨switch format identifier⟩ |
⟨switch list identifier⟩ |
⟨switch file designator⟩ |
⟨switch format designator⟩ |
⟨switch list designator⟩

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

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

Examples:

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

SEMANTICS.

A procedure statement causes a previously defined procedure, excluding typed procedures, to be activated (called for execution). (See Section 10, PROCEDURE Declarations.)

The procedure identifier references the procedure body which is to be executed. The actual parameter part contains a list of the actual parameters to be supplied to the procedure. A one-for-one correspondence must exist between the actual parameters in the actual parameter part and the formal parameters which appear in the formal parameter part of the PROCEDURE declaration. This correspondence is one of position, where the position of an actual parameter given in the procedure statement corresponds to the position of a formal parameter in the PROCEDURE declaration.

A general description of the operation of the procedure statement can be given as follows:

a. The formal parameters which are named in the VALUE part (call by value) of the PROCEDURE declaration are assigned the values of the corresponding actual parameters. These formal parameters are then treated as local to the procedure body.

b. The formal parameters not named in the VALUE part (call by name) are replaced, wherever they appear in the procedure body, by the corresponding actual parameters. Identifiers thus introduced into the procedure body may be identical to local identifiers already there. Each is handled in such a way, however, that no conflict occurs.

c. The procedure body, when modified as stated above, is then entered.

The above discussion covers the basic operation of the procedure statement. A more detailed analysis is necessary, however, because of the complexity of call by value, call by name, and execution of the procedure body.

VALUE ASSIGNMENT (CALL BY VALUE). The actual parameters that may be called by value are arithmetic, Boolean, and designational expressions. Where an arithmetic, Boolean, or designational expression is given as an actual parameter, the expression is evaluated according to the rules previously defined, and the resulting value is assigned to the appropriate formal parameter.

The evaluation of the actual parameters, and their subsequent assignment to the corresponding formal parameters, takes place according to the order indicated by the actual parameter list of the call statement. These assignments take place before entry is made into the procedure body.

NAME REPLACEMENT (CALL BY NAME). The actual parameters that may be called by name are general components, expressions, and array, switch, switch format, switch file, procedure, file, format, and list identifiers. The action taken in a call by name differs from that in a call by value. Instead of a value being assigned, the actual expression or pertinent identifier of the actual parameter replaces the corresponding formal parameter wherever it appears in the procedure body. A detailed analysis of this mechanism requires that each kind of allowable actual parameter be examined.

If a simple variable which is an actual parameter is called by name, the corresponding formal parameter is replaced, wherever it appears in the procedure body, by the identifier of the simple variable. The value represented by the simple variable is referenced each time the variable is encountered during the execution of the procedure body.

If a subscripted variable is an actual parameter, the subscripted variable is placed in the procedure body wherever the corresponding formal parameter appears. The subscript expression remains intact and is evaluated each time the subscripted variable is referenced during the execution of the procedure body.

If a partial word designator is given as an actual parameter, the partial word designator replaces the corresponding formal parameter throughout the procedure body, and is referenced each time it is encountered during the execution of the procedure body.

The formal parameter corresponding to a partial word designator must not appear in the left part of an assignment statement.

Where the actual parameter is a function designator, the corresponding formal parameter is replaced by the function designator wherever the formal parameter appears in the procedure body. The function designator is evaluated wherever it is encountered during the course of execution of the procedure body.

When an arithmetic, Boolean, or designational expression is called by name, the corresponding formal parameter is replaced by the expression in question. This expression is evaluated wherever it is encountered during the execution of the procedure body.

When the actual parameter called by name is an array identifier, the corresponding formal parameter is replaced by the array identifier wherever the formal parameter appears in the procedure body.

For those types of actual parameters thus far discussed, a call by value differs significantly from a call by name. A call by value (1) creates a quantity which is local to the procedure and which is identified by the formal parameter, (2) assigns to it the value

of the corresponding actual parameter, and (3) makes the corresponding actual parameter thereafter inaccessible to the procedure (unless the procedure is called again). A call by name, on the other hand, utilizes the actual parameter, or its constituents, as nonlocal quantities. Thus, the value of a quantity used as an actual parameter cannot be changed as a result of the procedure execution, provided that the corresponding formal parameter is called by value. If it is called by name, however, the actual parameter is accessible throughout the procedure and therefore can have its value altered.

If a switch, switch file, switch list, or switch format identifier is used as an actual parameter, the corresponding formal parameter is replaced by the respective identifier wherever the formal parameter occurs in the procedure body. Thus a switch, switch file, switch format which has been declared outside the procedure body can be accessed during the execution of the procedure body.

When a procedure identifier is passed as an actual parameter, the corresponding formal parameter is replaced by the procedure identifier wherever the formal parameter appears in the procedure body. Access can thus be made to another procedure which has been declared outside the procedure body.

When a file, format, or list identifier is passed as an actual parameter, the corresponding formal parameter is replaced by the identifier of the actual parameter wherever the formal parameter appears in the procedure body. Input/output statements in a procedure body can thus utilize files, formats, and lists which have been declared outside the procedure body.

RESTRICTIONS.
Formal and actual parameters must correspond both in type and in kinds of quantities.

A formal parameter which occurs as a left part variable in an assignment statement within the procedure body, and which is not called by value, can correspond only to an actual parameter which is a variable.

Any quantity that is nonlocal to a procedure is inaccessible to that procedure if that quantity is local to some other procedure, unless it has been declared OWN.

A stream procedure identifier must not be used as an actual parameter.

STREAM PROCEDURE CALL STATEMENT.

SYNTAX.

The syntax for ⟨stream procedure call statement⟩ is as follows:

3 ⟨stream procedure call statement⟩ ::= ⟨stream procedure identifier⟩ (⟨stream actual parameter list⟩)

3 ⟨stream procedure identifier⟩ ::= ⟨identifier⟩

3 ⟨stream actual parameter list⟩ ::= ⟨stream actual parameter⟩ | ⟨stream actual parameter list⟩, ⟨stream actual parameter⟩

3 ⟨stream actual parameter⟩ ::= ⟨stream value parameter⟩ | ⟨stream name parameter⟩

3 ⟨stream value parameter⟩ ::= ⟨arithmetic expression⟩ | ⟨Boolean expression⟩

3 ⟨stream name parameter⟩ ::= ⟨array identifier⟩ | ⟨array row⟩ | ⟨variable⟩ | ⟨file identifier⟩ | ⟨indexed file identifier⟩ | ⟨switch file designator⟩ | ⟨indexed switch file designator⟩ | ⟨format identifier⟩ | ⟨switch format designator⟩

3 ⟨indexed file identifier⟩ ::= ⟨file identifier⟩ (⟨arithmetic expression⟩)

3 ⟨indexed switch file designator⟩ ::= ⟨switch file designator⟩ (⟨arithmetic expression⟩)

Examples:

```
EDIT(FILEID, A)
MOVE(A[*], X, I + 1, A[I + 2])
SP(SWF[2] (0), F1(0))
```

SEMANTICS.

A stream procedure call statement causes the execution of a stream procedure body which has been previously defined by a STREAM PROCEDURE declaration (Section 11). It supplies the actual parameters to the stream procedure and then transfers control to the stream procedure body.

A stream procedure call statement must have an actual parameter part which may not be empty.

A one-to-one correspondence must exist between the actual parameters in the stream procedure call and the formal parameters appearing in the STREAM PROCEDURE declaration.

The formal parameters may be called by name or by value. Accordingly, the actual parameters are in two classes:

a. Stream value parameters which correspond to the VALUE part of the STREAM PROCEDURE declaration.

b. Stream name parameters which correspond to the call-by-name formal parameters of the STREAM PROCEDURE declaration.

STREAM VALUE PARAMETERS. Stream value parameters may be only arithmetic or Boolean expressions. The corresponding formal parameters are given the values of the stream actual parameters when the stream procedure call statement is executed.

STREAM NAME PARAMETERS. Stream name parameters may be array identifiers, file identifiers, indexed file identifiers, indexed switch file designators, variables, array rows, format identifiers,

switch format designators, and switch file designators. When the stream procedure call statement is executed, absolute addresses are supplied to the corresponding formal parameters.

If a stream name parameter is a file identifier or a switch file designator, an address of a pointer word is supplied. This pointer word contains the address of the file buffer. If a stream name parameter is a variable, the address of that variable is supplied.

It should be noted that arrays are mapped in memory by rows. Elements of a row are contiguous, but rows are not contiguous.

If a stream name parameter is an array identifier, the address supplied is:

a. The address of the lowest element of the array for a single-dimensional array.

b. The address of the lowest element of the first (highest-level) block of descriptors for a multidimensional array.

If a stream name parameter is an array row, the address supplied is that of the lowest element of that row.

If a stream name parameter is an indexed file identifier or indexed switch file designator, the address supplied is that of the left-most character of a word in the current buffer being used by the indicated file. The word is designated by the value of the arithmetic expression in the indexed file identifier or indexed switch file designator. The words in the buffer are numbered starting with zero. If the value of the arithmetic expression is of a type other than INTEGER, it is converted to an integer in accordance with the rules applicable to assignment statements (see page 6-4, Types).

A declared format specification may be changed during processing by means of a stream procedure. In such cases, a format

identifier or a switch format designator may be passed as an actual parameter. The address supplied will be that of the first word of the specified format array.

RESTRICTIONS.

Designational expressions, switch identifiers, switch file identifiers, switch format identifiers, switch list identifiers, switch list designators, list identifiers, and call by name expressions are not allowed as actual parameters to stream procedures.

INPUT/OUTPUT STATEMENTS.

SYNTAX.

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

```
3 ⟨I/O statement⟩ ::= ⟨read statement⟩ | ⟨write statement⟩ |
                      ⟨release statement⟩ | ⟨space statement⟩ |
                      ⟨close statement⟩ | ⟨rewind statement⟩ |
                      ⟨lock statement⟩
```

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.

READ STATEMENTS.

SYNTAX.

The syntax for ⟨read statement⟩ is as follows:

```
3 ⟨read statement⟩ ::= READ ⟨direction⟩ (⟨input parameters⟩)
                        ⟨action labels⟩
3 ⟨direction⟩ ::= ⟨empty⟩ | REVERSE
3 ⟨input parameters⟩ ::= ⟨file part⟩ ⟨buffer release⟩,
                          ⟨format and list part⟩ | ⟨file part⟩
                          ⟨buffer release⟩ | ⟨file part⟩
                          ⟨buffer release⟩, ⟨free-field part⟩
```

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

3 ⟨buffer release⟩ ::= ⟨empty⟩ | [NO]

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

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

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

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

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

3 ⟨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 below).

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 follows:

⟨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:

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

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

3 ⟨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:

<u>Example</u>:

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@29
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.

RELEASE STATEMENTS.

SYNTAX.

The syntax for ⟨release statement⟩ is as follows:

3 ⟨release statement⟩ ::= RELEASE (⟨file part⟩) | RELEASE (⟨file part⟩, ⟨word count⟩)

3 ⟨word count⟩ ::= ⟨arithmetic expression⟩

Examples:

```
RELEASE(FILEID)
RELEASE (FILEID, AE)
```

SEMANTICS.

If the file is an input file, the RELEASE statement causes the the buffer to be refilled with new input.

If the file is an output file, the RELEASE statement causes the information in the buffer to be written on the output unit.

The number of words released is determined by the buffer size,

unless a RELEASE statement indicates a word count; thereafter, the buffer size is considered equal to the last word count indicated by a RELEASE statement.

RESTRICTIONS.

The word count of a RELEASE statement must not exceed the buffer size. Due to the fact that READ, SPACE, and WRITE statements cause I/O descriptors associated with a file to be altered, RELEASE statements should not be mixed with READ, SPACE, or WRITE statements referencing the same file. One exception to this is that on files using only one buffer and containing only unblocked records, RELEASE statements may be freely mixed with READ, SPACE, and WRITE statements. RELEASE statements are not allowed on disk files when operating under the File Security System, and if used, the program will be terminated.

SPACE STATEMENTS.

SYNTAX.

The syntax for ⟨space statement⟩ is as follows:

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

3 ⟨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:

```
3 ⟨write statement⟩ ::= WRITE (⟨output parameters⟩)
3 ⟨output parameters⟩ ::= ⟨file part⟩ ⟨carriage control⟩ |
                          ⟨file part⟩ ⟨carriage control⟩
                          ⟨format and list part⟩
3 ⟨carriage control⟩ ::= [PAGE] | ⟨skip to channel⟩ | [DBL] |
                         [NO] | ⟨empty⟩
3 ⟨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 1 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.

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 <u>smaller</u>. 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:

3 ⟨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:

3 ⟨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:

3 ⟨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 multi-file input tapes, 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. The next reference to this file must be a READ in the opposite direction from that of the prior READ on the file. A system halt can occur if this rule is violated.

BREAK-OUT STATEMENTS.

SYNTAX.

The syntax for ⟨break-out statement⟩ is as follows:

3 ⟨break-out statement⟩ ::= BREAK

Examples:

BREAK
IF X = 2 THEN BREAK

SEMANTICS.

The break-out statement causes all information necessary to restart the program, from the point where the statement appeared, to be written on magnetic tape. The program continues in sequence after execution of the break-out statement.

WHEN STATEMENT.

SYNTAX.

The syntax for ⟨when statement⟩ is as follows:

3 ⟨when statement⟩ ::= WHEN (⟨seconds⟩)
3 ⟨seconds⟩ ::= ⟨arithmetic expression⟩

Examples:

WHEN (X)
WHEN (10)

SEMANTICS.

The WHEN statement provides a means for a program to suspend itself from processing for a given number of seconds. The parameter ⟨seconds⟩ specifies the number of seconds to suspend the program using this statement.

When a program executes a WHEN statement, the MCP suspends processing of that program and allows other processing to take place. Subsequently, after the designated number of seconds have elapsed, control is returned to the program and processing commences at the point immediately following the WHEN statement.

WAIT STATEMENT.

SYNTAX.

The syntax for ⟨wait statement⟩ is as follows:

3 ⟨wait statement⟩ ::= WAIT (⟨absolute address⟩, ⟨mask⟩)
3 ⟨absolute address⟩ ::= ⟨arithmetic expression⟩
3 ⟨mask⟩ ::= ⟨arithmetic expression⟩

Examples:

```
WAIT (ADDRESS, MASK)
WAIT (REALSTREAMADDR(A), 1023)
```

SEMANTICS.

The WAIT statement provides a program the means to temporarily suspend its processing until a specified Re-Initiate condition exists.

The first parameter ⟨absolute address⟩ of the WAIT statement must provide the absolute address of a test word. This address must be in the fifteen low-order bits of the parameter, while the remaining bits are ignored.

The second parameter ⟨mask⟩ of the WAIT statement is a mask which the test word is compared against. If a bit in the test word is to be tested, the corresponding bit in the value of the parameter ⟨mask⟩ must be set to ONE (1).

A Re-Initiate condition exists whenever any corresponding bit position of the test word and the mask expression both have a value of ONE (1). The value of the test word must be changed by another program since the program executing the WAIT statement has been suspended.

When a program executes a WAIT statement, the MCP suspends processing of that program, but allows other processing to take place. Periodically, the MCP examines the test word to determine if a Re-Initiate condition exists. When this occurs, control is returned to the program at the point immediately following the WAIT statement. If the test word value is not changed by some other program, the program which executed the WAIT statement is suspended indefinitely.

FAULT STATEMENT.

SYNTAX.

The syntax for ⟨fault statement⟩ is as follows:

```
3 ⟨fault statement⟩ ::= ⟨fault type⟩ ← 0 |
                        ⟨fault type⟩ ← ⟨designational
                        expression⟩
3 ⟨fault type⟩ ::= EXPOVR | INTOVR | 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 9-40).

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:

```
3 ⟨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 the 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:

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

3 ⟨label equation information⟩ ::= ⟨multi-file identification⟩ |
⟨multi-file identification⟩,
⟨file identification⟩ |
⟨multi-file identification⟩,
⟨file identification⟩,
⟨reel number⟩ |
⟨multi-file identification⟩,
⟨file identification⟩,

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
30 word segment { 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

```
                                        ⟨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 digit⟩
3 ⟨multi-file identification⟩ ::= ⟨arithmetic expression⟩ | *
3 ⟨file identification⟩ ::= ⟨arithmetic expression⟩ | *
3 ⟨reel number⟩ ::= ⟨arithmetic expression⟩ | *
3 ⟨date⟩ ::= ⟨arithmetic expression⟩ | *
3 ⟨cycle number⟩ ::= ⟨arithmetic expression⟩ | *
3 ⟨output media digit⟩ ::= ⟨arithmetic expression⟩ | *
```

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 before the designated file is 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 digit, 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.

Table 6-2

Values for Output Media Digit

⟨output media digit⟩ value	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
14	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

SORT STATEMENT AND MERGE STATEMENT.

Because of the requirements of the SORT and MERGE statement parameters, these two statements are explained in Section 12 of this manual.

EDIT AND MOVE STATEMENT.

SYNTAX.

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

```
3 ⟨edit and move statement⟩ ::= ⟨edit and move read⟩ |
                                ⟨edit and move write⟩
3 ⟨edit and move read⟩ ::= READ (⟨array row⟩,
                           ⟨format and list part⟩) |
                           READ (⟨array row⟩,
                           ⟨free field part⟩)
3 ⟨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:

```
3 ⟨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:

```
3 ⟨disk read statement⟩ :: READ ⟨direction⟩
                           (⟨disk input parameters⟩)
                           ⟨action labels⟩
3 ⟨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⟩
3 ⟨record address and release part⟩ ::= [⟨address⟩] | [NO] |
                                        ⟨empty⟩
3 ⟨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 pointer 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 <u>not</u> 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 a 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:

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

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

3 ⟨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 where the information is placed into the buffer through the use of a stream procedure.

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:

```
 3 ⟨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 9). 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 physical 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 <u>next</u> record to be processed is given by the ⟨address⟩.

<u>DISK SPACE STATEMENT</u>.

SYNTAX.

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

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

<u>Examples</u>:

```
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:

3 ⟨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:

3 ⟨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:

```
3 ⟨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.

DATA COMMUNICATIONS I/O STATEMENT.

SYNTAX.

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

```
3 ⟨data communications I/O statement⟩ ::= ⟨data comm read
                                          statement⟩ |
                                          ⟨data comm read lock
                                          statement⟩ |
```

<data comm read seek statement> |
<data comm write statement> |
<data comm write lock statement> |
<interrogate function> |
<data comm close statement> |
<data comm rewind statement>

SEMANTICS.

The data communications I/O statements allow the programmer to utilize the data communications equipment and converse with remote station devices. It is the responsibility of the Object Program to provide for various types of abnormal conditions that might occur. The MCP will make available to the user program all of the information that is known as to the status of a particular unit. This information is contained in a word called the "status word."

STATUS WORD.

FORMAT.

The format of the status word is as follows:

FIELD	DEFINITION
0:1	Flag bit = 0.
1:7	Undefined.
8:1	= 0 - a data transmission control unit (DTCU) is present.
9:4	Terminal unit number (1 through 15).
13:1	= 0 - automatic code translation was performed by DTCU on this I/O operation (remote stations device code to BCL).

FIELD (cont)	DEFINITION (cont)
14:4	Buffer address in terminal (0 through 15).
22:1	= 1 - station busy.
23:1	= 1 - abnormal condition sensed by the adapter.
24:1	= 1 - buffer is Read Ready.
25:1	= 0 - I/O operation terminated by a group mark. = 1 - I/O operation terminated by a Full Buffer condition with no group mark.
26:1	= 1 - BREAK key depressed during output.
27:1	= 1 - buffer is Write Ready. (The last message written did not contain a group mark.)
28:1	= 1 - input error (buffer overflow).
29:1	= 1 - write in process on remote station.
30:1	= 1 - remote station is not ready.
31:17	Undefined.

SEMANTICS.

The status word is provided to the user program as the first word in the I/O buffer on each READ statement execution on a data communications file. It can also be referenced implicitly by the use of the Interrogate function which has been provided for Object Programs which utilize data communications equipment. Any of the undefined fields on the status word will not be cleared to zero.

DATA COMMUNICATIONS READ STATEMENT.

SYNTAX.

The syntax for ⟨data comm read statement⟩ is as follows:

```
3 ⟨data comm read statement⟩ ::= READ (⟨data comm input
                                  parameters⟩) ⟨data comm input
                                  action labels⟩
3 ⟨data comm input parameters⟩ ::= ⟨file part⟩ ⟨data comm
                                    record address and release
                                    part⟩, ⟨format and list part⟩
                                    | ⟨file part⟩ ⟨data comm
                                    record address and release
                                    part⟩, ⟨free field part⟩
3 ⟨data comm record address and release part⟩ ::= (⟨terminal
                                                   buffer
                                                   specifier⟩
                                                   ⟨wait part⟩) |
                                                   ⟨empty⟩
3 ⟨terminal buffer specifier⟩ ::= ⟨arithmetic expression⟩ |
                                  ⟨empty⟩
3 ⟨wait part⟩ ::= ,⟨arithmetic expression⟩ | ⟨empty⟩
3 ⟨data comm input action labels⟩ ::= [⟨no-input label⟩ :
                                      ⟨abnormal-condition label⟩]
                                      | [⟨no-input label⟩] |
                                      [:⟨abnormal-condition
                                      label⟩] | ⟨empty⟩
3 ⟨no-input label⟩ ::= ⟨designational expression⟩
3 ⟨abnormal-condition label⟩ ::= ⟨designational expression⟩
```

Examples:

```
READ (DATACOM, 29, DATA[*])
READ (REMOTE(0&TU[9:44:4] & BUF[14:44:4]), FMT, LST)
READ (REMOTE(OLDSTATUS,2), 8, A[*]) [NOIN:WRONG]
READ (B300(0,1), FMT1, LST2) [NOGO:WHY]
```

SEMANTICS.

The data communications READ statement is the means by which information in a data communications buffer which has been attached to the Object Program by the MCP can be read and transferred to the list under control of the format.

The terminal buffer specifier indicates to the MCP the physical terminal and buffer from which the data is to be transferred. The terminal number must be in field 9:4 and the buffer number must be in the field 14:4 of this arithmetic expression. The terminal unit and buffer number specified must have integral values that correspond to equipment available in the hardware configuration.

If the terminal buffer specifier is zero or left empty, data will be read from any terminal buffer which has been attached to the Object Program by the MCP. This option makes it possible to read from any attached terminal buffer without performing READ statements on individual terminal buffers.

The wait part is checked to determine what action is to be taken on the action label for a no-input condition.

A branch to the no-input label will be made if:

a. A READ statement could not be executed within the time specified by the wait part because another job had exclusive use of the terminal buffer.

b. A READ statement could not be executed within the time specified by the wait part because the terminal buffer did not contain any input.

If a READ statement which does not contain a no-input label is executed, the job will be suspended until the terminal buffer becomes Read Ready, regardless of how long that might be.

A branch to the abnormal-condition label will be made after the READ statement has been executed if any of the abnormal conditions

are sensed. The Object Program must examine the status word provided on the READ, or obtained through the use of the Interrogate function. Some of the conditions that will cause an abnormal-condition branch are:

a. A parity error sensed on input.

b. An end-of-transmission, line-loss, or disconnect.

c. A buffer overflow occurred when information was entered as input before the buffer was read.

If the READ statement does not include an abnormal-condition label, and any abnormal condition is sensed, that program will continue without any indication that the condition occurred.

The wait part has significance only if the READ statement includes a no-input label. The value of the wait part is the number of seconds that the program is willing to wait for a READ READY condition of the terminal buffer.

If the wait part is empty in a READ statement that has a no-input label, the wait part value is equivalent to a wait part value of, for all practical purposes, infinity.

DATA COMMUNICATIONS READ LOCK STATEMENT.

SYNTAX.

The syntax for ⟨data comm read lock statement⟩ is as follows:

```
3 ⟨data comm read lock statement⟩ :: = READ LOCK (⟨data comm
                                        input parameters⟩)
                                        ⟨data comm action labels⟩
```

Examples:

```
READ LOCK (REMOTE(OLDSTATUS), 8, A[*]) [NOGO:WHY]
```

SEMANTICS.

The purpose of a data communications READ LOCK statement is to

allow a program to attach itself exclusively to a terminal buffer. This could be used if more than one remote station is attached to the terminal buffer (multi-point line). Programs that share such a terminal buffer must observe some mutually developed discipline. If a program uses the LOCK construct, the program should release the terminal buffer by excluding the word LOCK from the READ or WRITE statement. The LOCK functions in the same manner when a WRITE statement is executed.

The presence of LOCK on a READ or WRITE statement causes the following action:

a. Suspends the job until no other job has exclusive use of the specified terminal buffer.

b. Establishes this job as the exclusive user of the terminal buffer.

After exclusive use has been established, the READ or WRITE is performed. The exclusive use status is retained.

The absence of LOCK on a READ or WRITE statement causes the following action:

a. Suspends the job until no other job has exclusive use of the specified terminal buffer.

b. Performs the READ or WRITE operation.

c. Releases the exclusive use status _after_ the READ or WRITE has been performed if this job had exclusive use of the specified terminal buffer.

The semantics of the remainder of the READ LOCK statement are identical to the data communications READ statement.

DATA COMMUNICATIONS READ SEEK STATEMENT.

SYNTAX.

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

3 <data comm read seek statement> ::= READ SEEK (<data comm input parameters>) <data comm action labels>

Example:

READ SEEK (B300 (STATWORD))

SEMANTICS.

The data communications READ SEEK statement provides the programmer with the means of establishing this program as the exclusive user of the specified terminal buffer. In addition, the MCP will immediately fill the buffers of the specified file when a Read Ready Interrupt is received from the specified terminal buffer.

Only the file part and terminal buffer specifier are used in the READ SEEK statement. If other parts are included, they are ignored (including the action labels).

If the terminal buffer specifier is zero or empty, no action is performed and the program continues in sequence.

If the specified terminal buffer already has an exclusive user, no action takes place and the program continues in sequence.

If a READ SEEK had previously been performed on the specified file, the effect of the previous SEEK is negated. Therefore, a second READ SEEK will cause the previously specified terminal buffer to be released from the status caused by the first SEEK.

The seek feature allows asynchronous buffering of input from data communications equipment. When a READ is performed on a file which has been "seeked," the terminal buffer specifier of the READ statement is ignored and the first-in buffer is returned to the program.

DATA COMMUNICATIONS WRITE STATEMENT.

SYNTAX.

The syntax for ⟨data comm write statement⟩ is as follows:

3 ⟨data comm write statement⟩ ::= WRITE (⟨data comm output parameters⟩) ⟨data comm output action labels⟩

3 ⟨data comm output parameters⟩ ::= ⟨file part⟩ ⟨data comm record address and release part⟩ | ⟨file part⟩ ⟨data comm record address and release part⟩, ⟨format and list part⟩

3 ⟨data comm output action labels⟩ ::= [⟨output-impossible label⟩ : ⟨break label⟩] | [⟨output-impossible label⟩] | [: ⟨break label⟩] | ⟨empty⟩

3 ⟨output impossible label⟩ ::= ⟨designational expression⟩

3 ⟨break label⟩ ::= ⟨designational expression⟩

Examples:

```
WRITE (REMOTE(A[0]), 8, A[*])
WRITE(TYPER(STATWRD,15), FMT, LST) [NOTNOW:HOLDIT]
```

SEMANTICS.

The data communications WRITE statement provides the means of sending information to a remote station. The MCP will transfer the data in the list under control of the format to the specified terminal buffer.

A terminal buffer specifier must be present in all data communications WRITE statements. The format of the expression must be the same as specified for the READ statement for data communications. The terminal number and buffer number must correspond to equipment available in the hardware configuration.

The wait part is checked to determine what action is to be taken when an output-impossible condition is sensed.

A branch to the output-impossible label will be made if:

a. A WRITE statement could not be executed within the time specified by the wait part because another job had exclusive use of the terminal buffer.

b. A WRITE statement could not be executed within the time specified by the wait part because all of the output buffers are full.

c. A WRITE statement could not be executed because of the output buffers being full and the occurrence of one of the following conditions:

   1) The specified terminal buffer is Read Ready or Busy.

   2) An end-of-transmission, line-loss, or disconnect occurred.

If conditions a or b above occur and no output-impossible label has been provided, the job will be suspended until the WRITE can be performed.

If condition c above occurs and no output-impossible label has been provided, the job will be terminated.

A branch to the break label will be made on each WRITE statement after the BREAK key has been depressed on the remote station. This action will continue until a READ statement is executed on the specified terminal buffer. If no break label is included in the WRITE statement, the Object Program will not be aware that the BREAK key has been depressed.

When a branch is made to the output-impossible label, the reason for the branch must be determined by the examination of the status word. The status word can be obtained through the use of the Interrogate function only.

The wait part has no significance unless an output impossible label has been provided.

If the wait part is absent in a WRITE statement containing an output impossible label, the value of the wait part is considered to be, for all practical purposes, infinity.

DATA COMMUNICATIONS WRITE LOCK STATEMENT.

SYNTAX.

The syntax for ⟨data comm write lock statement⟩ is as follows:

```
3 ⟨data comm write lock statement⟩ ::= WRITE LOCK (⟨data comm
                                        output parameters⟩)
                                        ⟨data comm output
                                        action labels⟩
```

Examples:

```
WRITE LOCK (ALLMINE(A[0],SEC), 8, A[*])[NOPE:WHYNOT]
```

SEMANTICS.

The data communications WRITE LOCK statement allows a program to establish the specified terminal buffer to be assigned exclusively to this job. The semantics of the WRITE LOCK are the same as for the READ LOCK statement.

The semantics for the remainder of the WRITE LOCK statement are the same as for a normal WRITE statement.

INTERROGATE FUNCTION.

SYNTAX.

The syntax for ⟨interrogate function⟩ is as follows:

```
3 ⟨interrogate function⟩ ::= STATUS (⟨terminal buffer specifier⟩,
                               ⟨action part⟩)
3 ⟨action part⟩ ::= ⟨arithmetic expression⟩
```

Examples:

```
STATUS (O&I[9:44:4] &J [14:44:4], I)
STATUS (STATWRD, J)
```

SEMANTICS.

The purpose of this construct is to yield a value which is the status word. Since this is a function, the construct is an arithmetic expression.

The terminal buffer specifier must be specified and must have the same format as any status word.

The action part can only have a value of zero, one, three, or four.

If the action part value is zero, the value of the function will be the copy of the MCP's last status word at the time of the last interrupt on the specified terminal buffer.

If the action part value is one, the following action takes place:

a. The MCP will perform a Hardware Interrogate I/O operation on the terminal buffer.

b. The MCP's copy of the status word will be updated and the function will yield a copy of this newly-updated status word.

The action part with values of three or four are added to allow a program - which may be handling many remote users - to create a free file. The action taken by the MCP for each value follows:

a. If the value is three, then the USERCODE entry for the program is made empty. Any disk files that are entered into the disk directory are made free files.

b. If the value is four, then the program's USERCODE table is reestablished. Any disk files entered into the disk directory will be entered with the latest USERCODE entry.

The data communications Interrogate function has been extended to update the USERCODE table with the user code associated with

terminal buffer specified. This allows a program - which handles more than one user - to create and/or access disk files for the specific user code of each user.

The word STATUS may be used as declared in a program. If it is used, then the Interrogate function cannot be used within the scope of the declaration using STATUS.

DATA COMMUNICATIONS CLOSE STATEMENT.

SYNTAX.

The syntax for ⟨data comm close statement⟩ is as follows:

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

Examples:

    CLOSE (REMOTE)

SEMANTICS.

All CLOSE statements cause the same action for data communications files. The buffer areas are returned and the effect of a READ SEEK statement is released.

DATA COMMUNICATIONS REWIND STATEMENT.

SYNTAX.

The syntax for ⟨data comm rewind statement⟩ is as follows:

```
3 ⟨data comm rewind statement⟩ ::= REWIND (⟨file part⟩)
```

Examples:

    REWIND (REMOTE)

SEMANTICS.

A REWIND statement causes the core buffer areas to be returned and the effect of a READ SEEK statement is released.

CASE STATEMENT.

SYNTAX.

The syntax for ⟨case statement⟩ is as follows:

3 ⟨case statement⟩ ::= ⟨case statement header⟩ ⟨ compound statement⟩

3 ⟨case statement header⟩ ::= CASE ⟨arithmetic expression⟩ OF

Examples:

CASE I OF
BEGIN ⟨statement $_0$⟩;
⟨statement $_1$⟩;
.
.
.
⟨statement $_N$⟩ 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 selects which of the statements within the compound statement will be executed. Only that statement is executed and control is then transferred to the statement following the END of the compound statement. The statements within the compound statement can be any statement, including compound statements, blocks, CASE statements, and null statements. (A null statement is a dummy statement which occupies a position in a CASE statement.)

The value of the arithmetic expression, I, must be such that $0 \leq I \leq N$. If the value is less than zero or greater than N (N is the value of the last statement number), the Object Program will terminate with an invalid index.

SEARCH STATEMENT.

SYNTAX.

The syntax for the ⟨search statement⟩ is as follows:

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

3 ⟨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 if 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.

SECTION 7

CONDITIONAL STATEMENTS

GENERAL.

SYNTAX.

The syntax for ⟨conditional statement⟩ is as follows:

```
1 ⟨conditional statement⟩ ::= ⟨if statement⟩ |
                              ⟨if statement⟩ ELSE ⟨statement⟩ |
                              ⟨label⟩ : ⟨conditional statement⟩
2 ⟨if statement⟩ ::= ⟨if clause⟩ ⟨statement⟩
```

Examples:

Conditional Statements:

```
IF A > B THEN FOR I ← 1 STEP 1 UNTIL 5 DO R[I]←
P[I + 2]

IF A > B THEN A ← A + 1

IF GATE [1,2] AND GATE [1,3] THEN GO TO CHI ELSE
IF GATE [1,4] AND GATE [1,5] THEN GO TO BOS ELSE
GO TO ERROR1
```

IF Statements:

```
IF A > B THEN A ← A + 1
IF GATE [1,2] AND GATE [1,3] THEN GO TO L1
```

IF Clauses:

```
IF A > B THEN
IF GATE[1,2] AND GATE[1,3] THEN
```

SEMANTICS.

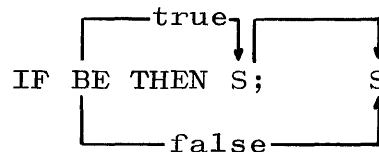
Conditional statements provide a means whereby the execution of a statement, or a series of statements, is dependent upon the logical value produced by a Boolean expression.

IF STATEMENT.

One of the permissible forms of a conditional statement is the IF statement. The IF statement operates as follows. The statement following the sequential operator THEN is executed if the logical value of the preceding Boolean expression is TRUE; otherwise, that statement is ignored.

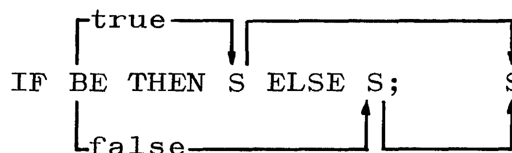
NOTE

In the examples which follow, BE represents any Boolean expression, and S represents any statement.



IF . . . ELSE STATEMENT.

A second form of the conditional statement contains the sequential operator ELSE. The operation of this conditional statement proceeds as follows. If the logical value produced by the Boolean expression is TRUE, the statement following the sequential operator THEN is executed and the statement following the sequential operator ELSE is ignored. If the logical value of the Boolean expression is FALSE, the statement following the sequential operator ELSE is executed and the statement following the sequential operator THEN is ignored.



NESTED IF STATEMENTS.

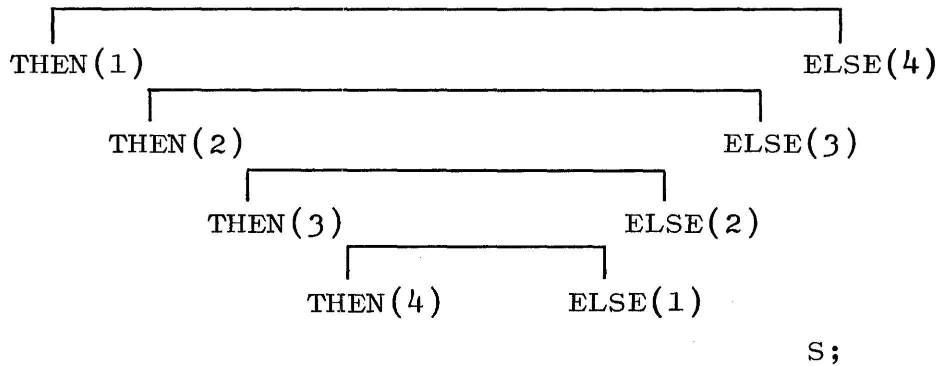
The statements following the delimiters THEN and ELSE, or both, may be conditional statements, or a series of nested conditional statements.

The Boolean expressions in the IF clauses of these statements are evaluated left to right in a manner similar to the evaluation of the conditional arithmetic expression. (See pages 4-1 through 4-8.)

When using nested conditional statements, the programmer must remain aware of the necessity of maintaining correspondence between the delimiters THEN and ELSE.

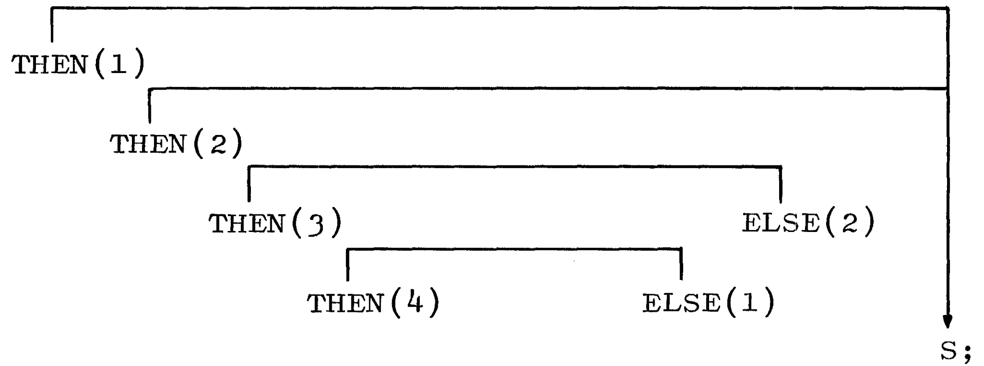
For explanatory purposes, let us assume that a given statement has equally matched THEN-ELSE pairs. In such a case, the innermost THEN and the immediately following (i.e., the innermost) ELSE will be treated as one pair, and from this center the pairs proceed outwards. This case is illustrated by:

Conditional S:



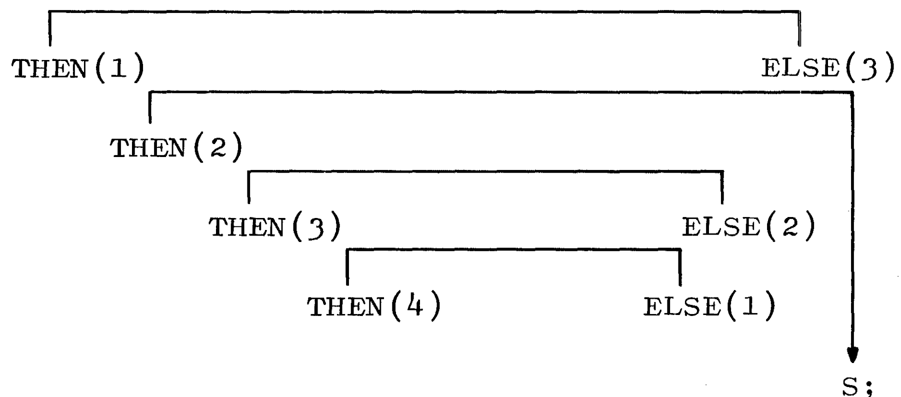
If THEN appears more often than ELSE in the statement, the pairs of delimiters are matched as described in the example above, and the first, and any following THEN not having a corresponding ELSE, will cause the program to transfer to the next statement if the Boolean expression yields a value of FALSE. This case is illustrated by:

Conditional S:



In the case illustrated by:

Conditional S:



the ALGOL Compiler would not produce the required result because ELSE(3) would be matched with THEN(2), and, if the Boolean expression preceding THEN(1) yielded a value of FALSE, the program would skip ELSE(3) and continue in sequence.

Since, however, a statement within a statement could itself be a compound statement or a block, the correspondence of the delimiters could be established clearly by defining the nested conditional statements as compound statements, the bracket words BEGIN and END indicating the different levels of nomenclature.

ENTERING A CONDITIONAL STATEMENT.

A GO TO statement may lead to a labeled statement within a conditional statement. The successor is then determined in the same way as if entrance had been made at the beginning of the conditional statement.

SECTION 8

ITERATIVE STATEMENTS

GENERAL.

SYNTAX.

The syntax for ⟨iterative statement⟩ is as follows:

```
3 ⟨iterative statement⟩ ::= ⟨for statement⟩ | ⟨do statement⟩ |
                            ⟨while statement⟩
```

SEMANTICS.

Iterative statements provide methods of forming loops in a program. They allow for the repetitive execution of a statement zero or more times.

FOR STATEMENTS.

SYNTAX.

The syntax for ⟨for statement⟩ is as follows:

```
1 ⟨for statement⟩ ::= ⟨for clause⟩ ⟨statement⟩ | ⟨label⟩ :
                      ⟨for statement⟩
1 ⟨for clause⟩ ::= FOR ⟨variable⟩ ← ⟨for-list⟩ DO
1 ⟨for-list⟩ ::= ⟨for-list element⟩ | ⟨for-list⟩,
                 ⟨for-list element⟩
3 ⟨for-list element⟩ ::= ⟨arithmetic expression⟩ |
                         ⟨arithmetic expression⟩
                         STEP ⟨arithmetic expression⟩
                         UNTIL ⟨arithmetic expression⟩ |
                         ⟨arithmetic expression⟩ WHILE
                         ⟨Boolean expression⟩ |
                         ⟨arithmetic expression⟩ STEP
                         ⟨arithmetic expression⟩ WHILE
                         ⟨Boolean 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 FOR statement can be best understood by isolating three distinct operational steps:

a. Value assignment to the controlled variable.

b. Test of limiting condition.

c. Execution of the statement following DO.

Each type of for-list describes a different process and will therefore be discussed separately. All, however, have one property in common, i.e., the initial value assigned to the variable of the FOR clause (called the controlled variable) is that of the left-most arithmetic expression in the for-list elements.

THE FOR-LIST. The for-list may contain more than one for-list element. However, for explanatory purposes, it will be assumed that there is only one. In order to expand the meaning of a single for-list element in a for-list to that of multiple for-list elements, one need only consider the following. The process

described by more than one for-list element in a for-list is exactly like that which would be described by writing a series of FOR statements, each with one of the for-list elements, identical controlled variables, and the same statement following each DO.

The for-list element determines what values are to be assigned to the controlled variable and what test to make of the controlled variable in order to decide whether or not to execute the statement following DO. When a for-list element has been exhausted, the next element in the for-list is considered, progressing from left to right. When all the elements in a for-list have been utilized, the for-list is considered exhausted and control is continued in sequence.

ARITHMETIC EXPRESSION ELEMENT. The format for a for-list using an arithmetic expression element is:

FOR V ← AE DO S_{do}; S

A for-list element may be simply an arithmetic expression, in which case only one value is assigned to the controlled variable, V. Since there is no limiting condition, no test is made. After assignment of the initial value to the controlled variable, the statement following DO is executed. The element is then exhausted. A concise description is:

V ← AE;
S_{do};
S

STEP-UNTIL ELEMENT. The format for a for-list using a STEP-UNTIL element is:

FOR V ← AE1 STEP AE2 UNTIL AE3 DO S_{do}; S

This element calls for a new value to be assigned to the controlled variable V each time the statement following DO is

executed. First, an initial value, that of AE1, is assigned to the controlled variable. All subsequent assignments are equivalent to: V ← V + AE2, and are made immediately after the DO statement is executed. The limiting condition on the value of V is given by AE3, which is evaluated anew each time through the loop.

A test is made immediately after each assignment of a value to V to determine whether or not the value of V has passed AE3. Whether AE3 is an upper or lower limit depends upon the sign of AE2; AE3 is an upper limit if AE2 is positive, and is a lower limit if AE2 is negative. If V has not passed AE3, the statement following DO is executed. If V has passed AE3, the element has been exhausted and the statement following DO is not executed. A concise description is:

V ← AE1;

L2: IF AE2 = 0 OR (SIGN(AE2) = +1 AND V $\leq$ AE3) OR (SIGN(AE2) = -1 AND V $\geq$ AE3) THEN BEGIN S_{do}; V ← V + AE2; GO TO L2 END;

S

It can readily be seen that if the value of AE2 is zero, the program will be caught in a closed loop.

WHILE ELEMENT. The format for a for-list using a WHILE element is:

FOR V ← AE WHILE BE DO S_{do}; S

This element causes the value of AE to be assigned to the controlled variable V as long as the logical value of the Boolean expression BE is TRUE. The detailed operation proceeds as follows.

First, the value of AE is assigned to the controlled variable. A test is made on the logical value produced by BE; if the value is TRUE, the statement following DO is executed. This process is

continued until the value of BE is FALSE, at which time the list element has been exhausted and control is transferred to the next statement in the program. A concise description is:

```
L2:  V ← AE;
     IF BE THEN BEGIN S_do; GO TO L2 END;
     S
```

STEP-WHILE ELEMENT. The format for a for-list using a STEP-WHILE element is:

```
FOR V ← AE1 STEP AE2 WHILE BE DO S_do;
S
```

This element calls for a new value to be assigned to the controlled variable V if the value of BE is TRUE each time the statement following DO is executed. First, an initial value, AE1, is assigned to the controlled variable. All subsequent assignments are V ← V + AE2, made immediately after the DO statement is executed. The limiting condition in this case is the logical value produced by BE. A test is made after each assignment to V to determine if the logical value produced by BE is TRUE. If the value of BE is TRUE, the statement following DO is executed; otherwise, control is transferred to the next succeeding statement. This can be stated concisely as:

```
     V ← AE1;
L3:  IF BE THEN BEGIN S_do; V ← V + AE2; GO TO L3 END;
     S
```

VALUE OF CONTROLLED VARIABLE ON EXIT FROM FOR STATEMENT.
Upon exit from the FOR statement, the value of the controlled variable is indeterminate.

RESTRICTION.
A transfer to a labeled statement within the scope of a FOR statement, through the use of a GO TO statement outside the FOR statement, is not allowed.

DO STATEMENTS.

SYNTAX.

The syntax for ⟨do statement⟩ is as follows:

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

Example:

DO SPACE (FILEID, -3) UNTIL A $\geq$ C

SEMANTICS.

The DO statement provides a method of controlling an iterative process in which exit from the loop depends on reaching a limit. The statement is first executed; the test is then made, and the execution of the statement is repeated as long as the Boolean expression is FALSE. A concise description is:

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

WHILE STATEMENTS.

SYNTAX.

The syntax for ⟨while statement⟩ is as follows:

3 ⟨while statement⟩ ::= WHILE ⟨Boolean expression⟩
DO ⟨statement⟩

Example:

WHILE C = A DO SPACE(FILEID, A + B - C)

SEMANTICS.

The WHILE statement provides a method of controlling an iterative process in which exit from the loop depends on exceeding a limit. The Boolean expression is first tested; the following statement is then executed as long as the value of the Boolean expression is TRUE. A concise description is:

LW: IF BE THEN BEGIN S_{while}; GO TO LW END

SECTION 9

DECLARATIONS

GENERAL.

PROCEDURE declarations are covered in Section 10, while Section 11 covers STREAM PROCEDURE declarations.

SYNTAX.

The syntax for ⟨declaration⟩ is as follows:

```
3 ⟨declaration⟩ ::= ⟨type declaration⟩ | ⟨array declaration⟩ |
                    ⟨switch declaration⟩ |
                    ⟨define declaration⟩ |
                    ⟨label declaration⟩ | ⟨file declaration⟩ |
                    ⟨switch file declaration⟩ |
                    ⟨format declaration⟩ |
                    ⟨switch format declaration⟩ |
                    ⟨list declaration⟩ |
                    ⟨switch list declaration⟩ |
                    ⟨forward reference declaration⟩ |
                    ⟨monitor declaration⟩ | ⟨dump declaration⟩ |
                    ⟨procedure declaration⟩ |
                    ⟨stream procedure declaration⟩ |
                    ⟨fault declaration⟩
```

SEMANTICS.

The purpose of a declaration is to define the characteristics of a quantity and assign an identifier to the quantity so that it may be referenced. The scope of a declaration is the block in which it appears. This means that, at the time of entry into a block (through the BEGIN, since the labels inside are local and therefore inaccessible from outside), all identifiers declared in the block head assume the significance implied by their declarations. Conversely, at the time of exit from a block (through an END or a GO TO statement), all identifiers declared in the associated block head lose their applicable significance.

A conflict of significance can arise when blocks are nested, that is, when one block is a statement in the compound tail of another block. This situation occurs when the same identifier is declared in the respective block heads of two or more nested blocks. The conflict is resolved as follows. Assume that block B is nested in block A, and that the identifier CC is declared in both block heads.

In block B, the identifier CC has the significance implied by its declaration in block head B. The quantity declared in block head A and identified by the common identifier CC is inaccessible in block B. This is the only case where an identifier loses its significance prior to exit of the program from the block in which the identifier is declared. When the program exits from block B, the identifier CC again assumes the significance given by the declaration in block head A.

Apart from the identifiers associated with the standard functions (pages 3-8 and 3-9, Standard Functions; page 3-9, Time Functions; pages 3-9 and 3-10, Type Transfer Functions; and page 3-10, Interrogate Function), all identifiers of a program must be declared.

RESTRICTION.

An identifier must not be declared to represent more than one entity in a single block head.

TYPE DECLARATIONS.

SYNTAX.

The syntax for ⟨type declaration⟩ is as follows:

1 ⟨type declaration⟩ ::= ⟨local or own type⟩ ⟨type list⟩
1 ⟨local or own type⟩ ::= ⟨type⟩ | OWN ⟨type⟩
3 ⟨type⟩ ::= REAL | INTEGER | BOOLEAN | ALPHA
1 ⟨type list⟩ ::= ⟨simple variable⟩ | ⟨type list⟩,
⟨simple variable⟩

Examples:

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

SEMANTICS.

A type declaration declares one or more identifiers to represent certain simple variables, and defines the types of values that may be represented by these variables.

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 (positive and negative values, including zero).

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

c. BOOLEAN (logical value of TRUE and FALSE).

d. ALPHA (any set of six (or fewer) characters, not including the illegitimate character ?).

ARRAY DECLARATIONS.

SYNTAX.

The syntax for ⟨array declaration⟩ is as follows:

3 ⟨array declaration⟩ ::= ⟨array kind⟩ ARRAY ⟨array list⟩ | SAVE ⟨array kind⟩ ARRAY ⟨array list⟩

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

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

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

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

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

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

1 ⟨lower bound⟩ ::= ⟨arithmetic expression⟩

1 ⟨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 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. This is necessary only when an array is being used in conjunction with a stream procedure (see Section 11, STREAM PROCEDURE Declarations) in order to maintain the validity of the stream address indexes upon exit from and reentrance to the stream procedure.

LOCAL OR OWN. An array may be declared as OWN with the same effect as that given for simple variables (see page 9-2, 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. Each array must be declared as to type, unless it is of type REAL. An array which is not declared as to type will be considered type REAL (see Restrictions below).

RESTRICTIONS. Arrays which are declared together must be of the same type. If the array is OWN, REAL must also be explicitly declared.

BOUND PAIR LIST. The bound pair list defines the dimensions of the array and the number of elements in each dimension. Bound pairs are formed by expressions (see page 3-3, Evaluation of Subscripts). 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.

If an array is declared OWN, the values of the corresponding subscripted variables are defined only for those variables which have subscripts within the most recently calculated bounds.

RESTRICTIONS.
Arrays declared in the outermost block must use constant bounds. Upper bounds must not be smaller than the corresponding lower bounds. No dimension may contain more than 1023 elements.

SWITCH DECLARATIONS.

SYNTAX.
The syntax for ⟨switch declaration⟩ is as follows:

```
1 ⟨switch declaration⟩ ::= SWITCH ⟨switch identifier⟩ ←
                              ⟨switch list⟩
1 ⟨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 a set of values corresponding to a switch identifier. These values are the designational expressions in the switch list. With each of these designational expressions there is associated a positive integer, 1, 2, ..., obtained by counting the items in the list from left to right. 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 4-14 through 4-16, 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.
SYNTAX.
The syntax for ⟨define declaration⟩ is as follows:

3 ⟨define declaration⟩ ::= DEFINE ⟨definition list⟩
3 ⟨definition list⟩ ::= ⟨definition part⟩ | ⟨definition list⟩, ⟨definition part⟩
3 ⟨definition part⟩ ::= ⟨defined identifier⟩ = ⟨definition⟩ #
3 ⟨defined identifier⟩ ::= ⟨identifier⟩
3 ⟨definition⟩ ::= ⟨well-formed construct⟩
3 ⟨well-formed construct⟩ ::= ⟨basic component set⟩ |

⟨well-formed construct⟩
⟨basic component set⟩

3 ⟨basic component set⟩ ::= ⟨delimiter⟩ | ⟨identifier⟩ |
⟨unsigned number⟩ | ⟨string⟩ |
⟨logical value⟩

Examples:

```
DEFINE RK = RUNGEKUTTA#, ROOT = (-B + SQRT(B *2 - 4 x A x
C))/(2 x A)#

DEFINE INT = INTEGRATE (X, Y, Z)#

DEFINE LP = (#, RP =)#,RTDIG = [42:6]#

DEFINE FORI = FOR I ← 1 STEP 1 UNTIL #
```

SEMANTICS.
The DEFINE declaration provides a method whereby an identifier can be defined to represent a well formed ALGOL construct.

The appearance of a defined identifier in a program is equivalent to the appearance of its definition.

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.

The reserved word COMMENT within a definition will be recognized and everything following it, up to the next semicolon, will be treated as a COMMENT.

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, etc.

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. If a definition ends with the word END, its defined identifier may be followed in the program only be 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	IDENTIFIER SIZE	MAXIMUM
1-5	2035	38-45	1995
6-13	2027	46-53	1987
14-21	2019	54-61	1979
22-29	2011	62-63	1971
30-37	2003		

LABEL DECLARATIONS.

SYNTAX.

The syntax for ⟨label declaration⟩ is as follows:

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

*Blanks are superfluous except in strings or when used as delimiters.

```
3 ⟨label list⟩ ::= ⟨label⟩ | ⟨label list⟩,⟨label⟩
2 ⟨label⟩ ::= ⟨identifier⟩
```

Examples:

LABEL START
LABEL ENTER, EXIT, START, LOOP

SEMANTICS.

As is true of all identifiers, a label must be declared before it is used. A label must be declared in the head of the inner-most block in which the associated labeled statement appears. If any statement in a procedure body is labeled, the declaration of this label must appear within the procedure body.

RESTRICTION.

A procedure body itself may not be labeled.

FILE DECLARATIONS.

SYNTAX.

The syntax for ⟨file declaration⟩ is as follows:

```
3 ⟨file declaration⟩ ::= ⟨file lock part⟩ ⟨mode part⟩ FILE
                          ⟨in-out part⟩ ⟨file identifier⟩
                          ⟨label equation part⟩ (⟨buffer part⟩
                          ⟨save factor⟩)
3 ⟨file lock part⟩ ::= ⟨empty⟩ | SAVE
3 ⟨mode part⟩ ::= ⟨empty⟩ | ALPHA
3 ⟨in-out part⟩ ::= IN | OUT | ⟨empty⟩
3 ⟨file identifier⟩ ::= ⟨identifier⟩
3 ⟨label equation part⟩ ::= ⟨output media part⟩ ⟨disk file
                             description⟩ ⟨label part⟩
3 ⟨output media part⟩ ::= ⟨output media digit⟩ | ⟨empty⟩ | DISK
                           ⟨disk access technique⟩
3 ⟨label part⟩ ::= ⟨file identification part⟩ |
                    ⟨multi-file identification part⟩
                    ⟨file identification part⟩ | ⟨empty⟩
```

3 ⟨disk file description⟩ ::= ⟨empty⟩ | [⟨number of areas⟩ : ⟨size of areas⟩]

3 ⟨number of areas⟩ ::= ⟨arithmetic expression⟩

3 ⟨size of areas⟩ ::= ⟨arithmetic expression⟩

3 ⟨disk access technique⟩ ::= SERIAL | RANDOM | UPDATE | ⟨empty⟩

3 ⟨file identification part⟩ ::= "{7 or less string characters}"

3 ⟨multi-file identification part⟩ ::= "{7 or less string characters}"

3 ⟨buffer part⟩ ::= ⟨number of buffers⟩, ⟨record specifications⟩

3 ⟨number of buffers⟩ ::= ⟨unsigned integer⟩

3 ⟨record specifications⟩ ::= ⟨unblocked specification⟩ | ⟨blocking specifications⟩

3 ⟨unblocked specification⟩ ::= ⟨fixed physical record size⟩

3 ⟨blocking specifications⟩ ::= ⟨fixed logical record size⟩, ⟨fixed physical record size⟩ | ⟨fixed physical record size⟩, ⟨fixed logical record size⟩

3 ⟨fixed logical record size⟩ ::= ⟨arithmetic expression⟩

3 ⟨fixed physical record size⟩ ::= ⟨arithmetic expression⟩

3 ⟨save factor⟩ ::= , SAVE ⟨arithmetic expression⟩ | ⟨empty⟩

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 14 (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, and has effect only if the file is opened by a stream procedure access

or a RELEASE statement. Data communications file declarations are required for input and output. The same data communications file cannot be used both in and out.

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 the 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 <u>must</u> <u>be</u> used on both input and output files referencing the SPO. An output media part of 14 <u>must</u> <u>be</u> 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 an input data communications file must be large enough to accommodate the largest hardware terminal buffer plus one word for the status word. The hardware terminal buffers can only be multiples of 28 characters, with a maximum of 448 characters.

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⟩ unless a Data Communications Read Seek statement is performed on the file.

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, while 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:

3 ⟨switch file declaration⟩ ::= SWITCH FILE ⟨switch file identifier⟩ ← ⟨switch file list⟩

3 ⟨switch file identifier⟩ ::= ⟨identifier⟩

3 ⟨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:

```
3 ⟨format declaration⟩ ::= FORMAT ⟨input or output⟩ ⟨format
                           part⟩
3 ⟨input or output⟩ ::= IN | OUT | ⟨empty⟩
3 ⟨format part⟩ ::= ⟨format identifier⟩ (⟨editing specifications⟩)
                    | ⟨format part⟩, ⟨format identifier⟩
                    (⟨editing specifications⟩)
3 ⟨format identifier⟩ ::= ⟨identifier⟩
3 ⟨editing specifications⟩ ::= ⟨editing segment⟩ |
                               ⟨editing specifications⟩ / |
                               / ⟨editing specifications⟩ |
                               ⟨editing specifications⟩ /
                               ⟨editing segment⟩
3 ⟨editing segment⟩ ::= ⟨editing phrase⟩ | ⟨repeat part⟩
                        (⟨editing specifications⟩) |
                        ⟨editing segment⟩, ⟨editing phrase⟩ |
                        ⟨editing segment⟩, ⟨repeat part⟩
                        (⟨editing specifications⟩)
3 ⟨editing phrase⟩ ::= ⟨repeat part⟩ ⟨editing phrase type⟩
                       ⟨field part⟩ | ⟨string⟩
3 ⟨repeat part⟩ ::= ⟨empty⟩ | ⟨unsigned integer⟩ | *
3 ⟨editing phrase type⟩ ::= A | D | E | F | I | L | O | R |
                            S | V | X
3 ⟨field part⟩ ::= ⟨empty⟩ | ⟨field width⟩ | ⟨field width⟩ .
                   ⟨decimal places⟩
3 ⟨field width⟩ ::= ⟨unsigned integer⟩ | *
3 ⟨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.

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.

---

1. The last character before the right parenthesis is the letter O, not zero.

Characteristics of the input editing phrase types are summarized in table 9-1.

Table 9-1

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

The definition of each input editing phrase type 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.

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 7 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:

±nn---n.	±nn---n.dd-d	nn---n.dd---d
nn---n.	±.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 9-2.

Table 9-2
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 0, the editing phrase will be skipped. If the repeat part pre-

ceding 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.

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 6-21.

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 9-3.

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:

bn.dd---d@±ee

Table 9-3

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)) xR (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 ($\underset{-}{b}$ = 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:

b̲nn---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 9-4 shows the effect of various values of field width.

Table 9-4

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:

3 ⟨switch format declaration⟩ ::= SWITCH FORMAT ⟨switch format identifier⟩ ← ⟨switch format list⟩

3 ⟨switch format identifier⟩ ::= ⟨identifier⟩

3 ⟨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:

3 ⟨list declaration⟩ ::= LIST ⟨list part⟩

3 ⟨list part⟩ ::= ⟨list identifier⟩ (⟨list⟩) | ⟨list part⟩, ⟨list identifier⟩ (⟨list⟩)

3 ⟨list identifier⟩ ::= ⟨identifier⟩

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

3 ⟨list segment⟩ ::= ⟨expression part⟩ | ⟨for clause⟩⟨list segment⟩ | ⟨for clause⟩[⟨expression list⟩]

3 ⟨expression part⟩ ::= ⟨arithmetic expression⟩ | ⟨Boolean expression⟩

3 ⟨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 6-18 through 6-21) for specifying the variables to be initialized and the order in which the initial-

izing 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 6-27 through 6-29) 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:

3 ⟨switch list declaration⟩ ::= SWITCH LIST ⟨switch list identifier⟩ ← ⟨switch list list⟩

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

3 ⟨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 by 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.

FORWARD REFERENCE DECLARATIONS.

SYNTAX.

The syntax for ⟨forward reference declaration⟩ is as follows:

3 ⟨forward reference declaration⟩ ::= ⟨forward procedure declaration⟩ | ⟨forward switch declaration⟩

3 ⟨forward procedure declaration⟩ ::= ⟨procedure type⟩ PROCEDURE ⟨procedure heading⟩ FORWARD

3 ⟨procedure type⟩ ::= ⟨empty⟩ | ⟨type⟩

3 ⟨forward switch declaration⟩ ::= SWITCH ⟨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 or 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.

MONITOR DECLARATIONS.

SYNTAX.

The syntax for ⟨monitor declaration⟩ is as follows:

3 ⟨monitor declaration⟩ ::= MONITOR ⟨monitor part⟩

3 ⟨monitor part⟩ ::= ⟨file identifier⟩ (⟨monitor list⟩) |
⟨monitor part⟩, ⟨file identifier⟩
(⟨monitor list⟩)

3 ⟨monitor list⟩ ::= ⟨monitor list element⟩ |
⟨monitor list⟩, ⟨monitor list element⟩

3 ⟨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:

3 ⟨dump declaration⟩ ::= DUMP ⟨dump part⟩

3 ⟨dump part⟩ ::= ⟨file identifier⟩ (⟨dump list⟩) ⟨label⟩:⟨dump indicator⟩ | ⟨dump part⟩, ⟨file identifier⟩ (⟨dump list⟩) ⟨label⟩: ⟨dump indicator⟩

3 ⟨dump list⟩ ::= ⟨dump list element⟩ | ⟨dump list⟩, ⟨dump list element⟩

3 ⟨dump list element⟩ ::= ⟨simple variable⟩ | ⟨subscripted variable⟩ | ⟨label⟩ | ⟨array identifier⟩

3 ⟨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. Diagnostic 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:

3 ⟨fault declaration⟩ ::= MONITOR ⟨fault list⟩

3 ⟨fault list⟩ ::= ⟨fault type⟩ | ⟨fault list⟩, ⟨fault type⟩ |
⟨fault list⟩, ⟨fault equate⟩

3 ⟨fault type⟩ ::= EXPOVR|INTOVR|INDEX|FLAG|ZERO

3 ⟨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 6-35.

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 10

PROCEDURE DECLARATIONS

GENERAL

SYNTAX.

The syntax for ⟨procedure declaration⟩ is as follows:

1 ⟨procedure declaration⟩ ::= PROCEDURE ⟨procedure heading⟩ ⟨procedure body⟩ | ⟨type⟩ PROCEDURE ⟨procedure heading⟩ ⟨procedure body⟩

1 ⟨procedure heading⟩ ::= ⟨procedure identifier⟩ ⟨formal parameter part⟩; ⟨value part⟩ ⟨specification part⟩

1 ⟨procedure identifier⟩ ::= ⟨identifier⟩

1 ⟨formal parameter part⟩ ::= ⟨empty⟩ | (⟨formal parameter list⟩)

1 ⟨formal parameter list⟩ ::= ⟨formal parameter⟩ | ⟨formal parameter list⟩ ⟨parameter delimiter⟩ ⟨formal parameter⟩

1 ⟨formal parameter⟩ ::= ⟨identifier⟩

1 ⟨value part⟩ ::= VALUE ⟨identifier list⟩; | ⟨empty⟩

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

2 ⟨specification part⟩ ::= ⟨empty⟩ | ⟨specification list⟩

3 ⟨specification list⟩ ::= ⟨specification⟩; | ⟨specification list⟩ ⟨specification⟩;

3 ⟨specification⟩ ::= ⟨specifier⟩ ⟨identifier list⟩ | ⟨array specification⟩

2 ⟨specifier⟩ ::= LABEL | ⟨type⟩ | SWITCH | PROCEDURE | ⟨type⟩ PROCEDURE | FILE | LIST | FORMAT | SWITCH FILE | SWITCH FORMAT | SWITCH LIST

3 ⟨array specification⟩ ::= ARRAY ⟨array specifier list⟩ |
⟨type⟩ ARRAY ⟨array specifier list⟩

3 ⟨array specifier list⟩ ::= ⟨array specifier⟩ |
⟨array specifier list⟩, ⟨array specifier⟩

3 ⟨array specifier⟩ ::= ⟨array identifier list⟩ [⟨lower bound list⟩]

3 ⟨array identifier list⟩ ::= ⟨identifier list⟩

3 ⟨lower bound list⟩ ::= ⟨specified lower bound⟩ | ⟨lower bound list⟩, ⟨specified lower bound⟩

3 ⟨specified lower bound⟩ ::= ⟨integer⟩ | *

2 ⟨procedure body⟩ ::= ⟨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 declares an identifier to represent a procedure, and defines what this procedure shall be. Whenever

the identifier followed by the appropriate parameters appears in the program, it produces a call upon the procedure (see pages 6-10 through 6-15, Procedure Statements).

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 the 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 values 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 nonlocal to the body, depending on whether they are declared within the body or outside the body. Those which are nonlocal to the body may be local to the block which contains the PROCEDURE declaration in its head.

Any quantity that is nonlocal 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.

SECTION 11

STREAM PROCEDURE DECLARATIONS

GENERAL.

SYNTAX.

The syntax for ⟨stream procedure declaration⟩ is as follows:

```
3 ⟨stream procedure declaration⟩ ::= STREAM PROCEDURE
                                      ⟨stream procedure heading⟩
                                      ⟨stream block⟩ |
                                      ⟨type⟩ STREAM PROCEDURE
                                      ⟨stream procedure heading⟩
                                      ⟨stream block⟩
3 ⟨stream procedure heading⟩ ::= ⟨procedure identifier⟩
                                  ⟨stream formal parameter part⟩;
                                  ⟨value part⟩
3 ⟨stream formal parameter part⟩ ::= (⟨formal parameter list⟩)
3 ⟨stream block⟩ ::= ⟨stream block head⟩;⟨compound stream tail⟩
3 ⟨stream block head⟩ ::= BEGIN ⟨stream declaration⟩ |
                          ⟨stream block head⟩;⟨stream declaration⟩
3 ⟨compound stream tail⟩ ::= ⟨stream statement⟩ END |
                             ⟨stream statement⟩;⟨compound
                             stream tail⟩
3 ⟨stream declaration⟩ ::= ⟨stream variable declaration⟩ |
                           ⟨label declaration⟩
3 ⟨stream variable declaration⟩ ::= LOCAL ⟨stream variable list⟩
                                    | ⟨empty⟩
3 ⟨stream variable list⟩ ::= ⟨stream simple variable⟩ |
                             ⟨stream variable list⟩,
                             ⟨stream simple variable⟩
3 ⟨stream simple variable⟩ ::= ⟨variable identifier⟩
```

Example:

```
STREAM PROCEDURE MOVE (SOURCE, DESTINATION, DIV32, MOD32);
VALUE DIV32, MOD32;
```

```
BEGIN
  COMMENT THIS PROCEDURE WILL MOVE N WORDS FROM A FILE TO A
  TWO-DIMENSIONAL ARRAY OR VICE VERSA;
  LOCAL SOURCEDESC, DESTINATIONDESC;
  SI ← SOURCE; DI ← LOC SOURCEDESC; DS ← WDS;
  SI ← DESTINATION; DI ← LOC DESTINATIONDESC; DS ← WDS;
  SI ← SOURCEDESC; DI ← DESTINATIONDESC;
  DIV32 (DS ← 32 WDS); DS ← MOD32 WDS;
END MOVE
```

SEMANTICS.

The STREAM PROCEDURE declaration defines an identifier which represents a special kind of procedure, the stream procedure. The stream procedure is designed exclusively for the manipulation of words, characters, and bits. For this reason, the language used to describe a stream procedure differs from that of conventional procedures.

Some of the problems to which a stream procedure can be applied are those involving complex editing of information on input and output operations, packing and unpacking of data for more efficient information storage, and scanning operations for comparison of data. These are but a few of the many applications in which stream procedures can be of significant value to the programmer.

FORMAL PARAMETERS AND VALUE PART. All formal parameters of a stream procedure are treated as local to the stream block. The corresponding actual parameters provide initial values for the formal parameters as indicated by the VALUE part.

The formal parameters listed in the VALUE part (call by value) are assigned the values of the corresponding actual parameters when the stream procedure is called. The formal parameters not listed in the VALUE part (call by name) are assigned the absolute addresses of the corresponding actual parameters.

STREAM DECLARATIONS. All stream simple variables in a stream block must be declared by a stream variable declaration (LOCAL).

All stream simple variables are therefore local to the stream block. All labels in a stream block must be listed in a LABEL declaration.

COMPOUND STREAM TAIL. The stream block includes, in addition to the variable and LABEL declarations, a stream statement or a series of stream statements. Before describing the stream statements individually, it is necessary to clarify certain concepts applicable to every statement in the STREAM PROCEDURE declaration.

The basic delimiters used in stream procedures are SI and DI. SI (source index) denotes the core address from which information is to be taken. DI (destination index) denotes the core address to which information is to be moved.

As has been stated, stream procedures manipulate not only words, but individual characters or bits as well. Hence, for the sake of brevity as well as clarity, the following notation has been adopted for discussing the various stream statements:

a. SI_w - word address portion of source index.

b. DI_w - word address portion of destination index.

c. SI_c - character designator portion of source index; SI_c = 0 for left-most character of word, 7 for right-most character.

d. DI_c - character designator portion of destination index; DI_c = 0 for left-most character of word, 7 for right-most character.

e. SI_b - bit designator portion of source index; SI_b = 0 for left-most bit of character, 5 for right-most bit.

f. DI_b - bit designator portion of destination index; DI_b = 0 for left-most bit of character, 5 for right-most bit.

g. CI_w - word address portion of control index.

h. CI_s - syllable designator portion of control index; $CI_s = 0$ for left-most syllable of word, 3 for right-most syllable.

i. ri - repetitive indicator.

AUTOMATIC INDEX ADJUSTMENT. Before certain stream statements are executed, either the source index, the destination index, or both may be automatically adjusted. These adjustments are conditional and fall into two categories. The controlling conditions and the adjustments made are outlined below and are referenced throughout the succeeding discussion whenever applicable.

a. Adjustment Category I.

1) Source index.

If $SI_b \neq 0$ or $SI_c \neq 0$, then $SI_w \leftarrow SI_w + 1$; $SI_b \leftarrow SI_c \leftarrow 0$.

If $SI_b = 0$ and $SI_c = 0$, then no adjustment is made.

2) Destination index.

If $DI_b \neq 0$ or $DI_c \neq 0$, then $DI_w \leftarrow DI_w + 1$; $DI_b \leftarrow DI_c \leftarrow 0$.

If $DI_b = 0$ and $DI_c = 0$, then no adjustment is made.

b. Adjustment Category II.

1) Source index.

If $SI_b \neq 0$, then $SI_b \leftarrow 0$; $SI_c \leftarrow SI_c + 1$ (overflow into SI_w may occur).

If $SI_b = 0$, then no adjustment is made.

2) Destination index.

If $DI_b \neq 0$, then $DI_b \leftarrow 0$; $DI_c \leftarrow DI_c + 1$ (overflow into DI_w may occur).

If $DI_b = 0$, then no adjustment is made.

STREAM STATEMENTS.

SYNTAX.

The syntax for ⟨stream statement⟩ is as follows:

3 ⟨stream statement⟩ ::= ⟨unlabeled stream statement⟩ |
⟨label⟩:⟨stream statement⟩

3 ⟨unlabeled stream statement⟩ ::= ⟨unconditional stream statement⟩ |
⟨conditional stream statement⟩

SEMANTICS.

Stream statements are unique to STREAM PROCEDURE declarations and may not be used outside such declarations. Stream statements and their uses are discussed in the following paragraphs.

UNCONDITIONAL STREAM STATEMENTS.

SYNTAX.

The syntax for ⟨unconditional stream statement⟩ is as follows:

3 ⟨unconditional stream statement⟩ ::= ⟨stream address statement⟩
| ⟨destination string statement⟩ |
⟨stream go to statement⟩ |
⟨skip bit statement⟩ |
⟨stream tally statement⟩ |
⟨stream nest statement⟩ |
⟨stream release statement⟩
| ⟨compound stream statement⟩ |
⟨stream dummy statement⟩

3 ⟨stream address statement⟩ ::= ⟨set address statement⟩ |
⟨store address statement⟩ |
⟨skip address statement⟩ |
⟨recall address statement⟩

SEMANTICS.

The various types of unconditional stream address statements are described individually in the following paragraphs.

SET ADDRESS STATEMENTS.

SYNTAX.

The syntax for ⟨set address statement⟩ is as follows:

3 ⟨set address statement⟩ ::= SI ← ⟨source address part⟩ |
DI ← ⟨destination address part⟩
3 ⟨source address part⟩ ::= LOC ⟨stream simple variable⟩ | SC
3 ⟨destination address part⟩ ::= LOC ⟨stream simple variable⟩ | DC

Examples:

SI ← SC
DI ← LOC Q1

SEMANTICS.

The set address statement using the delimiter LOC causes either the source or destination index to be set to the core location of the indicated stream variable.

The set address statement using the delimiter SC or DC (see b, page 11-4) assigns the value contained in the next 18 bits of the applicable string to the source or destination index.

STORE ADDRESS STATEMENTS.

SYNTAX.

The syntax for ⟨store address statement⟩ is as follows:

3 ⟨store address statement⟩ ::= ⟨stream simple variable⟩ ←
⟨stream address index⟩
3 ⟨stream address index⟩ ::= SI | DI | CI

Examples:

T2 ← DI
T3 ← CI

SEMANTICS.
The store address statement causes the current value of the indicated index to be assigned to the indicated stream variable.

The CI (Control Index) register contains the core address of the program word and the next program syllable to be executed.

SKIP ADDRESS STATEMENTS.

SYNTAX.
The syntax for ⟨skip address statement⟩ is as follows:

3 ⟨skip address statement⟩ ::= DI ← DI ⟨stream arithmetic expression⟩ | SI ← SI ⟨stream arithmetic expression⟩
3 ⟨stream arithmetic expression⟩ ::= ⟨adding operator⟩ ⟨stream primary⟩
1 ⟨adding operator⟩ ::= + | -
3 ⟨stream primary⟩ ::= ⟨unsigned integer⟩ | ⟨stream simple variable⟩

Examples:

SI ← SI + 3
DI ← DI - T4

SEMANTICS.
The skip address statement causes SI_c or DI_c to be increased or decreased by the value of the stream primary.

RESTRICTION.
The source index (SI) and the destination index (DI) must never point to the same location, that is, SI_w must never equal DI_w.

RECALL ADDRESS STATEMENTS.

SYNTAX.

The syntax for ⟨recall address statement⟩ is as follows:

```
3 ⟨recall address statement⟩ ::= ⟨stream address index⟩ ←
                                 ⟨stream simple variable⟩
```

Examples:

SI ← SOURCE

DI ← T2

SEMANTICS.

The recall address statement causes the value of a stream variable to be assigned to the indicated index.

DESTINATION STRING STATEMENTS.

SYNTAX.

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

```
3 ⟨destination string statement⟩ ::= DS ← ⟨transfer part⟩
3 ⟨transfer part⟩ ::= ⟨source string transfer⟩ | ⟨literal trans-
                      fer⟩ | ⟨blank replacement transfer⟩
3 ⟨source string transfer⟩ ::= ⟨repetitive indicator⟩
                               ⟨transfer type⟩
3 ⟨repetitive indicator⟩ ::= ⟨stream repeat part⟩ |
                             ⟨stream simple variable⟩
3 ⟨stream repeat part⟩ ::= ⟨empty⟩ | ⟨unsigned integer⟩
3 ⟨transfer type⟩ ::= ⟨transfer words⟩ | ⟨transfer characters⟩ |
                      ⟨transfer and convert⟩ | ⟨transfer and add⟩
                      | ⟨transfer character portions⟩
3 ⟨transfer words⟩ ::= WDS
3 ⟨transfer characters⟩ ::= CHR
3 ⟨transfer and convert⟩ ::= ⟨input convert⟩ | ⟨output convert⟩
3 ⟨input convert⟩ ::= OCT
3 ⟨output convert⟩ ::= DEC
3 ⟨transfer and add⟩ ::= ADD | SUB
3 ⟨transfer character portions⟩ ::= ZON | NUM
```

3 ⟨literal transfer⟩ ::= ⟨literal characters⟩ | ⟨literal bits⟩

3 ⟨literal characters⟩ ::= ⟨unsigned integer⟩ LIT ⟨string⟩

3 ⟨literal bits⟩ ::= ⟨repetitive indicator⟩ SET | ⟨repetitive indicator⟩ RESET

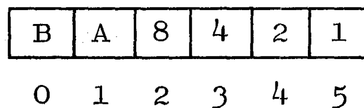
3 ⟨blank replacement transfer⟩ ::= ⟨repetitive indicator⟩ FILL

Examples:

Transfer words:	DS ← 6 WDS
Transfer characters:	DS ← 5 CHR
Input convert:	DS ← 6 OCT
Output convert:	DS ← 5 DEC
Transfer and add:	DS ← 3 ADD
	DS ← 2 SUB
Transfer zone bits:	DS ← VARY ZON
Transfer numeric bits:	DS ← 4 NUM
Literal transfer:	DS ← 7 LIT "HEADING"
Literal bits:	DS ← X SET
	DS ← Y RESET
Blank replacement:	DS ← 8 FILL

SEMANTICS.

To be able to use the bit manipulating possibilities of the destination string statements, it is necessary to know that, within a character, the bit positions are designated as shown below:



The B and A bits are referred to as the zone bits of the character. The 8, 4, 2, and 1 bits are referred to as the numeric part of the character. It is possible to operate independently on either the zone or the numeric part of a character.

TOGGLE is the name of a TRUE/FALSE indicator which can be set and reset by various stream procedure operations.

TRANSFER WORDS. The transfer words option (see a, page 11-4) causes the number of words specified by the repetitive indicator to be transferred from the source string to the destination string. The execution of this statement affects SI and DI as follows:

$SI_w \leftarrow SI_w + ri$

$DI_w \leftarrow DI_w + ri$

TRANSFER CHARACTERS. The transfer characters option (see b, page 11-4) causes the number of characters specified by the repetitive indicator to be transferred from the source string to the destination string. The execution of this statement affects SI and DI as follows:

$SI_c \leftarrow SI_c + ri$ (overflow into SI_w can occur)

$DI_c \leftarrow DI_c + ri$ (overflow into DI_w can occur)

INPUT CONVERT. The input convert option (page 11-4) causes the number of source characters (numeric bits only) specified by the repetitive indicator to be transferred and converted to one octal word in the destination string. The resulting octal word is an integer. The sign of the integer is determined by the zone bits (B,A) of the right-most character in the source field (1,0 = minus; any other combination = plus). The execution of this statement affects SI and DI as follows:

$SI_c \leftarrow SI_c + ri$ (overflow into SI_w can occur)

$DI_w \leftarrow DI_w + 1$

RESTRICTION. The value of the repetitive indicator must not be greater than 8.

OUTPUT CONVERT. The output convert option (page 11-4) causes one octal word in the source string to be transferred and converted to the number of decimal destination characters specified by the repetitive indicator. The octal word is treated as an integer. The sign is placed in the zone bits (B,A) of the right-most destination character (1,0 = minus; any other combination = plus). All other destination zone bits are set to ZERO.

If the converted value requires more than the specified number of destination characters, the most-significant digits are lost and TOGGLE is set to FALSE; otherwise, TOGGLE is set to TRUE. The execution of this statement affects SI and DI as follows:

$SI_w \leftarrow SI_w + 1$

$DI_c \leftarrow DI_c + ri$ (overflow into DI_w can occur)

RESTRICTION. The value of the repetitive indicator must not be greater than 8.

TRANSFER AND ADD. The transfer and add option (see page 11-4) causes the number of source characters specified by the repetitive indicator to be algebraically added to or subtracted from a like number of destination characters. The signs of the two fields are the zone bits (B,A) of their respective right-most characters (1,0 = minus; any other combination = plus). All other source zone bits are ignored and all other destination zone bits are set to zero. The sign of the result is placed in the zone bits of the right-most destination character. If overflow occurs in the destination field, TOGGLE is set to TRUE; otherwise, it is set to FALSE. The execution of this statement affects SI and DI as follows:

$SI_c \leftarrow SI_c + ri$ (overflow into SI_w can occur)

$DI_c \leftarrow DI_c + ri$ (overflow into DI_w can occur)

TRANSFER CHARACTER PORTIONS. The transfer character portions option (see page 11-4) causes either the zone bits or the numeric bits of the number of source characters specified by the repetitive indicator to be transferred to the same portions of a like number of destination characters.

When the transfer zone bits option is used, the numeric portions of the destination characters are not affected. When numeric bits only are transferred, however, the zone portions of the destination characters are set to zero. TOGGLE is set only when numeric bits alone are transferred as follows: If the zone bits (B,A) of the

right-most source character are 1,0 (minus), TOGGLE is set to TRUE; otherwise, it is set to FALSE. The execution of this statement affects SI and DI as follows:

$SI_c \leftarrow SI_c$ + ri (overflow into SI_w can occur)

$DI_c \leftarrow DI_c$ + ri (overflow into DI_w can occur)

LITERAL CHARACTERS. The literal characters option causes the number of string characters specified by the unsigned integer to be placed in the destination string. The unsigned integer should equal the number of characters in the string. If it is greater than the number of string characters, repetitive left-to-right use is made of the string characters until the designated number of destination characters are filled. If it is less, the right-most string characters are ignored. The execution of this statement affects DI only, as follows:

$DI_c \leftarrow DI_c$ + unsigned integer (overflow into DI_w can occur)

LITERAL BITS. "Literal bits" causes the number of destination bits specified by the repetitive indicator to be set to ONE or reset to ZERO. The execution of this statement affects DI only, as follows:

$DI_b \leftarrow DI_b$ + ri (overflow into DI_c can occur, as well as overflow into DI_w)

REPETITIVE INDICATOR. The value of the repetitive indicator must never exceed 63.

BLANK REPLACEMENT. The blank replacement transfer only affects the destination string and the destination string address. The destination string address is adjusted so that the field begins at a character boundary. Each character of the destination string is examined and if it is equal to or less than zero (≤ 0) in the collating sequence, that character is replaced with the blank character code. Replacement stops if a character is equal to or greater than one.

STREAM GO TO STATEMENTS.

SYNTAX

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

3 ⟨stream go to statement⟩ ::= GO TO ⟨label⟩

Example:

GO TO START

SEMANTICS.

The stream GO TO statement causes transfer of control to the statement with the designated label. The label must be one declared in the stream block.

RESTRICTION.

A stream GO TO statement must not cause transfer into or out of a stream nest statement.

SKIP BIT STATEMENTS.

SYNTAX.

The syntax for ⟨skip bit statement⟩ is as follows:

3 ⟨skip bit statement⟩ ::= SKIP ⟨repetitive indicator⟩ ⟨source or destination bit⟩

3 ⟨source or destination bit⟩ ::= SB | DB

Examples:

SKIP N SB

SKIP 12 DB

SEMANTICS.

The SKIP bit statement affects only SI or DI, and does so as follows:

$SI_b \leftarrow SI_b + ri$ (overflow into SI_c can occur, as well as overflow into SI_w)

$DI_b \leftarrow DI_b + ri$ (overflow into DI_c can occur, as well as overflow into DI_w)

STREAM TALLY STATEMENTS.

SYNTAX.

The syntax for ⟨stream tally statement⟩ is as follows:

```
3 ⟨stream tally statement⟩ ::= TALLY ← ⟨stream primary⟩ |
                               TALLY ← TALLY + ⟨stream primary⟩ |
                               ⟨stream simple variable⟩ ← TALLY
```

Examples:

```
TALLY ← ABLE
TALLY ← TALLY + 1
TALLY ← TALLY + BETA
GAMMA ← TALLY
```

SEMANTICS.

The stream TALLY statement provides a counting mechanism for stream procedures. TALLY may contain values ranging from 0 to 63. The counter may be stepped by adding an integer to its current value. All overflows are lost. To reset or decrement TALLY, the program must increment it to or beyond the overflow point.

STREAM NEST STATEMENTS.

SYNTAX.

The syntax for ⟨stream nest statement⟩ is as follows:

```
3 ⟨stream nest statement⟩ ::= ⟨repetitive indicator⟩
                              (⟨compound nest⟩)
3 ⟨compound nest⟩ ::= ⟨nest⟩ | ⟨nest⟩;⟨compound nest⟩
3 ⟨nest⟩ ::= ⟨stream statement⟩ | ⟨jump out statement⟩ |
             ⟨label⟩ : ⟨jump out statement⟩
3 ⟨jump out statement⟩ ::= JUMP OUT | JUMP OUT ⟨number of nests⟩
                           TO ⟨label⟩
3 ⟨number of nests⟩ ::= ⟨empty⟩ | ⟨unsigned integer⟩
```

Examples:

```
25 (IF SC = "E" THEN JUMP OUT; SI ← SI + 1; TALLY ← TALLY + 1)
```

```
30 (IF 8 SC = DC THEN 8 (IF SC = ALPHA THEN JUMP OUT 2 TO
 L2; SI ← SI + 1); TALLY ← TALLY + 1); L2: S
```

SEMANTICS.

The stream nest statement serves as a repetitive control statement by means of which loops can be described and the number of passes specified by the repetitive indicator. Any stream statement may appear in a compound nest.

An additional statement, the JUMP OUT statement, is allowed only in a compound nest. The simple form of JUMP OUT statement transfers control to the statement immediately beyond the next right parenthesis. The JUMP OUT to a label form may be used to escape from as many nests as desired and to a specific labeled statement. The JUMP OUT statement itself may be labeled. The number of nests (right parentheses) over which a JUMP OUT is to be effective must be given as an integer. If the integer is 1, it may be omitted.

RESTRICTIONS.

A stream nest statement may be entered only at its beginning. The JUMP OUT statement must not be used in any construct other than a stream nest statement.

STREAM RELEASE STATEMENTS.

SYNTAX.

The syntax for ⟨stream release statement⟩ is as follows:

3 ⟨stream release statement⟩ ::= RELEASE (⟨formal parameter⟩)

Example:

```
RELEASE (FILENAME1)
```

SEMANTICS.

The actual parameter corresponding to the formal parameter of a stream RELEASE statement must be a file identifier. If the identifier is that of an input file, the stream RELEASE statement causes one buffer of the file to be filled with new data. If the identifier is that of an output file, the stream RELEASE statement

causes the contents of one output buffer to be transferred to the appropriate output device.

Both SI and DI must be reset since the values of both are lost with a stream RELEASE statement.

RESTRICTION.

The formal parameter of a stream RELEASE statement must not be called by value.

COMPOUND STREAM STATEMENTS.

SYNTAX.

The syntax for ⟨compound stream statement⟩ is as follows:

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

Example:

```
BEGIN SI ← LOC Q1; T2 ← DI; DI ← T1 END
```

SEMANTICS.

The compound stream statement is a set of stream statements bounded by BEGIN and END.

STREAM DUMMY STATEMENTS.

SYNTAX.

The syntax for ⟨stream dummy statement⟩ is as follows:

3 ⟨stream dummy statement⟩ ::= ⟨empty⟩

Examples:

```
BOTTOM:
FINI:
```

SEMANTICS.

A dummy statement executes no operation. It may serve to place a label.

CONDITIONAL STREAM STATEMENTS.

SYNTAX.

The syntax for ⟨conditional stream statement⟩ is as follows:

3 ⟨conditional stream statement⟩ ::= ⟨stream if clause⟩ ⟨unconditional stream statement⟩ | ⟨stream if clause⟩ ⟨label⟩ : ⟨unconditional stream statement⟩ | ⟨conditional stream statement⟩ ELSE ⟨stream statement⟩

3 ⟨stream if clause⟩ ::= IF ⟨test⟩ THEN

3 ⟨test⟩ ::= ⟨source with literal⟩ | ⟨source with destination⟩ | ⟨source bit⟩ | TOGGLE | ⟨source for alpha⟩

3 ⟨source with literal⟩ ::= SC ⟨relational operator⟩ "⟨string character⟩" | SC ⟨relational operator⟩ "⟨string bracket character⟩"

3 ⟨source with destination⟩ ::= ⟨repetitive indicator⟩ SC ⟨relational operator⟩ DC

3 ⟨source bit⟩ ::= SB

3 ⟨source for alpha⟩ ::= SC = ALPHA

Examples:

Conditional Stream Statements:

Stream IF clause: IF SC = "E" THEN GO TO CONTINUE

Stream IF clause with labeled statement: IF SC > "E" THEN REPLACE: DS ← 5 LIT "FALSE"

Conditional Stream Tests:

Source with literal: IF SC = "E" THEN GO TO CONTINUE

Conditional Stream Tests (cont):

Source with destination:	IF 8 SC < DC THEN GO TO TUSCON
Source bit:	IF SB THEN SI ← SI + 1
Toggle:	IF TOGGLE THEN DS ← X ZON
Source for alpha:	IF SC = ALPHA THEN SI ← SI + 1

SEMANTICS.
The conditional stream statement causes the stream statement following the IF clause to be executed if the test is TRUE; otherwise, the statement is ignored. The execution of every conditional stream statement sets TOGGLE to TRUE or FALSE according to the result of the test. One exception to this is that a test of TOGGLE does not change the TOGGLE value.

SOURCE WITH LITERAL. The source with literal option (see b, page 11-4, for SI only) causes one source character to be compared with the character indicated in the test.

SOURCE WITH DESTINATION. The source with destination option (see b, page 11-4) compares a specified number of source characters with the same number of destination characters. The execution of this statement affects SI and DI as follows:

$SI_c \leftarrow SI_c + ri$ (overflow into SI_w can occur)

$DI_c \leftarrow DI_c + ri$ (overflow into DI_w can occur)

SOURCE BIT. This test causes one source bit to be tested for 1.

TOGGLE. This test is merely one for the value of TOGGLE. As mentioned above, it causes no change in the value of TOGGLE.

SOURCE FOR ALPHA. The source for alpha option (see b, 11-4, for SI only) tests one source character for equal condition only. A syntax error will result if the test is made for unequal condition. If the source character is a letter or a digit, TOGGLE is set to TRUE; otherwise, TOGGLE is set to FALSE.

SECTION 12

SORT STATEMENT AND MERGE STATEMENT

SORT STATEMENT.

SYNTAX.

The syntax for ⟨sort statement⟩ is as follows:

```
3 ⟨sort statement⟩ ::= SORT (⟨output option⟩, ⟨input option⟩,
                         ⟨number of tapes⟩, ⟨hivalue procedure⟩,
                         ⟨compare procedure⟩, ⟨record length⟩
                          ⟨size specifications⟩)
3 ⟨size specifications⟩ ::= ⟨empty⟩ | ⟨core size⟩ | ⟨core size⟩
                             ⟨disk size⟩
3 ⟨core size⟩ ::= , ⟨arithmetic expression⟩
3 ⟨disk size⟩ ::= , ⟨arithmetic expression⟩
3 ⟨record length⟩ ::= ⟨arithmetic expression⟩
3 ⟨compare procedure⟩ ::= ⟨identifier⟩
3 ⟨hivalue procedure⟩ ::= ⟨identifier⟩
3 ⟨number of tapes⟩ ::= ⟨arithmetic expression⟩
3 ⟨input option⟩ ::= ⟨file part⟩ | ⟨input procedure⟩
3 ⟨input procedure⟩ ::= ⟨identifier⟩
3 ⟨output option⟩ ::= ⟨file part⟩ | ⟨output procedure⟩
3 ⟨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 modules).

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. An example of a hivalue procedure that could be used by the compare procedure on page 12-2 follows:

```
PROCEDURE HV (A);
ARRAY A[0];
FILL A[*] WITH OCT7777777777777777;
```

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:

```
 3 ⟨merge statement⟩ ::= MERGE (⟨output option⟩, ⟨hivalue
                         procedure⟩, ⟨compare procedure⟩,
                         ⟨record length⟩, ⟨merge file list⟩)
 3 ⟨merge file list⟩ ::= ⟨merge file⟩, ⟨merge file⟩ |
                         ⟨merge file⟩, ⟨merge file list⟩
 3 ⟨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.

APPENDIX A

RESERVED WORDS

Some reserved words in Extended ALGOL may be used as identifiers in certain constructs. Hence, the following list of reserved words is divided into four types as follows:

Type 1 - reserved throughout Extended ALGOL.

Type 2 - reserved in Stream Procedures only.

Type 3 - 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 4 - 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

ADD	DS	RESET	TALLY
CHR	JUMP	SB	TOGGLE
CI	LIT	SC	WDS
DB	LOC	SET	ZON
DC	LOCAL	SI	
DEC	NUM	SKIP	
DI	OCT	SUB	

Type 3

ABS	ENTIER	SIGN	STATUS
ARCTAN	EXP	SIN	TIME
COS	LN	SQRT	

Type 4

BREAK	INTOVR	PUNCH	TIMES
DBL	LB	PURGE	UPDATE
DISK	LEQ	RANDOM	WAIT
EQL	LSS	RB	WHEN
EXPOVR	MERGE	REVERSE	ZERO
FLAG	NEQ	SEARCH	ZIP
GEQ	NO	SEEK	
GTR	PAGE	SERIAL	
INDEX	PRINT	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 0101
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

INDEX

METALINGUISTIC VARIABLES

The syntactical definition of each Extended ALGOL metalinguistic variable will be found on the pages shown below.

⟨abnormal-condition label⟩ 6-52
⟨absolute address⟩ 6-33
⟨action labels⟩ 6-19
⟨action part⟩ 6-59
⟨actual parameter⟩ 3-7
⟨actual parameter list⟩ 3-7
⟨actual parameter part⟩ 3-7
⟨adding operator⟩ 4-2
⟨address⟩ 6-42
⟨arithmetic expression⟩ 4-1
⟨arithmetic operator⟩ 2-2
⟨array declaration⟩ 9-3
⟨array identifier⟩ 3-1
⟨array identifier list⟩ 10-2
⟨array kind⟩ 9-3
⟨array list⟩ 9-4
⟨array row⟩ 6-11
⟨array segment⟩ 9-4
⟨array specification⟩ 10-2
⟨array specifier⟩ 10-2
⟨array specifier list⟩ 10-2
⟨assignment statement⟩ 6-2
⟨basic component set⟩ 9-8
⟨basic statement⟩ 6-2
⟨basic symbol⟩ 2-1
⟨bits in field⟩ 3-3
⟨blank replacement transfer⟩ 11-9
⟨block⟩ 5-1
⟨block head⟩ 5-1
⟨blocking specifications⟩ 9-11
⟨Boolean expression⟩ 4-8
⟨Boolean factor⟩ 4-8
⟨Boolean primary⟩ 4-9
⟨Boolean secondary⟩ 4-8
⟨Boolean term⟩ 4-8
⟨bound pair⟩ 9-4
⟨bound pair list⟩ 9-4
⟨bracket⟩ 2-2
⟨break label⟩ 6-57
⟨break-out statement⟩ 6-32
⟨buffer part⟩ 9-11
⟨buffer release⟩ 6-19
⟨carriage control⟩ 6-27
⟨case statement⟩ 6-62
⟨case statement header⟩ 6-62
⟨character⟩ 1-4
⟨close statement⟩ 6-30
⟨compare procedure⟩ 12-1
⟨compound nest⟩ 11-14
⟨compound statement⟩ 5-1
⟨compound stream statement⟩ 11-16
⟨compound stream tail⟩ 11-1
⟨compound tail⟩ 5-1
⟨concatenate expression⟩ 4-16
⟨concatenate operator⟩ 4-16
⟨conditional statement⟩ 7-1
⟨conditional stream statement⟩ 11-17
⟨core size⟩ 12-1
⟨cycle number⟩ 6-39
⟨data comm close statement⟩ 6-61

⟨data comm input action labels⟩ 6-52
⟨data comm input parameters⟩ 6-52
⟨data communications I/O statement⟩ 6-49
⟨data comm output action labels⟩ 6-57
⟨data comm output parameters⟩ 6-57
⟨data comm read lock statement⟩ 6-54
⟨data comm read seek statement⟩ 6-56
⟨data comm read statement⟩ 6-52
⟨data comm record address and release part⟩ 6-52
⟨data comm rewind statement⟩ 6-61
⟨data comm write lock statement⟩ 6-59
⟨data comm write statement⟩ 6-57
⟨date⟩ 6-39
⟨decimal fraction⟩ 2-6
⟨decimal number⟩ 2-6
⟨decimal places⟩ 9-19
⟨declaration⟩ 9-1
⟨declarator⟩ 2-2
⟨define declaration⟩ 9-7
⟨defined identifier⟩ 9-7
⟨definition⟩ 9-7
⟨definition list⟩ 9-7
⟨definition part⟩ 9-7
⟨delimiter⟩ 2-2
⟨designational expression⟩ 4-14
⟨destination address part⟩ 11-6
⟨destination string statement⟩ 11-8
⟨digit⟩ 2-1
⟨direction⟩ 6-18
⟨disk access technique⟩ 9-11
⟨disk close statement⟩ 6-48
⟨disk file description⟩ 9-11
⟨disk input parameters⟩ 6-42
⟨disk I/O statement⟩ 6-42
⟨disk lock statement⟩ 6-49
⟨disk output parameters⟩ 6-44
⟨disk read seek statement⟩ 6-45
⟨disk read statement⟩ 6-42
⟨disk rewind statement⟩ 6-48
⟨disk space statement⟩ 6-47
⟨disk size⟩ 12-1
⟨disk write statement⟩ 6-44
⟨do statement⟩ 8-6
⟨double constant⟩ 6-9
⟨double expression⟩ 6-8
⟨double operator⟩ 6-9
⟨double primary⟩ 6-8
⟨double statement⟩ 6-8
⟨dummy statement⟩ 6-5
⟨dump declaration⟩ 9-38
⟨dump indicator⟩ 9-38
⟨dump list⟩ 9-38
⟨dump list element⟩ 9-38
⟨dump part⟩ 9-38
⟨edit and move read⟩ 6-41
⟨edit and move statement⟩ 6-41
⟨edit and move write⟩ 6-41
⟨editing phrase⟩ 9-19
⟨editing phrase type⟩ 9-19
⟨editing segment⟩ 9-19
⟨editing specifications⟩ 9-19
⟨empty⟩ 1-4
⟨end-of-file label⟩ 6-19
⟨exponent part⟩ 2-6
⟨expression⟩ 4-1
⟨expression list⟩ 9-33
⟨expression part⟩ 9-33
⟨factor⟩ 4-1

⟨fault declaration⟩ 9-40
⟨fault equate⟩ 9-40
⟨fault list⟩ 9-40
⟨fault statement⟩ 6-34
⟨fault type⟩ 6-34
⟨field⟩ 6-22
⟨field delimiter⟩ 6-22
⟨field description⟩ 3-3
⟨field part⟩ 9-19
⟨field width⟩ 9-19
⟨file declaration⟩ 9-10
⟨file identification⟩ 6-39
⟨file identification part⟩ 9-11
⟨file identifier⟩ 9-10
⟨file lock part⟩ 9-10
⟨file part⟩ 6-19
⟨fill statement⟩ 6-6
⟨fixed logical record size⟩ 9-11
⟨fixed physical record size⟩ 9-11
⟨for clause⟩ 8-1
⟨for-list⟩ 8-1
⟨for-list element⟩ 8-1
⟨for statement⟩ 8-1
⟨formal parameter⟩ 10-1
⟨formal parameter list⟩ 10-1
⟨formal parameter part⟩ 10-1
⟨format⟩ 6-19
⟨format and list part⟩ 6-19
⟨format declaration⟩ 9-19
⟨format identifier⟩ 9-19
⟨format part⟩ 9-19
⟨forward procedure declaration⟩ 9-35
⟨forward reference declaration⟩ 9-35
⟨forward switch declaration⟩ 9-35
⟨free-field data⟩ 6-21
⟨free-field part⟩ 6-19
⟨function designator⟩ 3-7
⟨general components⟩ 3-1
⟨general primary⟩ 4-16
⟨go to statement⟩ 6-5
⟨hivalue procedure⟩ 12-1
⟨identifier⟩ 2-5
⟨identifier list⟩ 10-1
⟨if clause⟩ 4-1
⟨if statement⟩ 7-1
⟨illegitimate character⟩ 1-4
⟨implication⟩ 4-8
⟨indexed file identifier⟩ 6-15
⟨indexed switch file designator⟩ 6-15
⟨initial value⟩ 6-6
⟨in-out part⟩ 9-10
⟨input convert⟩ 11-8
⟨input option⟩ 12-1
⟨input or output⟩ 9-19
⟨input procedure⟩ 12-1
⟨input parameters⟩ 6-18
⟨integer⟩ 2-6
⟨interrogate function⟩ 6-59
⟨I/O statement⟩ 6-18
⟨iterative statement⟩ 8-1
⟨jump out statement⟩ 11-14
⟨label⟩ 4-14
⟨label declaration⟩ 9-9
⟨label equation information⟩ 6-37
⟨label equation part⟩ 9-10
⟨label equation statement⟩ 6-37
⟨label list⟩ 9-10
⟨label part⟩ 9-10
⟨least-significant portion⟩ 6-9
⟨least-significant variable⟩ 6-9

⟨left base⟩ 4-16
⟨left bit of field⟩ 3-3
⟨left bit of left base⟩ 4-16
⟨left bit of right base⟩ 4-16
⟨left part⟩ 6-3
⟨left part list⟩ 6-3
⟨letter⟩ 2-1
⟨letter string⟩ 2-8
⟨library call statement⟩ 6-8
⟨library designator⟩ 6-8
⟨link description⟩ 4-16
⟨link part⟩ 4-16
⟨list⟩ 9-33
⟨list declaration⟩ 9-33
⟨list identifier⟩ 9-33
⟨list part⟩ 9-33
⟨list segment⟩ 9-33
⟨literal bits⟩ 11-9
⟨literal characters⟩ 11-9
⟨literal transfer⟩ 11-9
⟨local or own type⟩ 9-2
⟨lock statement⟩ 6-30
⟨logical operator⟩ 2-2
⟨logical value⟩ 2-2
⟨lower bound⟩ 9-4
⟨lower bound list⟩ 10-2
⟨mask⟩ 6-33
⟨merge file⟩ 12-5
⟨merge file list⟩ 12-5
⟨merge statement⟩ 12-5
⟨mode part⟩ 9-10
⟨monitor declaration⟩ 9-36
⟨monitor list⟩ 9-36
⟨monitor list element⟩ 9-36
⟨monitor part⟩ 9-36
⟨most-significant portion⟩ 6-9
⟨most-significant variable⟩ 6-9
⟨multi-file identification⟩ 6-39
⟨multi-file identification part⟩ 9-11
⟨multiplying operator⟩ 4-2
⟨nest⟩ 11-14
⟨no-input label⟩ 6-52
⟨number⟩ 2-6
⟨number of areas⟩ 9-11
⟨number of bits in link⟩ 4-16
⟨number of buffers⟩ 9-11
⟨number of nests⟩ 11-14
⟨number of records⟩ 6-26
⟨number of tapes⟩ 12-1
⟨octal digit⟩ 6-6
⟨octal number⟩ 6-6
⟨operator⟩ 2-2
⟨output convert⟩ 11-8
⟨output-impossible label⟩ 6-57
⟨output media digit⟩ 6-39
⟨output media part⟩ 9-10
⟨output option⟩ 12-1
⟨output parameters⟩ 6-27
⟨output procedure⟩ 12-1
⟨parameter delimiter⟩ 3-7
⟨parity label⟩ 6-19
⟨partial word designator⟩ 3-3
⟨partial word operand⟩ 3-3
⟨primary⟩ 4-1
⟨procedure body⟩ 10-2
⟨procedure declaration⟩ 10-1
⟨procedure heading⟩ 10-1
⟨procedure identifier⟩ 3-7
⟨procedure statement⟩ 6-10
⟨procedure type⟩ 9-35
⟨program⟩ 5-1
⟨program designator⟩ 6-8
⟨proper string⟩ 2-7

⟨read statement⟩ 6-18
⟨recall address statement⟩ 11-8
⟨record address part⟩ 6-44
⟨record address and release part⟩ 6-42
⟨record length⟩ 12-1
⟨record specifications⟩ 9-11
⟨reel number⟩ 6-39
⟨relation⟩ 4-9
⟨relational operator⟩ 2-2
⟨release statement⟩ 6-25
⟨repeat part⟩ 9-19
⟨repetitive indicator⟩ 11-8
⟨rewind statement⟩ 6-29
⟨right base⟩ 4-16
⟨row⟩ 6-6
⟨row designator⟩ 6-6
⟨save factor⟩ 9-11
⟨search statement⟩ 6-62
⟨seconds⟩ 6-32
⟨separator⟩ 2-2
⟨sequential operator⟩ 2-2
⟨set address statement⟩ 11-6
⟨simple arithmetic expression⟩ 4-1
⟨simple Boolean⟩ 4-8
⟨simple designational expression⟩ 4-14
⟨simple variable⟩ 3-1
⟨single space⟩ 1-4
⟨size of areas⟩ 9-11
⟨size specifications⟩ 12-1
⟨skip address statement⟩ 11-7
⟨skip bit statement⟩ 11-13
⟨skip to channel⟩ 6-27
⟨sort statement⟩ 12-1
⟨source address part⟩ 11-6
⟨source bit⟩ 11-17
⟨source for alpha⟩ 11-17
⟨source or destination bit⟩ 11-13
⟨source string transfer⟩ 11-8
⟨source with destination⟩ 11-17
⟨source with literal⟩ 11-17
⟨space⟩ 1-4
⟨space statement⟩ 6-26
⟨specification⟩ 10-1
⟨specification list⟩ 10-1
⟨specification part⟩ 10-1
⟨specificator⟩ 2-2
⟨specified lower bound⟩ 10-2
⟨specifier⟩ 10-1
⟨statement⟩ 6-1
⟨status word⟩ 6-50
⟨store address statement⟩ 11-6
⟨stream actual parameter⟩ 6-15
⟨stream actual parameter list⟩ 6-15
⟨stream address index⟩ 11-6
⟨stream address statement⟩ 11-6
⟨stream arithmetic expression⟩ 11-7
⟨stream block⟩ 11-1
⟨stream block head⟩ 11-1
⟨stream declaration⟩ 11-1
⟨stream dummy statement⟩ 11-16
⟨stream formal parameter part⟩ 11-1
⟨stream go to statement⟩ 11-13
⟨stream if clause⟩ 11-17
⟨stream name parameter⟩ 6-15
⟨stream nest statement⟩ 11-14
⟨stream primary⟩ 11-7
⟨stream procedure call statement⟩ 6-15
⟨stream procedure declaration⟩ 11-1
⟨stream procedure heading⟩ 11-1
⟨stream procedure identifier⟩ 6-15

⟨stream release statement⟩ 11-15
⟨stream repeat part⟩ 11-8
⟨stream simple variable⟩ 11-1
⟨stream statement⟩ 11-5
⟨stream tally statement⟩ 11-14
⟨stream value parameter⟩ 6-15
⟨stream variable declaration⟩ 11-1
⟨stream variable list⟩ 11-1
⟨string⟩ 2-7
⟨string bracket character⟩ 1-4
⟨string character⟩ 1-4
⟨subscript expression⟩ 3-1
⟨subscript list⟩ 3-1
⟨subscripted variable⟩ 3-1
⟨switch declaration⟩ 9-6
⟨switch designator⟩ 4-14
⟨switch file declaration⟩ 9-18
⟨switch file designator⟩ 3-4
⟨switch file identifier⟩ 3-4
⟨switch file list⟩ 9-18
⟨switch format declaration⟩ 9-32
⟨switch format designator⟩ 3-5
⟨switch format identifier⟩ 3-5
⟨switch format list⟩ 9-32
⟨switch identifier⟩ 4-14
⟨switch list⟩ 9-6
⟨switch list declaration⟩ 9-34
⟨switch list designator⟩ 3-6
⟨switch list identifier⟩ 9-34
⟨switch list list⟩ 9-34
⟨term⟩ 4-1
⟨terminal buffer specifier⟩ 6-52
⟨test⟩ 11-17
⟨transfer and add⟩ 11-8
⟨transfer and convert⟩ 11-8
⟨transfer character portions⟩ 11-8
⟨transfer characters⟩ 11-8
⟨transfer part⟩ 11-8
⟨transfer type⟩ 11-8
⟨transfer words⟩ 11-8
⟨type⟩ 9-2
⟨type declaration⟩ 9-2
⟨type list⟩ 9-2
⟨unblocked specification⟩ 9-11
⟨unconditional statement⟩ 6-2
⟨unconditional stream statement⟩ 11-5
⟨unlabeled basic statement⟩ 6-2
⟨unlabeled block⟩ 5-1
⟨unlabeled compound statement⟩ 5-1
⟨unlabeled stream statement⟩ 11-5
⟨unsigned integer⟩ 2-6
⟨unsigned number⟩ 2-6
⟨upper bound⟩ 9-4
⟨value list⟩ 6-6
⟨value part⟩ 10-1
⟨variable⟩ 3-1
⟨variable identifier⟩ 3-1
⟨visible string character⟩ 1-4
⟨wait part⟩ 6-52
⟨wait statement⟩ 6-33
⟨well-formed construct⟩ 9-7
⟨when statement⟩ 6-32
⟨while statement⟩ 8-6
⟨word count⟩ 6-25
⟨write statement⟩ 6-27
⟨zip statement⟩ 6-36

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:

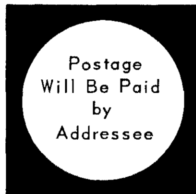
FROM: NAME ____________________
TITLE ____________________
COMPANY ____________________
ADDRESS ____________________
____

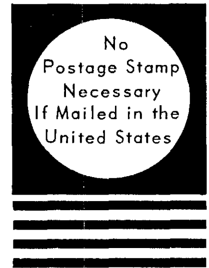
DATE ____________________

cut along dotted line

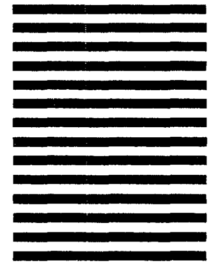
STAPLE

FOLD DOWN SECOND FOLD DOWN





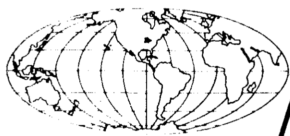
BUSINESS REPLY MAIL
First Class Permit No. 817, Detroit, Mich. 48232



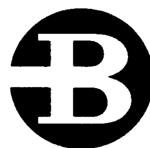
Burroughs Corporation
6071 Second Avenue
Detroit, Michigan 48232

attn: Sales Technical Services
Systems Documentation

FOLD UP FIRST FOLD UP



Wherever There's
Business There's



Burroughs