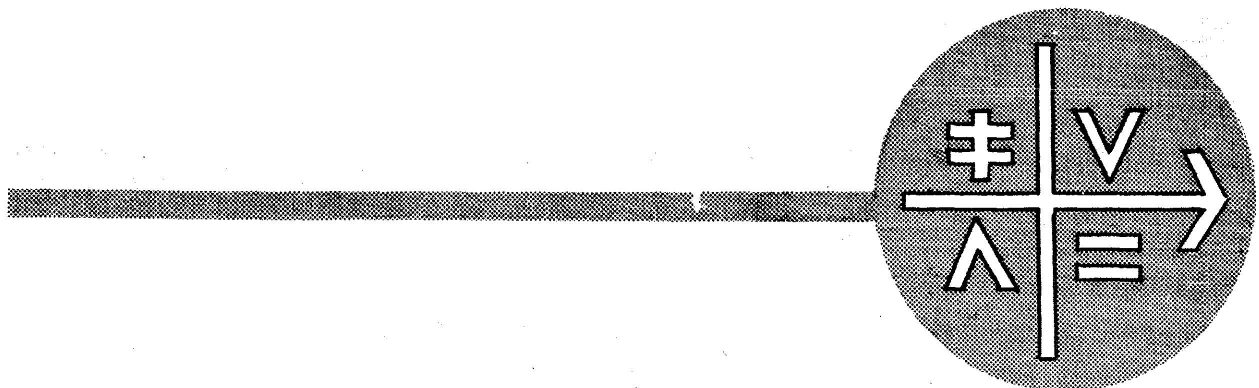
EXTENDED ALGOL REFERENCE MANUAL

for the

Burroughs B 5000



5000-21012
November, 1962
(Revised August, 1963)

EXTENDED ALGOL REFERENCE MANUAL

for the

Burroughs B 5000

Equipment and Systems Marketing Division

Sales Technical Services

Burroughs Corporation

Detroit , Michigan

COPYRIGHT © 1962 BURROUGHS CORPORATION

PREFACE

One of the programing languages utilized by the Burroughs B 5000 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.

The reader is assumed to have had some experience in systems programing. For those unfamiliar with ALGOL 60, the Burroughs B 5000 Information Processing System, or both, the following publications are suggested:

1. An Introduction to ALGOL 60 (Bulletin 5000-21001-P)
2. Master Control Program Characteristics for the Burroughs B 5000 Information Processing System (Bulletin 5000-21003-P)
3. The Descriptor, a Definition of the B 5000 Information Processing System (Bulletin 5000-20002-P)
4. File Control on the Burroughs B 5000
5. Stream Procedure, a Part of Extended ALGOL for the Burroughs B 5000
6. Operational Characteristics of the Processors for the Burroughs B 5000
7. Input-Output Facilities—A Part of Extended ALGOL for the Burroughs B 5000
8. Naur, P., et al., Report on the Algorithmic Language ALGOL 60 (Communications of the Association for Computing Machinery, Vol. 3, No. 5; May, 1960).
9. McCracken, Daniel D., An Introduction to ALGOL Programming (New York, New York: John Wiley and Sons, 1962)

In many cases, portions of reference 8 have been reproduced in this manual with little or no change, in order to adhere as closely as possible to the formal definition of ALGOL 60.

Every effort has been made to produce a complete, precise, and concise definition of Extended ALGOL. Suggestions and corrections for future editions of this manual will be appreciated, and should be addressed to:

Manager, Sales Technical Services
Burroughs Corporation
6071 Second Avenue
Detroit , Michigan

NOTE: Changes and additions made in the revision of August, 1963 are reflected in the Table of Contents. However, the Index is virtually unaffected, and therefore has not been revised.

TABLE OF CONTENTS

(NOTE: The various elements of Extended ALGOL are discussed in subsections or paragraphs labelled Syntax, Examples, and Semantics, immediately following each pertinent subject heading. To avoid meaningless repetition, these subordinate headings have been omitted from the Table of Contents.)

Introduction 1

1.0 Structure of the Language 3

1.1 Conventions Used in the Description of the Language 4

1.2 Character Set 5

2.0 Basic Components: Basic Symbols, Identifiers, Numbers, and Strings . . 6

2.1 Letters 6

2.2 Digits 7

2.3 Logical Values. 7

2.4 Delimiters. 7

2.4.3 Spacing. 8

2.4.4 The Use of COMMENT 9

2.5 Identifiers 9

2.6 Numbers 10

2.6.4 Size Limitations of Numbers. 11

2.7 Strings 11

2.7.4 Use of Strings 12

2.8 Quantities, Kinds, and Scopes 12

2.9 Values and Types. 12

3.0 Expressions 13

3.1 Variables 13

3.1.4 Simple Variables 14

3.1.5 Subscripted Variables. 14

3.1.5.1 Number of Subscripts. 14

3.1.5.2 Evaluation of Subscripts. 15

3.2 Partial Word Designators. 15

3.2.4 Values Allowed for Field 16

3.3 Function Designators 16
3.3.4 Standard Functions 17
3.3.5 Type Transfer Functions 18
3.3.5.1 ENTIER 18
3.3.5.2 REAL 18
3.3.5.3 BOOLEAN 18
3.4 Arithmetic Expressions 19
3.4.3.1 Simple Arithmetic Expressions 20
3.4.3.2 General Arithmetic Expressions 21
3.4.4 Operators and Types 21
3.4.4.1 Arithmetic Operators 21
3.4.4.2 Arithmetic Expression Types 23
3.4.5 Precedence of Operators 23
3.4.6 Numerical Limitations and Significant Digits 24
3.5 BOOLEAN Expressions 25
3.5.3.1 Simple Boolean Expressions 26
3.5.3.2 General Boolean Expressions 27
3.5.4 Types 28
3.5.5 Relational and Logical Operators 28
3.5.5.1 Relational Operators 28
3.5.5.2 Logical Operators 28
3.5.6 Precedence of Operators 29
3.6 Designational Expressions 29
3.6.3.1 Simple Designational Expressions 30
3.6.3.2 General Designational Expressions 31
3.6.4 The Subscript Expression of a Switch Designator 31
3.6.5 Unsigned Integers as Labels 31

4.0 Statements 33
4.1 Compound Statements and Blocks 33
4.1.3.1 Nested Blocks 35
4.1.3.2 Disjoint Blocks 35
4.2 Assignment Statements 35
4.2.4 Types 36
4.3 GO TO Statements 37
4.4 Dummy Statements 38

4.5 Conditional Statements 38
4.5.3.1 IF Statement 39
4.5.3.2 IF...THEN...ELSE Statement 39
4.5.3.3 IF...FOR Statement 40
4.6 FOR Statements 40
4.6.4 The For List. 41
4.6.4.1 Arithmetic Expression Element. 42
4.6.4.2 Step-Until Element 42
4.6.4.3 WHILE Element 43
4.6.4.4 STEP-WHILE Element 43
4.6.5 Value of the Controlled Variable Upon Exit from the FOR Statement . . 44
4.6.6 GO TO Leading Into a FOR Statement. 44
4.7 Procedure Statements 44
4.7.3.1 Value Assignment (Call by Value) 46
4.7.3.2 Name Replacement (Call by Name). 47
4.7.4 Restrictions. 49
4.8 I-O Statements 49
4.8.2 READ Statement. 49
4.8.3 WRITE Statement 51
4.8.4 RELEASE Statement 52
4.9 FILL Statement 53
4.9.3.1 Row Designator 53
4.9.3.2 Value List 54

5.0 Declarations 55
5.1 Type Declarations. 56
5.1.4 Local or OWN. 56
5.1.5 Type. 57
5.2 ARRAY Declarations 57
5.2.3.1 SAVE Arrays 58
5.2.3.2 Local or OWN 58
5.2.3.3 Type 58
5.2.4 Bound Pair List 59
5.3 SWITCH Declarations. 59
5.3.4 Evaluation of Expressions in the Switch List. 60
5.3.5 Influence of Scope. 60

5.4 DEFINE Declarations. 60
5.4.1 Syntax 60
5.4.2 Examples 60
5.4.3 Semantics. 60
5.4.4 Influence of Scope 60
5.4.5 Restrictions 61
5.5 LABEL Declarations 61
5.6 FILE Declarations. 61
5.6.3.1 Buffer Part 62
5.6.3.2 I-O Unit Control. 62
5.6.3.3 Disposition 63
5.6.3.4 Blocking. 63
5.6.3.5 End-of-File 63
5.6.3.6 Save Factor 64
5.6.3.7 FILE REVERSE. 64
5.6.3.8 Scope 64
5.6.3.9 Restrictions. 64
5.7 FORMAT Declarations. 65
5.7.3.1 Input Editing Specifications. 65
5.7.3.2 Input Editing Phrases 66
5.7.3.3 Output Editing Specifications 68
5.7.3.4 Output Editing Phrases. 68
5.8 LIST Declarations. 70
5.9 FORWARD Reference Declarations 72
5.10 Diagnostic Declarations. 72
5.10.4 MONITOR 73
5.10.4.1 Monitor List Elements. 73
5.10.5 DUMP. 74
5.10.5.1 Dump List Elements 74
5.11 PROCEDURE Declarations 76
5.11.3.1 Procedure Heading. 77
5.11.3.2 Procedure Body 78
5.11.3.3 Scope of Identifiers Other Than Formal Parameters. . . 78
5.11.4 Values of Function Designators. 78
5.11.5 Restriction on 78

5.12 STREAM PROCEDURE Declarations 79
5.12.3.1 Value Part 80
5.12.3.2 Stream Declarations. 80
5.12.3.3 Compound Stream Tail 81
5.12.3.4 Automatic Index Adjustment 81
5.12.4 Stream Statements 82
5.12.5 Stream Address Statement. 82
5.12.5.3 Set Address Statement. 83
5.12.5.4 Store Address Statement. 83
5.12.5.5 Recall Address Statement 84
5.12.5.6 Skip Address Statement 84
5.12.6 Destination String Statement. 84
5.12.6.3 Transfer Words 85
5.12.6.4 Transfer Characters. 85
5.12.6.5 Input Convert. 86
5.12.6.6 Output Convert 86
5.12.6.7 Transfer and Add 86
5.12.6.8 Transfer Character Portions. 87
5.12.6.9 Literal Characters 87
5.12.6.10 Literal Bits 88
5.12.7 Stream GO TO Statement. 88
5.12.8 SKIP Bit Statement. 88
5.12.9 Stream TALLY Statement. 89
5.12.10 Stream Nest Statements. 89
5.12.11 Stream RELEASE Statement. 90
5.12.12 Compound Stream Statement 90
5.12.13 Conditional Stream Statement. 91
5.12.13.4 Source with Literal. 91
5.12.13.5 Source with Destination. 92
5.12.13.6 Source Bit 92
5.12.13.7 TOGGLE 92
5.12.13.8 Source for Alpha 92

Appendix A: B 5000 Internal Character Codes (with text references) A-1
Index
Metalinguistic Variables. I-1
Reserved Words. I-7

INTRODUCTION

Extended ALGOL, one of the languages used for programing the Burroughs B 5000 Information Processing System, is based on the definitive "Report on the Algorithmic Language ALGOL 60" (Communications of the ACM, Vol. 3, No. 5; May, 1960). 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. The section and paragraph numbering used herein is similar to that employed in the document referenced above.

1.0 STRUCTURE OF THE LANGUAGE

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. Reserved words and symbols are grouped in a manner prescribed by the syntax to form the various constructs of the language. These constructs can be divided into three major categories: expressions, statements, and declarations.

Whereas ALGOL 60 itself is concerned with the formation of rules for calculation of a value or values, Extended ALGOL also includes the means required by a programmer to communicate with the computing equipment.

The rules for calculation are called expressions. Three different forms of expressions are present in the language: arithmetic, Boolean, and designational.

The results produced by the evaluation of arithmetic and Boolean expressions can be assigned as the values of variables by means of assignment statements. These assignment statements are the principal 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 labelled.

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 a variable may assume, or the existence of subroutines. All identifiers which are to be used in a program 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.)

1.1 Conventions Used in the Description of the Language.

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

⟨ ⟩	The left and right broken brackets are used to contain one or more characters which represent a metalinguistic variable whose value is given by a metalinguistic formula.
::=	The symbol ::= means "is defined as," and separates the metalinguistic variable on the left of the formula from its definition on the right.
|	The symbol | means "or." This symbol separates multiple definitions of a metalinguistic variable.
{ }	Braces are used to enclose metalinguistic variables which are defined by the meaning of the English language 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 combined to form 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 variable indicated.

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:

1 Same as ALGOL 60 except for character set*

2 Different from ALGOL 60

3 In addition to ALGOL 60 (all or in part)

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

n ⟨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.

This metalinguistic formula defines a recursive relationship by which a construct called an identifier may be formed. Evaluation of the formula shows that an identifier begins with a letter; the letter may stand alone, or may be followed by any mixture of letters and digits.

The number n (1, 2, or 3) indicates the departure of the defined construct from the definitions of ALGOL 60, as noted above.

1.2 Character Set

1.2.1 Syntax

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 | × | 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

---

* Formulas preceded by the number 1 represent the material presented in "Report on the Algorithmic Language, ALGOL 60" (Journal 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).

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

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

3 ⟨string bracket character⟩ ::= "

3 ⟨illegitimate character⟩ ::= ?

(NOTE: This character is not used in writing Extended ALGOL programs. It serves to represent, in the B 5000, any illegitimate card code detected during a card read operation. It is shown here merely to complete the illustration of the character set.)

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

1.2.2 Semantics

The above character set has been defined for the B 5000; 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.

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

2.0.1 Syntax

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

2.0.2 Semantics

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

2.1 Letters

2.1.1 Syntax

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

2.1.2 Semantics

The alphabet defined for Extended ALGOL is restricted to the upper-case letters of the English alphabet. The lower-case letters are specifically disallowed. Individual letters do not have individual meaning but serve to form identifiers and strings (see section 2.4, Identifiers, and section 2.6, Strings).

2.2 Digits

2.2.1 Syntax

1 ⟨digit⟩ ::= 0|1|2|3|4|5|6|7|8|9

2.2.2 Semantics

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

2.3 Logical Values

2.3.1 Syntax

1 ⟨logical value⟩ ::= TRUE|FALSE

2.3.2 Semantics

Logical values are the only values defined for BOOLEAN quantities (see section 5.1, Type Declarations).

2.4 Delimiters

2.4.1 Syntax

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

1 ⟨operator⟩ ::= ⟨arithmetic operator⟩ | ⟨relational operator⟩ | ⟨logical operator⟩ | ⟨sequential operator⟩

3 ⟨arithmetic operator⟩ ::= +|-|×|/|DIV|*|MOD

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

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

3 ⟨sequential operator⟩ ::= GO|TO|IF|THEN|ELSE|FOR|DO|READ|WRITE|RELEASE|DS|TOGGLE|JUMP|SKIP|DB|DI|SET|CI|SC|DC|RESET|SB|SI|TALLY|PAGE|DBL|NO

3 ⟨separator⟩ ::= ,|.|@|:|;|←|⟨single space⟩|STEP|UNTIL|WHILE|COMMENT|LOC|WDS|ADD|SUB|LIT|CHR|NUM|ZON|DEC|OCT|WITH|FILL

3 ⟨bracket⟩ ::= (|)|[|]|"|BEGIN|END|#

3 ⟨declarator⟩ ::= OWN|BOOLEAN|INTEGER|REAL|DOUBLE|ARRAY|SWITCH|LABEL|
LOCAL|FORWARD|SAVE|PROCEDURE|STREAM|LIST|FORMAT|
IN|OUT|MONITOR|DUMP|FILE|ALPHA|DEFINE|REVERSE

3 ⟨specificator⟩ ::= VALUE

2.4.2 Semantics

Delimiters are the class of operators, separators, brackets, and specificators. As the word "delimiter" indicates, an important function of these elements is to separate the various entities which go to make up a program. The exact meaning of each delimiter will be made clear as it appears in various constructs below. The delimiters may not be used for any purpose other than that defined by the language description. The symbols and words which make up the delimiters constitute the reserved vocabulary of Extended ALGOL. For a complete list of the reserved words see Appendix A.

2.4.3 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 this approach is not practical because of hardware limitations. 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, along with variables and numbers, to form expressions, statements, and declarations. Because certain 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 it is required that a space separate any two basic components of the following forms:

multi-character delimiter
identifier
logical value
unsigned number

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

2.4.4 The Use of COMMENT

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 characters not containing END or ; or . or ELSE or WHILE or UNTIL } except %	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.

2.5 Identifiers

2.5.1 Syntax

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

2.5.2 Examples

I

ID

A5

G76D3

ARITHMETICMEAN

2.5.3 Semantics

Identifiers are composed of letters and digits only, and must begin with a letter. No space may appear within an identifier. Identifiers have no inherent meaning, but serve to name variables, arrays, labels, switches, procedures, files, formats, and lists.

The identifiers used in a program may be chosen freely except for the following restrictions:

1. Reserved words of Extended ALGOL may not be used as identifiers.
2. An identifier must start with a letter, which can be followed by any combination of letters or digits or both. This restriction also applies to labels, as integer labels are specifically disallowed.
3. Identifiers may be as short as one letter or as long as 63 letters and digits.

2.6 Numbers

2.6.1 Syntax

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⟩

2.6.2 Examples

Unsigned integers:

    5
    69

Decimal fractions:

    .5
    .69

Decimal numbers:

    1354
    .546
    -1354.543

Integers:

    +546
    -62256

Exponent parts:

    @68
    @-46
    @+54

Unsigned numbers:

    1354.543
    @68
    1354.543@68

Numbers:

    0
    +549755813887
    1.75@-46
    4.314@68

2.6.3 Semantics

Numbers may be of two basic types, INTEGER and REAL. Integers are of type INTEGER. All other numbers are of type REAL.

Unsigned numbers are composed of digits and four basic symbols: .@+ and -. No space may appear within an unsigned number, otherwise it will be interpreted as more than one number. Spaces OK up to 63 within number

2.6.4 Size Limitations of Numbers

In general, the number of digits (disregarding the decimal point and exponentiation, if any) in a decimal 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 exponentiation, they do not exceed 549755813887 in value. The last series of examples, Numbers (subsection 2.6.2), 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 section 3.4.6, Numerical Limitations and Significant Digits.

2.7 Strings

2.7.1 Syntax

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

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

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

2.7.2 Examples

Letter string:

A
ABCDEF
ALGOL

Proper string:

#
#A&FG
ALGOL

String:

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

2.7.3 Semantics

Strings in general are of three forms:

1. A letter string
2. A proper string delimited on both ends with the string bracket character.
3. """"

The first case, a 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.

2.7.4 Use of Strings

Strings can be used to form arithmetic expressions (see section 3.4, Arithmetic Expressions), format declarations (section 5.8, Format Declarations), and destination string statements (section 5.13, Stream Procedure Declarations). A letter string may be used for explanatory purposes in an actual parameter list (section 4.7, Procedure Statements) or a formal parameter list (section 5.12, Procedure Declarations). A string may not exceed 63 characters in length.

2.8 Quantities, Kinds, and Scopes

The following kinds of quantities are distinguished in Extended ALGOL: simple variables, arrays, labels, switches, and procedures. In addition certain other constituents are declared: files, formats, defines, lists, forward references, and diagnostics. The scope of any quantity or constituent is the block in which the quantity or constituent is declared. All the above quantities and constituents must be declared before they are referenced in any manner.

2.9 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 or a set of logical values.

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

3.0 EXPRESSIONS

Expressions are basic to any algorithmic process. The following kinds of expressions are defined for Extended ALGOL: arithmetic, Boolean, and designational. Expressions are rules for combining basic components in a fashion such that meaningful values can be obtained. Expressions are formed from the following quantities: logical values, numbers, variables, function designators, partial word designators, and elementary arithmetic, relational, logical, and sequential operators.

Because expressions are used to define subscripted variables and function designators, and these quantities are in turn used to define expressions, the definition of expressions is necessarily recursive.

3.0.1 Syntax

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

3.1 Variables

3.1.1 Syntax

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⟩

3.1.2 Examples

Simple variables:

```
ALPHAINFO
BETA4
Q
```

Subscript lists:

```
5
ITH
ITH,JTH
ITH + 2,JTH - ITH
IF BETA = 30 THEN -2 ELSE K + 2
```

Subscripted variables:

```
A[5]
A[ITH]
KRONECKER[ITH + 2,JTH - ITH]
MAXQ[IF BETA = 30 THEN -2 ELSE K + 2]
```

3.1.3 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 section 4.2, Assignment Statements). There are two forms of variables: simple and subscripted.

3.1.4 Simple Variables

A simple variable is defined as being composed of a variable identifier which is in turn defined as an identifier. The type of value that a simple variable may represent is defined by its type declaration (see section 5.1, Type Declarations).

3.1.5 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 section 5.2, Array Declarations). The subscript list specifies one element of the array. A subscript expression is defined as an arithmetic expression; each arithmetic expression used as a subscript expression occupies a subscript position in the subscript list, and is referred to as a subscript.

3.1.5.1 Number of Subscripts

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

3.1.5.2 Evaluation of Subscripts

Each subscript expression in the subscript list is evaluated from left to right. Each subscript expression is treated as though it were a variable of type INTEGER. If, upon evaluation, the subscript expression yields a value of type REAL, the following transfer operation is automatically invoked:

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

(see subsection 3.3.5, Transfer Functions).

The values which result from the evaluation of the subscript expressions provide the actual integral values of the subscripts by which the array component is referenced. If the value of a subscript falls outside the limits declared for the array, the value of the element so referenced is undefined, and a program error will result.

3.2 Partial Word Designators

3.2.1 Syntax

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⟩

3.2.2 Examples

Field descriptions:

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

Partial word designators:

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

3.2.3 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. The quantities to which partial word designators can be applied consist of simple and subscripted variables, function designators (see section 3.3, Function Designators) and arithmetic expressions enclosed in parentheses (see section 3.4, Arithmetic Expressions).

3.2.4 Values Allowed for Field

The value of a partial word operand is contained in a word 48 bits in length. The addressable bits in this word are numbered left-to-right from 1 to 47. (Bit 0 cannot be addressed.) Therefore the value of ⟨left bit of field⟩ and ⟨bits in field⟩ may not exceed 47. In addition, ⟨left bit of field⟩ is restricted to values ranging from 1 to 47, and the sum of ⟨left bit of field⟩ and ⟨bits in field⟩ must not be greater than 48 (e.g., [46:2] specifies bits 46 and 47).

3.3 Function Designators

3.3.1 Syntax

```
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 identifier⟩|
                          ⟨switch identifier⟩|⟨procedure identifier⟩|
                          ⟨file identifier⟩|⟨format identifier⟩|
                          ⟨list identifier⟩
2  ⟨parameter delimiter⟩ ::= ,|)"⟨letter string⟩"(
```

3.3.2 Examples

Actual parameter parts:

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

Function Designators:

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

3.3.3 Semantics

A function designator defines a single value. This value is produced as a result of the application of a given set of rules defined by a special form of a PROCEDURE declaration (section 5.11, 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 arithmetic or Boolean expressions (see section 3.4, Arithmetic Expressions, and section 3.5, Boolean Expressions).

3.3.4 Standard Functions

The standard functions supplied for Extended ALGOL are listed below with appropriate definitions. Where 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 principal 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 will all yield values of type REAL, except for SIGN (AE) which produces a value of type INTEGER. The function ABS (AE) will also produce a result of type INTEGER when the value which results from 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, as they are an integral part of the compiler itself.

3.3.5 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)

3.3.5.1 ENTIER

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

3.3.5.2 REAL

The function REAL (BE) yields a value of type REAL. The use of this function does not alter the internal B 5000 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

3.3.5.3 BOOLEAN

The function BOOLEAN (AE) yields a value of type BOOLEAN. The use of this function does not alter the internal B 5000 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 (section 3.5.5.2) operate on the entire word in the B 5000.

---

* With the exception that arithmetic expressions of type DOUBLE are truncated to type REAL.

3.4 Arithmetic Expressions

3.4.1 Syntax

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

1 ⟨adding operator⟩ ::= +|-

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

3.4.2 Examples

Primaries:

```
5.678
Y1[1,2]
COS(A + B)
(IF X = 1 THEN 5.5 ELSE Q/2)
I.[9:39]
"ALPHA"
```

Factors:

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

Terms:

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

Simple Arithmetic Expressions:

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

Arithmetic Expressions:

```
(IF X = 1 THEN 5.5 ELSE Y/2)
Q*V*2
P MOD 2
+3
IF ERROR[I] = 1 THEN "OVFLOW" ELSE "UNFLOW"
IF B = 0 THEN X ELSE Y + 2
```

3.4.3 Semantics

An arithmetic expression is a rule for computing a numerical value. Arithmetic expressions may be divided into two categories, simple and general.

3.4.3.1 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 of which it is composed. The arithmetic operators are explained in detail in subsection 3.4.4.

The value of a primary is obvious when it is a number. If the primary is a variable, the value of the primary is the current value of that variable. When the primary is a partial word designator, its value is that portion of the current value of the indicated variable designated by the field (see section 3.2, Partial Word Designators).

The value of a function designator is that obtained by applying those computing rules defined by the PROCEDURE declaration (section 5.11, Procedure Declarations) to the current values of the actual parameters. In the case of the standard functions, the computing rules to be applied are inherent in the language and are not explicitly defined by a PROCEDURE declaration (see subsection 3.3.4.). Finally, for a primary which is an arithmetic expression enclosed in parentheses, the value derived must be described in terms of the primaries from which it is formed.

A special case results when a primary is a string. If a primary is a string it must not exceed ~~six~~ characters in length, and if it is used in an arithmetic calculation it will be treated as a variable of type REAL (see section 5.1, Type Declarations).

3.4.3.2 General Arithmetic Expressions

A general arithmetic expression is composed of an IF clause followed by a simple arithmetic expression which is delimited by ELSE and followed by an arithmetic expression (see the last example under Arithmetic Expressions in subsection 3.4.2).

The evaluation of the general arithmetic expression proceeds as follows: The Boolean expression is evaluated (see section 3.5, Boolean Expressions). If the value of the Boolean expression is TRUE, the simple arithmetic expression is evaluated and the evaluation of the general 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 expression following the delimiter ELSE may also be a general arithmetic expression. As a result, the general arithmetic expression could contain a series of IF clauses. The Boolean expressions in these clauses would be evaluated as above from left to right until a logical value of TRUE was found. Then the value of the succeeding simple arithmetic expression would be the value of the entire arithmetic expression. If no Boolean expression had the value TRUE, the value of the entire arithmetic expression would be that of the simple arithmetic expression following the last ELSE.

3.4.4 Operators and Types

The constituent variables of an arithmetic expression must be of type INTEGER, REAL, DOUBLE, or ALPHA. Note, however, that variables of type BOOLEAN may occur in an IF clause of an arithmetic expression. (See section 5.1, Type Declarations.) Definitions of the various arithmetic operators are given in the paragraphs below.

3.4.4.1 Arithmetic Operators

The operators +, −, × and / have the conventional mathematical meanings: addition, subtraction, multiplication, and division.

The operator DIV is defined only for operands of type INTEGER. It yields a result defined as follows:

Y DIV Z = SIGN (Y/Z) × 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 * denotes exponentiation. Its meaning depends on the types and values of the operands involved, as shown below. Consider Y * Z:

	IF Z IS TYPE INTEGER AND			IF Z IS TYPE REAL OR DOUBLE 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 × Y ×...× Y (Z times).

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

Note 3: Y * Z = EXP(Z × LN(Y)).

Note 4: Value of expression is undefined.

The operator MOD produces a result defined as follows:

Y MOD Z = Y - Z × (SIGN(Y / Z) × ENTIER (ABS(Y / Z)))

The operator MOD is undefined if either or both operands are of type DOUBLE.

3.4.4.2 Arithmetic Expression Types

The type of a value resulting from an arithmetic operation depends upon the types of the operands as well as the arithmetic operators used in obtaining that value. All cases are shown in the following table. (See Note 1, below.)

OPERAND ON LEFT	OPERAND ON RIGHT	VALUE RESULTING FROM THE ARITHMETIC OPERATOR +,-,×	/	DIV	*	MOD
Integer	Integer	Integer	Real	Integer	Note 2	Integer
Integer	Real	Real	Real	Undefined	Real	Real
Integer	Double	Double	Double	Undefined	Real	Undefined
Real	Integer	Real	Real	Undefined	Real	Real
Real	Real	Real	Real	Undefined	Real	Real
Real	Double	Double	Double	Undefined	Real	Undefined
Double	Integer	Double	Double	Undefined	Double	Undefined
Double	Real	Double	Double	Undefined	Real	Undefined
Double	Double	Double	Double	Undefined	Real	Undefined

Note 1: In arithmetic operations, operands of type ALPHA are handled as if they were of type REAL.

Note 2: If the operand on the right is less than zero, REAL; otherwise, INTEGER.

3.4.5 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 the following rules of precedence.

Each arithmetic operator has one of three orders of precedence associated with it as follows:

First: *
Second: × / DIV MOD
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.

3.4.6 Numerical Limitations and Significant Digits

Normally the result of an arithmetic operation involving the operators +, -, and × is of type INTEGER if both operands are of type INTEGER (see section 3.4.4, Operators and Types). If the value of the result exceeds 549,755,813,887 however, it will become of type REAL to ensure that least-significant rather than most-significant digits are lost. Therefore the maximum absolute value of type INTEGER that an arithmetic operation may yield is 549,755,813,887.

Since the B 5000 utilizes an octal number system the range of absolute REAL values can best be expressed as follows:

(8 * 13 - 1) × 8 * 63 to 8 * (- 63), and zero
or approximately
4.3 @68 to 7.8 @- 56, and zero

The ranges of DOUBLE values are the same as those for REAL values. The difference between DOUBLE and REAL is in the number of significant digits. Whereas REAL values have 13 significant octal digits, DOUBLE values have 26.

3.5 BOOLEAN Expressions

3.5.1 Syntax

```
1  ⟨Boolean expression⟩ ::= ⟨simple Boolean⟩|⟨if clause⟩⟨simple Boolean⟩
                            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⟩

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

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

3.5.2 Examples

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]
```

Boolean Secondaries:

```
TRUE
NOT A ≠ C
```

Boolean Factors:

```
GATE[1,2]
NOT A ≠ C
Q.[16:1] AND 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]
```

Implications:

```
TRUE
GATE[1,2]
NOT A ≠ C IMP GATE[1,2]
```

Simple Boolean Expressions:

```
TRUE
DIODE
NOT A ≠ C IMP GATE[1,2]
```

Boolean Expressions:

```
TRUE
NOT A ≠ C
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]
```

3.5.3 Semantics

A Boolean expression is a rule for computing a logical value. Boolean expressions can be divided into two categories: simple Boolean expressions and general Boolean expressions.

3.5.3.1 Simple Boolean Expressions

A simple Boolean expression is formed of 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 subsection 3.5.5).

The logical operators are analyzed in detail below.

The value which results upon evaluation of a simple Boolean expression depends upon the primary or primaries which are used to form the expression. When the primary involved is a logical value, the value is obvious. If the primary is a Boolean variable, the value is that logical value currently represented by the variable. When the primary is a partial word designator, the logical value is the field specified (see section 3.2, Partial Word Designators). In the case where the primary is a function designator, the logical value is obtained by applying those computing rules defined by the associated PROCEDURE declaration (see section 5.11, Procedure Declarations) to the current values

of the actual parameters. If the primary is a relation, the simple arithmetic expressions which form the relation are evaluated, the values produced are tested against each other according to the operation of the specific relational operator involved, and a logical value is produced as a result of this test. Finally, when a primary is a Boolean expression enclosed in parentheses, the logical value derived must be described in terms of the primaries from which it is formed.

3.5.3.2 General Boolean Expressions

The general Boolean expression is composed of an IF clause, followed by a simple Boolean expression delimited by ELSE, which is in turn followed by a Boolean expression. The IF clause is composed of IF followed by a Boolean expression delimited by THEN.

The simplest form of the general Boolean expression occurs when the IF clause contains a simple Boolean expression. The evaluation of the general Boolean expression in this case proceeds as follows: The simple Boolean expression of the IF clause is evaluated according to the methods described previously (paragraph 3.5.3.1, Simple Boolean Expressions). If the resulting logical value is TRUE, the simple Boolean expression following the delimiter THEN is evaluated, thus completing the evaluation of the general Boolean expression. If the logical value produced in the IF clause is FALSE the Boolean expression following the delimiter ELSE is evaluated, thereby completing the evaluation of the general Boolean expression.

The Boolean expression in the IF clause, or the one following the delimiter ELSE, or both, can be a general Boolean expression. In this event, the IF clause can consist of a series of IF clauses and the Boolean expression following the delimiter ELSE can also consist of a series of IF clauses. Such a construct is said to be nested. The innermost Boolean expression of a nested Boolean expression is evaluated first, then the next outer, and so on, following the procedure described above.

3.5.4 Types

The quantities which are used to form Boolean expressions must have been declared as type BOOLEAN (see section 5.1, Type Declarations, and subsection 5.11.4, Values of Function Designators), with the exception of the constituents of relations and those quantities which are under the influence of type transfer functions (see subsection 3.3.5, Type Transfer Functions).

3.5.5 Relational and Logical Operators

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

3.5.5.1 Relational Operators

The relational operators denote the following relations:

$<$	is less than
$\leq$	is less than or equal to
$=$	is equal to
$\geq$	is greater than or equal to
$>$	is greater than
$\neq$	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.

3.5.5.2 Logical Operators

The operation of the logical operators NOT (negation), AND (logical product), OR (logical sum), IMP (implication), and EQV (logical equivalence) is defined by the following table.

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 ≤ B2
B1 EQV B2	true	false	false	true	B1 = B2

3.5.6 Precedence of operators

The sequence of operations within a simple Boolean expression is generally from left to right, with the additional rules shown below.

3.5.6.1 The following specific rules of precedence are defined.

First: arithmetic expressions according to subsection 3.4.5.

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

Third: NOT ~

Fourth: AND ∧

Fifth: OR ∨

Sixth: IMP ≤

Seventh: EQV =

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

3.6 Designational Expressions

3.6.1 Syntax

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

3.6.2 Examples

Switch designators:

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

Simple designational expressions:

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

Designational expressions:

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

3.6.3 Semantics

A designational expression is a rule for obtaining a label of a statement (see section 4.0, Statements). Again, designational expressions may be differentiated as simple designational and general designational expressions.

3.6.3.1 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 formed of a switch designator, the actual numerical value of the subscript expression (see subsection 3.6.4) designates one of the elements in the switch list. The element selected may be any form of simple designational expression, which is then evaluated as stated above, or it may be a general designational expression, which is evaluated as stated in paragraph 3.6.3.2, below. If a simple designational expression is formed from a designational expression in parentheses, the latter is evaluated according to the applicable rules (see foregoing and paragraph 3.6.3.2).

3.6.3.2 General Designational Expressions

The evaluation of a general designational expression proceeds as follows. The Boolean expression contained in the IF clause is evaluated (see section 3.5, Boolean Expressions). If a logical value of TRUE results, the simple designational expression following the IF clause is evaluated, thus completing the evaluation of the general 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 expression following the delimiter ELSE can be a general designational expression, the analysis of the operation of a designational expression becomes recursive, in a manner similar to that of general arithmetic and Boolean expressions. In the case of a designational expression, however, the result produced is a label.

3.6.4 The Subscript Expression of a Switch Designator

The value of the switch designator is defined only if the subscript expression produces one of the positive integer values 1, 2, 3,..., $\underline{n}$, where $\underline{n}$ is the number of entries in the switch list. (See section 5.3, Switch Declarations.) If the value of the subscript expression is of a type other than INTEGER, it is converted to type INTEGER in accordance with the rules applicable to assignment statements (see subsection 4.2.4, Types).

3.6.5 Unsigned Integers as Labels

The use of unsigned integers as labels is not allowed in Extended ALGOL.

4.0 Statements

4.0.1 Syntax

```
1  ⟨statement⟩ ::= ⟨unconditional statement⟩|⟨conditional statement⟩|
                   ⟨for statement⟩

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

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

3  ⟨unlabelled basic statement⟩ ::= ⟨assignment statement⟩|
                                    ⟨go to statement⟩|
                                    ⟨dummy statement⟩|
                                    ⟨procedure statement⟩|
                                    ⟨I-O statement⟩|
                                    ⟨fill statement⟩
                                    ⟨stream call procedure statement⟩
```

4.0.2 Semantics

The basic constituents of an Extended ALGOL program are statements. Statements may be divided into three major groups: unconditional, conditional, and FOR 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, as 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 FOR 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 successor.

4.1 Compound Statements and Blocks

4.1.1 Syntax

```
1  ⟨compound statement⟩ ::= ⟨unlabelled compound statement⟩|
                            ⟨label⟩:⟨compound statement⟩

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

1  ⟨block⟩ ::= ⟨unlabelled block⟩|⟨label⟩:⟨block⟩

2  ⟨program⟩ ::= ⟨block⟩|⟨compound statement⟩.
```

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

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

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

4.1.2 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 4.0), it should be kept in mind that S in the above examples could itself be a compound statement or a block.

4.1.3 Semantics

A series of statements which are common to each other by virtue of their 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 5.0, Declarations), be specified to be local to the block in question. Such a declaration means that:

1) The entity represented by this identifier inside the block has no existence outside the block, and

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

An identifier occurring within a block and not declared to the block will be nonlocal to it, i.e., 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 non-local to a block must be understood recursively. Thus an identifier which is non-local to block A may or may not be non-local to the block B in which A is one statement.

4.1.3.1 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.

4.1.3.2 Disjoint Blocks

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

4.2 Assignment Statements

4.2.1 Syntax

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⟩ ←

4.2.2 Examples

Left parts:

A ←
PROCID ←

Left part lists:

A ←
Q.[30:1] ←
X ← Y ← Z ←

Assignment Statements:

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

4.2.3 Semantics

The assignment statement causes the value represented either by an expression or a string to be assigned to the variable appearing to the left of each assignment symbol. As shown in the last example, 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:

> The subscript expressions of the left part variables are evaluated from left to right.
>
> The expression on the right side of the assignment symbol is evaluated.
>
> 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.

4.2.4 Types

All the variables in the left part list must be of the same declared type; see section 5.1, Type Declarations. If these variables are of type Boolean, the value to be assigned must be that of a Boolean expression. If these variables are of type BOOLEAN, the value to be assigned must be that of a Boolean expression. If these variables are of type ALPHA, the value to be assigned must be that of a string.

If not BOOLEAN or ALPHA, the variables in a left part list are of an arithmetic type: REAL, INTEGER, or DOUBLE. The value to be assigned, then, must be that of an arithmetic expression, which in turn may also be any one of the three arithmetic types.

A difference in arithmetic types is handled as shown below.

(1) If the left part list is of type REAL, and

(a) if the expression value is of type INTEGER, the value is stored unchanged;

(b) if the expression value is of type DOUBLE, the least-significant digits are truncated and the resulting single-precision value is stored.

(2) If the left part list is of type INTEGER, and

(a) if 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;

(b) if the expression value is of type DOUBLE, the least-significant digits are truncated and the resulting single-precision value is taken to be the value of E in the transfer function ENTIER (E + 0.5); the value thus obtained is stored.

(3) If the left part list is of type DOUBLE, and

if the expression value is of type REAL or INTEGER, least-significant digits of zero are appended to the single-precision value to produce a double-precision value, which is then stored.

4.3 GO TO Statements

4.3.1 Syntax

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

4.3.2 Examples

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

4.3.3 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 section 3.6, Designational Expressions).

The normal consecutive sequence of statement execution is unaltered in the case of an undefined switch designator; see subsection 3.6.4, The Subscript Expression. 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, that is, each block must be entered at the block head so that the associated declarations can be invoked.

4.4 Dummy Statements

4.4.1 Syntax

```
1  ⟨dummy statement⟩ ::= ⟨empty⟩
```

4.4.2 Examples

```
L1:
EXIT:
```

4.4.3 Semantics

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

4.5 Conditional Statements

4.5.1 Syntax

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

1  ⟨if statement⟩ ::=  ⟨if clause⟩⟨unconditional statement⟩

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

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

4.5.2 Examples

IF Clauses:

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

IF Statements:

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

Conditional Statements:

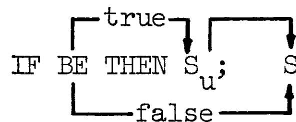
```
IF A > B THEN FOR I ← 1 STEP 1 UNTIL 5 DO R[I] ← P[I + 2]
IF A > B THEN A ← A + 1
IF GATE[1,2] AND GATE[1,3] THEN GO TO CHI ELSE
     IF GATE[1,4] AND GATE[1,5] THEN GO TO BOS ELSE GO TO ERROR1
```

4.5.3 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.

4.5.3.1 IF Statement

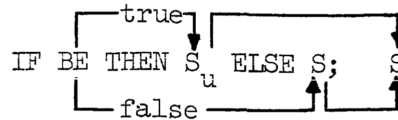
One of the permissible forms of a conditional statement is the IF statement. The IF statement operates as follows: The unconditional statement following the sequential operator THEN is executed if the logical value of the preceding Boolean expression is TRUE, otherwise the statement is ignored.



4.5.3.2 IF... THEN...ELSE Statement

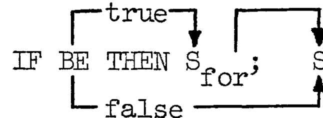
A second form of a conditional statement contains the sequential operator ELSE. The operation of this conditional statement proceeds as follows: If the logical value produced by the Boolean expression is TRUE, the unconditional statement following the sequential operator THEN is executed and the statement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

ELSE is executed. This latter statement could also be another conditional statement.



4.5.3.3 IF...FOR Statement

The third allowable form of the conditional statement specifies a FOR statement after the IF clause. It is completely analogous to the first form (see paragraph 4.5.3.1).



In the most general case, a conditional statement can be a series of conditions and the evaluation continues until a logical value of TRUE is found. When this occurs, the next succeeding unconditional statement is executed. If none of the Boolean expressions has a logical value of TRUE, the statement following the rightmost ELSE is executed. If no ELSE appears after the rightmost THEN, control continues in sequence.

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.

4.6 FOR Statements

4.6.1 Syntax

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⟩

4.6.2 Examples

For List Elements:

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

For Lists:

```
A + 2

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

FOR Clauses:

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

FOR Statements:

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

4.6.3 Semantics

A FOR statement provides a method of forming loops in a program. It allows for the repetitive execution of a statement zero or more times.

The FOR statement can best be understood by isolating three distinct operational steps:

1) value assignment to the controlled variable
2) test of limiting condition
3) execution of the statement following DO.

Each type of for list describes a different process and therefore will be discussed separately. All, however, have one property in common: The initial value assigned to the controlled variable is that of the leftmost arithmetic expression in the for list element.

4.6.4 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.

4.6.4.1 Arithmetic Expression Element

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. There is no limiting condition, therefore no test is made. After assigning the initial value to the controlled variable the statement following DO is executed. The element then is exhausted. A concise description is:

V ← AE;
S_{do};
S;

4.6.4.2 STEP-UNTIL Element

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 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 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 ≤ AE3) OR (SIGN(AE2) = -1 AND V ≥ AE)
          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.

4.6.4.3 WHILE Element

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;
```

4.6.4.4 STEP-WHILE Element

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, that of AE1, is assigned to the controlled variable. All subsequent assignments are: V ← V + AE2, and are 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:

V ← AE1;
L3: IF BE THEN BEGIN S_{do}; V ← V + AE2; GO TO L3 END

4.6.5 Value of the Controlled Variable Upon Exit from the FOR Statement.
Upon exit out of the statement S (supposed to be compound) through a GO TO statement, the value of the controlled variable will be the same as it was immediately preceding the execution of the GO TO statement.

If the exit is due to exhaustion of the for list, on the other hand, the value of the controlled variable is not accessible after the exit.

4.6.6 GO TO Leading Into a FOR Statement

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

4.7 Procedure Statements

4.7.1 Syntax

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

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 identifier⟩|
⟨switch identifier⟩|⟨procedure identifier⟩|
⟨file identifier⟩|⟨format identifier⟩|
⟨list identifier⟩|⟨array row⟩

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

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

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

3 ⟨row designator⟩ ::= * | ⟨row⟩, *

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

4.7.2 Examples

ALGORITHM123 (A + 2)

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

4.7.3 Semantics

A PROCEDURE statement causes a procedure body which has been previously defined by a PROCEDURE declaration (see section 5.11, PROCEDURE Declarations) to be activated (called for execution). It may also cause the activation of a stream block which has previously been defined by a STREAM PROCEDURE declaration. The following discussion concerns conventional procedure calls only; stream procedure calls are discussed in section 5.12.

The procedure identifier references the procedure body which is to be executed. The actual parameter part contains a list of the actual parameters that are 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:

1) The formal parameters which are named in the value part of the PROCEDURE declaration are assigned the values currently represented by the corresponding actual parameters. These formal parameters will subsequently be treated as local to the procedure body.
2) The formal parameters not named in the value part 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.
3) 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.

4.7.3.1 Value Assignment (Call by Value)

The actual parameters that may be called by value are expressions and array identifiers. Expressions which are called by value fall into several categories: simple variables, subscripted variables, partial word designators, and function designators; and arithmetic, Boolean, and designational expressions.

In the case of the simple variable, the assignment of its value is straightforward. Where a subscripted variable is an actual parameter, the subscript expression is evaluated and the appropriate element of the array is assigned to the corresponding formal parameter.

If a partial word designator is given as an actual parameter, the field designated is assigned to the associated formal parameter in accordance with the rules governing the operation of partial word designators (see section 3.2, Partial Word Designators). Where a function designator has been listed as an actual parameter, the function is evaluated and the resulting value is assigned to the corresponding formal parameter. In the situation where an arithmetic, Boolean, or designational expression is given as an actual parameter, the expression is evaluated according to the rules previously defined (see sections 3.4, 3.5, and 3.6), and the resulting value is assigned to the appropriate formal parameter.

In the case of an array identifier, an array local to the procedure is created which is the same in all respects as the actual array. All values of the actual array are then assigned to the corresponding elements of the new array. When an array identifier is given as an actual parameter, the corresponding formal parameter must be used as an array identifier (see section 5.11, PROCEDURE Declarations).

The evaluation of the actual parameters, and their subsequent assignment to the corresponding formal parameters, takes place according to the order indicated by the value part of the procedure declaration.

These assignments take place before entry is made into the procedure body.

4.7.3.2 Name Replacement (Call by Name)

The actual parameters that may be called by name are expressions and array, switch, procedure, file, format, and list identifiers. The action taken in a call by name differs somewhat from that in a call by value. Instead of a value's being assigned, the actual expression or pertinent identifier of the actual parameter replaces the corresponding formal parameter wherever it appears in the procedure body. As in the case of call by value, a detailed analysis of this mechanism requires that each kind of actual parameter allowed 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.

Where 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. The partial word designator is referenced each time it is encountered during the execution of the procedure body. The corresponding formal parameter 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 whenever it is encountered during the course of execution of the procedure body.

In the case where 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 whenever 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. In this case, no local array is created, and any appearance of the formal parameter in the procedure body makes reference to the actual array designated by the array identifier.

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 actual parameter, and (3) makes the corresponding actual parameter therafter inaccessible to the procedure (unless the procedure is called again, of course). 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 can therefore have its value altered.

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

Where a procedure identifier has been given 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 a procedure which has been declared outside the procedure body. However, a STREAM PROCEDURE identifier must not be used as an actual parameter.

Where a file, format, or list identifier has been given 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. I-O statements in a procedure body can thus utilize files, formats, and lists which have been declared outside the procedure body being executed.

4.7.4 Restrictions

In order that the PROCEDURE statement operate correctly, it is necessary that the call by value and call by name mechanisms be capable of being carried out. This means that the formal and actual parameters must correspond in type and in kinds of quantities. In addition, 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. *Up level addressing not yet available*

4.8 I-O Statements

The I-O statements cause values to be communicated to and from the program. These values are introduced through an input process and assigned to designated variables, or are results which the programmer desires to include in an output process.

4.8.1 Syntax

3 ⟨I-O statement⟩ ::= ⟨read statement⟩ | ⟨write statement⟩ |
⟨release statement⟩

4.8.2 READ Statement

4.8.2.1 Syntax

3 ⟨read statement⟩ ::= READ (⟨input parameters⟩)

3 ⟨input parameters⟩ ::= ⟨file identifier⟩,⟨format identifier⟩,
⟨list⟩ | ⟨file identifier⟩,⟨format identifier⟩,
⟨list 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⟩ ::= ⟨expression part⟩ | ⟨expression list⟩,
                              ⟨expression part⟩ | ⟨list segment⟩ |
                              ⟨expression list⟩, ⟨list segment⟩

4.8.2.2 Examples

```
READ (FILE1, EDIT2, LIST4)
READ (F3, EDIT3, X, Y, A[3])
READ (FIL4, FORM, FOR I ← 0 STEP 1 UNTIL 10 DO A[I], X, Z)
```

4.8.2.3 Semantics

The READ statement assigns values to one or more variables through an input process.

Since the B 5000 interrogates all input media, each file physically present introduces itself, in effect, to the system. For this reason, programs never need be altered to accomodate changes in file identification or input medium.

The input parameters of a READ statement specify the file identifier, the format, and the variables to be initialized. The file identifier (see section 5.6, File Declarations) in a program is associated with the corresponding physical file by means of a program parameter card. Thus, neither the kind of input equipment nor its physical unit number is ever specifically given in an Extended ALGOL program for the B 5000.

The format identifier refers to a format declaration (see section 5.7, Format Declarations) which defines the editing specifications for the incoming information.

The list indicates the variables that are to be assigned values. The variables may be contained in the READ statement, or may be referenced by a list identifer (see section 5.8, List Declarations), or both. Since a list declaration may be referred to in both READ and WRITE statements, the syntactical definition of a list includes constructs which are meaningless in an input process. An input list must contain variables only.

4.8.3 WRITE Statement

4.8.3.1 Syntax

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

3 ⟨output parameters⟩ ::= ⟨file identifier⟩ ⟨format and list parameters⟩

3 ⟨format and list parameters⟩ ::= , ⟨format and list part⟩ | ⟨empty⟩ |
[⟨carriage control⟩] |
[⟨carriage control⟩],⟨format and list part⟩

3 ⟨format and list part⟩ ::= ⟨format identifier⟩ | ⟨format identifier⟩,⟨list⟩

3 ⟨carriage control⟩ ::= ⟨skip to next page⟩ | ⟨skip to channel⟩ |
⟨double space⟩ | ⟨no space⟩

3 ⟨skip to next page⟩ ::= PAGE

3 ⟨skip to channel⟩ ::= ⟨unsigned integer⟩

3 ⟨double space⟩ ::= DBL

3 ⟨no space⟩ ::= NO

4.8.3.2 Examples

```
WRITE (FILE2, EDIT2, LIST4)
WRITE (F2[DBL], EDIT5, X, Y, A[I])
WRITE (FILE1[PAGE])
WRITE (F5, EDIT4, FOR I ← 0 STEP 1 UNTIL 10
       DO [A[I], B[I]], X, LIST2)
```

4.8.3.3 Semantics

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

Since several kinds of output equipment are available and various formating alternatives are provided, it is necessary to specify both when calling for an output operation. The output parameters of a WRITE statement serve to specify these as well as the information that is to be written (punched, printed, recorded on magnetic tape, etc.).

The file identifier (see section 5.6, File Declarations) is used in a manner

analogous to its use in the READ statement (see paragraph 4.8.2.3, Read Statement, Semantics).

Carriage control may be included to allow for paper control on the line printer. If the specified output equipment is other than the line printer, carriage control is irrelevant and is ignored.

The format identifier refers to a format declaration (see section 5.7, Format Declarations) which defines the editing specifications desired for the information. It may also specify messages to be included. If such a specified message constitutes the entire output, no list is needed.

The list specifies the values that are to be included in the output. The values may be included in the WRITE statement or referenced by a list identifier (see section 5.8, List Declarations) or both. An output list may contain any arithmetic or Boolean expression.

4.8.4 RELEASE Statement

4.8.4.1 Syntax

3 ⟨release statement⟩ ::= RELEASE (⟨file identifier⟩⟨word count⟩)

3 ⟨word count⟩ ::= ⟨empty⟩ | , ⟨arithmetic expression⟩

4.8.4.2 Examples

RELEASE (FILE3)

RELEASE (F1, N)

4.8.4.3 Semantics

The RELEASE statement causes the information contained in a buffer area associated with a file identifier (see section 5.6, File Declarations) to be released by the program.

If the file identifier is that of an input file, the RELEASE statement causes the buffer to be filled with new information from the applicable input device.

If the file identifier is that of an output file, the RELEASE statement causes the information contained in the buffer to be transferred to the applicable output device.

The number of words transferred is determined by buffer size in the File Declaration until a RELEASE statement associated with that file indicates an actual word count. Subsequently, the number of words transferred is determined by the last indicated word count.

4.9 FILL Statement

4.9.1 Syntax

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

4.9.2 Examples

FILL MATRIX[*] WITH 458.54, +546, -1354.543@6, 16@-12

FILL GROUP [1,*] WITH .25, "ALGOL", """, OCT14, "365"

4.9.3 Semantics

The FILL statement causes one row of an array to be filled with a list of specified values.

4.9.3.1 Row Designator

The row designator indicates which row is to be filled by designating a specific value for each subscript position except the rightmost, in which position the symbol * appears. If the value of a row expression is other than INTEGER, it is converted to type INTEGER, in accordance with the rules applicable to assignment statements (see subsection 4.2.4, Types).

4.9.3.2 Value List

Initial values have three forms, and a value list may contain a mixture of these forms. The concept of type does not apply to initial values, therefore no transfer functions are invoked because of the declared type of the array being filled.

A number is converted to its octal equivalent, then stored. A string causes the 6-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 8 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 more, the rightmost values in the value list are not used.

4.9.3.3 Restrictions

A defined identifier must not be used in a FILL statement.
There must be no space between OCT and the octal number which follows.

4.10 Stream Call Procedure Statement

4.10.1 Syntax

```
⟨stream call procedure statement⟩ ::= ⟨stream procedure identifier⟩
                                      ⟨stream actual parameter part⟩
⟨stream procedure identifier⟩ ::= ⟨identifier⟩
⟨stream actual parameter part⟩ ::= (⟨stream actual parameter list⟩)
⟨stream actual parameter list⟩ ::= ⟨stream actual parameter⟩|
                                   ⟨stream actual parameter list⟩,
                                   ⟨stream actual parameter⟩
⟨stream actual parameter⟩ ::= ⟨stream value parameter⟩ | ⟨stream name parameter⟩
⟨stream value parameter⟩ ::= ⟨arithmetic expression⟩ | ⟨Boolean expression⟩
⟨stream name parameter⟩ ::= ⟨array identifier⟩ | ⟨array row⟩ |
                            ⟨file identifier⟩ | ⟨variable⟩
⟨array row⟩ ::= ⟨array identifier⟩ [ ⟨row designator⟩ ]
```

4.10.2 Examples

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

4.10.3 Semantics

A stream call procedure statement causes the actual parameters to be supplied to the stream procedure and then transfers control to the stream procedure body. A stream procedure must have an actual parameter part; it may not be empty.

A one-to-one correspondence must exist between the actual and formal parameters. The formal parameters are either call by name or call by value. The actual parameters are therefore in two classes: those actual parameters whose corresponding formal parameters are in the value part of the stream procedure declaration ⟨stream value parameter⟩, and those actual parameters which correspond to call-by-name formal parameters ⟨stream name parameter⟩.

4.10.4 Value

Stream value parameters may only be arithmetic or Boolean expressions. The corresponding formal parameters are supplied the value of the stream actual parameter when the stream call procedure statement is executed.

4.10.5 Name

A stream name parameter may be array and file identifiers, variables, and array rows. When the stream call procedure statement is executed an absolute address is supplied to the corresponding formal parameters.

If a stream name parameter is a file identifier, 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.

Arrays are mapped in memory by rows. Elements of a row are contiguous but rows are not.

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

1. For a single dimensional array the address is that of the lowest element of the array.

2. For a multi-dimensional array the address is that of the lowest element of the highest level dope-vector.

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

4.10.6 Restrictions:

Designational expressions, switch identifiers, list identifier, formal identifiers, and call-by-name expressions are not allowed as actual parameters to stream procedures.

5.0 Declarations

5.0.1 Syntax

3 ⟨declaration⟩ ::= ⟨type declaration⟩ | ⟨array declaration⟩ |
⟨switch declaration⟩ | ⟨label declaration⟩ |
⟨define declaration⟩ | ⟨file declaration⟩ |
⟨format declaration⟩ | ⟨list declaration⟩ |
⟨diagnostic declaration⟩ | ⟨forward reference declaration⟩ |
⟨procedure declaration⟩ | ⟨stream procedure declaration⟩

5.0.2 Semantics

The purpose of a declaration in a program 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. Then, at the time of exit from a block (through END, or by a GO TO statement), all identifiers which were 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 two block heads of nested blocks. The conflict is resolved as follows.

Assume: Block A — outer block
Block B — inner block

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

Apart from the identifiers associated with the standard functions (see subsections 3.3.4, Standard Functions, and 3.3.5, Type Transfer Functions, all identifiers of a program must be declared. An identifier may not be declared to represent more than one quantity in a single block head.

5.1 Type Declarations

5.1.1 Syntax

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

5.1.2 Examples

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

5.1.3 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.

5.1.4 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 value. The values of variables not declared OWN are undefined upon reentry into the block, and those variables must be initialized again.

5.1.5 Type

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

REAL	Positive and negative values including zero
INTEGER	Positive and negative integral values including zero
BOOLEAN	Logical values of TRUE and FALSE
DOUBLE	Double precision values of type REAL
ALPHA	Any set of six or fewer characters not including ⟨?⟩

5.2 ARRAY Declarations

5.2.1 Syntax

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⟩

5.2.2 Examples

Bound Pair Lists:

1:10

1:10,3:9

A + 2:B + 4

IF B1 THEN A + K ELSE A + I:IF B2 THEN B + K ELSE B + I

Array Segments:

MATRIX [1:10]

MATRIX, GROUP [1:10]

Array Lists:

MATRIX [1:10]

MATRIX, GROUP [1:10], GATE [1:10, 3:9]

Array Declarations:

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

5.2.3 Semantics

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

5.2.3.1 SAVE Arrays

The declarator SAVE causes an area of core memory to be reserved for the declared array. If this area must be overlaid, the array will be placed back into the same area in memory when the array is again referenced. This absolute storage allocation is necessary only when an array is being used in conjunction with a STREAM PROCEDURE (see section 5.12, STREAM PROCEDURE Declarations). In order to maintain the validity of the stream address indexes upon exit and reentrance to the STREAM PROCEDURE, the location of the referenced area must remain constant. Whenever the SAVE declarator is used inefficient allocation of core storage may result.

5.2.3.2 Local or OWN

An array may be declared as OWN with the same effect as that given for simple variables (see section 5.0, Declarations, and section 5.1, 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.

5.2.3.3 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. Arrays which are declared together must be of the same type.

5.2.4 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 of expressions and are evaluated in the same manner as subscript expressions (see paragraph 3.1.5.2, Subscript Expressions). 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. It follows that arrays declared in the outermost block must use constant bounds.

An array is defined only when the values of all upper bounds are not smaller than those of the corresponding lower bounds. In addition, no dimension may contain more than 1023 elements. 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.

5.3 SWITCH Declarations

5.3.1 Syntax

1 ⟨switch declaration⟩ ::= SWITCH ⟨switch identifier⟩ ← ⟨switch list⟩

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

5.3.2 Examples

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

5.3.3 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 section 3.6, 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.

5.3.4 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.

5.3.5 Influence of Scope

If a switch designator occurs outside the scope of a quantity entering into a designational expression in the switch list, and an evaluation of this switch designator selects this designational expression, the quantity which is otherwise inaccessible is used for the evaluation of the selected designational expression.

5.4 DEFINE Declarations

5.4.1 Syntax

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

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

3 ⟨definition⟩ ::= ⟨defined identifier⟩ = ⟨well-formed construct⟩#

3 ⟨defined identifier⟩ ::= ⟨identifier⟩

3 ⟨well-formed construct⟩ ::= ⟨basic component set⟩

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

5.4.2 Examples

DEFINE RK = RUNGEKUTTA#, ROOT = (−B + SQRT(B*2−4×A×C))/(2×A)#

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

5.4.3 Semantics

The DEFINE declaration serves to define an identifier as a set of basic Extended ALGOL components. Any basic component used in a definition must be completely contained therein. All identifiers in the definition must have been previously declared and the meaning thereby given is always employed whenever the defined identifier is used.

The appearance of a defined identifier results in it being replaced by its associated definition. Such replacement must result in a legitimate Extended ALGOL construct.

5.4.4 Influence of Scope

If a defined identifier appears in a nested block with respect to its scope, any redeclaration of an identifier which was in its definition does not alter that definition and quantities otherwise inaccessible may thereby be referenced.

5.4.5 Restrictions

The well-formed construct of a definition must not contain a declarator or a specificator. A defined identifier must not be used in a FILL statement, a FORMAT declaration nor as an actual parameter.

5.5 LABEL Declarations

5.5.1 Syntax

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

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

2 ⟨label⟩ ::= ⟨identifier⟩

5.5.2 Examples

```
LABEL START
LABEL ENTER,EXIT,START,LOOP
```

5.5.3 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 innermost 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.

A PROCEDURE body itself must not be labeled.

5.6 FILE Declarations

5.6.1 Syntax

3 ⟨file declaration⟩ ::= FILE ⟨I-O part⟩⟨file part⟩

3 ⟨file part⟩ ::= ⟨file identifier⟩⟨buffer part⟩⟨I-O unit control⟩
⟨file control part⟩)|
⟨file part⟩,⟨file identifier⟩(⟨buffer part⟩⟨I-O unit control⟩
⟨file control part⟩)

3 ⟨I-O part⟩ ::= IN|OUT|REVERSE

3 ⟨file identifier⟩ ::= ⟨identifier⟩

3 ⟨buffer part⟩ ::= ⟨number of buffers⟩,⟨buffer size⟩

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

3 ⟨buffer size⟩ ::= ⟨unsigned integer⟩

3 ⟨I-O unit control⟩ ::= ⟨empty⟩|,0|,1

3 ⟨file control part⟩ ::= ⟨empty⟩|[⟨disposition⟩⟨blocking⟩]⟨end of file⟩|
[⟨disposition⟩⟨blocking⟩]⟨save factor⟩

3 ⟨disposition⟩ ::= 0|1|2|3⟨simple variable⟩

3 ⟨blocking⟩ ::= ,⟨blocking option⟩ ⟨records per block⟩|⟨empty⟩

3 ⟨blocking option⟩ ::= 0|1|2|3

3 ⟨records per block⟩ ::= ⟨empty⟩|,⟨unsigned integer⟩

3 ⟨end of file⟩ ::= ⟨label⟩|⟨empty⟩

3 ⟨save factor⟩ ::= ⟨unsigned integer⟩|⟨empty⟩

5.6.2 Examples

```
FILE IN F0 (2,500),F1(1,10)
FILE IN F2 (2,1000[1,1,100] EOF)
FILE OUT F3 (2,15,1)
FILE OUT F4 (1,100[1,1,25])
FILE OUT F5 (2,100[0,3]30)
FILE REVERSE F2 (2,1000,1[2,1,100]FEND)
FILE REVERSE F4 (1,100,0[3,1,25])
```

5.6.3 Semantics

The file declaration associates a file identifier with the specifications which govern the handling of a file.

5.6.3.1 Buffer Part

The buffer part specifies the size (number of B 5000 words) needed for a buffer area, and the number of buffer areas desired.

The information in one punched card interpreted in the alpha mode requires a buffer of 10 words. A buffer of 15 words is required for one line of print on the line printer. A variable number of words may be contained in one magnetic tape block, but must not exceed 1023.

If more than one buffer is specified and storage is inadequate to accommodate the number designated, the program is automatically run with a lesser number.

5.6.3.2 I-O Unit Control

The maximum number of I-O units required at any one time in a program is indicated, according to type, in the program parameter card. When the program has passed the point requiring the maximum number, an I-O unit may be released to the system for use by another program by means of the appropriate I-O unit control in the File Declaration.

If the I-O unit control is empty or 0, the I-O unit will remain reserved for use by the program. If the I-O unit control is 1, the I-O unit will be released to the system for use outside the program.

When the disposition is 0 or 1 for a magnetic tape file, the fact that the I-O unit must be retained by the program is implied. Therefore, I-O unit control is automatically given a value of 0 in all such cases.

5.6.3.3 Disposition

The disposition digit designates the action to be taken when exit is made from the block in which the file declaration appears, and is meaningful for magnetic tape files only. If the file is not on magnetic tape, the disposition digit is ignored. The disposition digits have the meanings shown below:

- 0: Rewind the tape for further use by the program.
- 1: Do not rewind the tape for further use by the program.
- 2: Rewind and lock the tape for removal and retention.
- 3: Rewind and release the tape to the system for other use.

If the file control part is empty, or if the disposition variable has a value other than 0, 1, 2, or 3, the action indicated by digit 0 is carried out.

5.6.3.4 Blocking

Blocking specifies the method to be used in blocking magnetic tape records. The digits which control blocking have the meanings shown below.

- 0: No blocking is to be performed; each block contains one record, therefore records per block should be empty or 1.
- 1: Each block contains a number of fixed-length records as indicated by records per block.
- 2: Each block contains a number of variable-length records as indicated by records per block.
- 3: Each block contains a variable number of records and each record is of variable length, therefore records per block should be empty.

If no blocking option digit is present, the action specified for digit 0 is carried out.

5.6.3.5 End-of-File

End-of-file is applicable to input files only, and specifies the labeled statement to which control is transferred when the entire file has been exhausted. If the file control part is empty and the end of the file is encountered, con-

trol continues in sequence; that is, the statement following the READ statement will be executed.

5.6.3.6 Save Factor

The save factor is applicable to output magnetic tape files only, and represents the number of days to be added to the current date to give the purge date. If the file control part is empty, a save factor of zero will be added. If the file is not on magnetic tape, the save factor is ignored.

5.6.3.7 FILE REVERSE

A FILE REVERSE declaration applies to input magnetic tape files only, and specifies that information is to be read in the reverse direction.

Before a file identifier may be declared REVERSE, it must be declared IN or OUT in an earlier disjoint block.

5.6.3.8 Scope

The scope of a file identifier differs from that of other identifiers in a program. All file identifiers in a program must be unique. An identifier is declared to be a file identifier in the program parameter card. Therefore file identifiers designate specific physical files throughout the program regardless of the block structure of the program.

The file declaration does not declare an identifier to be a file identifier, but declares a method of file handling to be applied to the file designated by the file identifier. The block structure thus specifies the scope of a file handling method.

5.6.3.9 Restrictions

A program may contain more than one file declaration involving the same file identifier, but each such declaration must appear in a block which is disjoint from the others.

A file identifier may designate a file on a multiple-file magnetic tape (as indicated in the program parameter card). More than one such file may be used

in a program; each file declaration involving these files, however, must appear in a block which is disjoint from the others.

5.7 FORMAT Declarations

5.7.1 Syntax

```
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⟩,⟨repart 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|X
3 ⟨field part⟩ ::= ⟨empty⟩|⟨field width⟩|⟨field width⟩.⟨decimal places⟩
3 ⟨field width⟩ ::= ⟨unsigned integer⟩
3 ⟨decimal places⟩ ::= ⟨unsigned integer⟩
```

5.7.2 Examples

```
FORMAT IN EDIT (X4,2I6,5E9.2,3F5.1,X4)
FORMAT F1 (A6, 5(X3, 2E10.2, 2F6.1), 3I7), F2 (3D,930,4D)
FORMAT OUT FORM1 (X56,"HEADING",X57),FORM2 (X10,4A6/X7,5A6/X2,5A6)
FORMAT OUT F3 (1023 O)
```

5.7.3 Semantics

The format declaration associates a set of editing specifications with a format identifier. The discussion of format declarations is divided into two parts: those used for input and those used for output.

5.7.3.1 Input Editing Specifications

Input data can be introduced to the B 5000 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 of two ways: either as a set of 6-bit characters (see Appendix A, showing internal codes for characters), or as a set of 48-bit B 5000 words. 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.

5.7.3.2 Input Editing Phrases

Editing phrase types D and O cause the input string to be processed as 48-bit B 5000 words. Other editing phrase types designate six-bit character processing. The editing specifications associated with one format identifier must designate only one of the two methods of processing. Characteristics of the input editing phrase types are summarized in the following table.

Editing Phrase Type	Editing Phrase Example	Processed As	Type of Variable Being Initialized	Example of Field Contents
A	A6	6-bit characters	ALPHA	TX@95%
D	D	48-bit words	None	Any 48 bits
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	48-bit words	Any type but DOUBLE	Any B 5000 operand
X	X7	6-bit characters	None	Any 7 characters

The definition of each input editing phrase type is given below.

A - Initializes a variable to the characters found in the field described by ⟨field width⟩. If ⟨field width⟩ is greater than six, the rightmost six characters are taken as the value to be assigned to the variable. If ⟨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.

D - Causes one B 5000 word of 48 bits in the input data string to be ignored.

E - Initializes a variable to the number found in the field described by ⟨field width⟩. ⟨Field width⟩ must be at least 7 greater than ⟨decimal places⟩, since the input data is required to be of the following form:

± 0.dd---d@±ee

The sign of the number must appear first. A zero and a decimal point must follow the sign. One or more digits may follow the decimal point. The number of digits must equal ⟨decimal places⟩ in the editing phrase. Following the digits must be the symbol @, the sign of the exponent, and a two-digit exponent.

The sign may be indicated by +, -, or a single space which is interpreted as positive. The number must be right-justified in the designated field.

F - Initializes a variable to the number found in the field described by field width. The input data must be in one of the following forms:

± nn---n.dd--d

± .dd--d

The sign of the number must appear first. A decimal point must be present; zero or more digits may appear between it and the sign. Finally there must be one or more digits after the decimal point; the number of these digits must equal ⟨decimal places⟩ in the editing phrase. Rules for indicating the sign and for justification of the number are the same as for editing phrase type E.

I - Initializes a variable to the integer found in the field described by ⟨field width⟩. The sign of the integer must appear first, followed by one or more digits. The sign and justification conventions given for editing phrase type E are applicable.

L - Initializes a variable to the logical value found in the field described by ⟨field width⟩. There are two possible values, TRUE and FALSE; these must be right-justified in the field.

O - Initializes a variable to the contents of one B 5000 word of the input string. The field part is ignored and should be empty.

X - Causes the number of characters indicated by ⟨field width⟩ to be ignored.

An input editing phrase must not be a string; the string form is defined for output only.

Each editing phrase, except the D and X types, describes a portion of the input data string in which the initial value of one variable is to be found. A phrase such as 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.

The field part should be empty when the editing phrase type is O or D. If the editing phrase type is not O or D, the field part must not be empty.

5.7.3.3 Output Editing Specifications

Output can be performed by the B 5000 through various media such as magnetic tape and 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 A), or from a set of 48-bit B 5000 words. The output editing specifications, through the editing phrases, designate where and in what form the values of expressions are to be placed in this string.

5.7.3.4 Output Editing Phrases

Editing phrase types D and O designate that the output string be built from 48-bit B 5000 words. Other editing phrase types specify that the output string

be built from six-bit characters. The editing specifications associated with one format identifier must designate only one of the two building processes. Characteristics of the output editing phrase types are summarized in the following table. The letter b represents a blank.

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	48-bit words	None	48 zero bits
E	E11.4	6-bit characters	REAL	-0.0125@+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	48-bit words	Any type but DOUBLE	Any B 5000 operand
X	X8	6-bit characters	None	8 blanks

Each output editing phrase type is defined below.

A - Places the value of one expression, in this case six characters, in the field described by ⟨field width⟩. If ⟨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 ⟨field width⟩ is less than six, the rightmost characters of the expression value are placed in the field.

D - Places one 48-bit B 5000 word of all zeros in the output data string.

E - Places the value of one expression in the field described by ⟨field width⟩. This value has the following form when placed in the output data string:

±0.dd---d@±ee

If ⟨field width⟩ is more than 7 greater than ⟨decimal places⟩, leading single spaces are used to complete the field. Then the sign of the number, a zero, and a decimal point are inserted. The value of the expression is rounded to the number of places indicated by ⟨decimal places⟩ of the editing phrase. If the number of significant digits in the expression value is less than ⟨decimal places⟩, 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 number is represented by a single space if positive, and a minus sign if negative. The sign of the exponent is either + or -.

F — places the value of one expression in the field described by ⟨field width⟩. This value has the following form when placed in the output string:

-nn---n.dd--d

The expression value is rounded to the number of designated decimal places. If the number of significant digits thus obtained is less than ⟨field width⟩ minus two, leading single spaces are used to complete the field. If the digits number more than ⟨field width⟩ minus two, most-significant digits are lost. The sign of the number and the significant digits with an appropriately placed decimal point complete the field.

The sign of the number is the same as for the E editing phrase type.

I — places the value of one expression in the field described by ⟨field width⟩. The expression value is rounded to an integer and placed right-justified in the field, preceded by the sign of the number and leading single spaces, if any.

The sign of the number is the same as for the E editing phrase type.

Most-significant digits are lost if the expression value contains more digits than ⟨field width⟩ minus one.

L - Places the value of one Boolean expression in the field designated by ⟨field width⟩. Since the expression yields a logical value, either TRUE or FALSE will be placed in the output string, therefore ⟨field width⟩ should be 5 or more. If ⟨field width⟩ is greater than the number of characters in the logical value, leading single spaces are inserted to fill out the field.

O - Places the value of one expression, in B 5000 48-bit form, in the output string. The field part is ignored and should be empty.

X - Places a number of single spaces, as indicated by ⟨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.

Each editing phrase describes a portion of the output data string into which information is to be placed. This information may be one of three things: the value of an expression, the characters of the editing phrase itself (when the editing phrase is a string), or the insert characters 0 and single space.

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.

The field part should be empty when the editing phrase type is O or D. If the editing phrase type is not O or D, the field part must not be empty.

5.8 LIST Declarations

5.8.1 Syntax

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⟩ ::= ⟨expression part⟩ | ⟨list segment⟩ | ⟨expression list⟩, ⟨expression part⟩ | ⟨expression list⟩, ⟨list segment⟩

5.8.2 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]]])

5.8.3 Semantics

A list declaration serves to associate a set of expressions (arithmetic or Boolean) with a list identifier.

The list identifier may be used in a READ statement (section 4.8.2) for specifying the variables to be initialized and the order in which the initializing is to be done. Since any expression other than a variable is meaningless in an input operation, a list identifier used in a READ statement must refer to a LIST declaration which includes variables only. When used for input, the variables in a LIST declaration must be of type REAL, INTEGER, BOOLEAN, or ALPHA.

The list identifier may be used in a WRITE statement (section 4.8.3) 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 non-local to the block in which the LIST declaration appears.

5.9 FORWARD Reference Declarations

5.9.1 Syntax

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

5.9.2 Examples

SWITCH SELECT FORWARD

INTEGER PROCEDURE SUM (A,B,C); VALUE A,B,C; INTEGER A,B,C; FORWARD

5.9.3 Semantics

Before a procedure or a switch can be called in a program, it must have been declared previously. Two cases of interest arise: (1) a procedure which calls another procedure, which in turn calls the first procedure; (2) a switch which references another switch, which in turn references the first switch.

The use of a FORWARD reference declaration does not eliminate the need for the normal PROCEDURE and SWITCH declarations which must follow as soon as possible in the program.

5.10 Diagnostic Declarations

5.10.1 Syntax

3 ⟨diagnostic declaration⟩ ::= MONITOR ⟨monitor part⟩|
DUMP ⟨dump 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⟩

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⟩
```

5.10.2 Examples

```
MONITOR ANSWER (A,Q[I,J],GROUP1,START,SELECT,INTEGRATE)
DUMP INPUTDATA (A,Q[I,J],GROUP1,START) ENTER:4,
     OUTPUTDATA (A,GROUP1) EXIT:X
```

5.10.3 Semantics

The diagnostic declarations MONITOR and DUMP declare certain quantities to be placed under surveillance during the execution of the program, and cause the identifiers for these quantities and their associated values to be released to an output device under specified conditions.

5.10.4 MONITOR

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.

5.10.4.1 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 leftmost 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 subscripted variable, with an array identifier which is given in the monitor list, is encountered as the leftmost 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 identifier is written on the designated file.

5.10.5 DUMP

Diagnostic information requested by means of the DUMP declaration is written on the designated file when a statement, carrying a label which is in the dump list, has been executed a number of times equal to the 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 to only one pass through the DUMP declaration block. The number is not cumulative from one pass to the next.

5.10.5.1 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 element}
{value of second element}
.
.
.
{value of last element}

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 rightmost 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 rightmost 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.

A label in the dump list causes a tally to be supplied which represents the number of times the labeled statement has been executed during this pass through the block containing the DUMP declaration. The tally is supplied in the following form:

⟨label⟩ {number of times statement has been executed}

5.11 PROCEDURE Declarations

5.11.1 Syntax

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

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⟩

5.11.2 Example

```
PROCEDURE ROOT (A, B, C, N, X1, X2, X3);
     VALUE A, B, C, 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 × A[I] × C[I];
                    IF DISC < 0 THEN X3[I] ← "IMAG" ELSE
                         BEGIN X1[I] ← (-B[I] + SQRT (DISC))/(2 × A[I]);
                              X2[I] ← (-B[I] - SQRT (DISC))/(2 × A[I]);
                              X3[I] ← "REAL" END
               END
     END ROOT
```

5.11.3 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 section 4.7, PROCEDURE Statements).

A PROCEDURE declaration is composed of two parts: the procedure heading and the procedure body.

5.11.3.1 Procedure Heading

The procedure heading contains the identifier for the procedure and information about the formal parameters.

The formal parameter part contains a list 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 parameter list. Those formal parameters not in the value part are called by name. The specification part indicates certain characteristics of the formal parameters, that is, the kinds of identifiers they represent.

If formal parameter is VALUE INTEGER actual parameter is REAL it will not be integerized

In the case of formal parameters used as array identifiers, information about the lower bounds may be given. An integer specified lower bound 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.

5.11.3.2 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 subsection 4.0.1 (covering the syntax of statements) and therefore may be a procedure statement calling upon itself. Procedures may thus be called recursively.

5.11.3.3 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 non-local 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.

5.11.4 Values of Function Designators

Certain procedures are called by means of function designators. In such cases, the procedure declaration must start with a type declarator. In addition, the procedure body must contain, and cause to be executed, an assignment statement with the procedure identifier in the left part list.

5.11.5 Restrictions

In using a GO TO statement in a typed procedure, the following restriction must be observed: No GO TO statement appearing in a typed procedure may lead outside that procedure.

Also, in using a procedure statement in a typed PROCEDURE, any PROCEDURE thereby called for execution must not contain a GO TO statement leading outside of itself.

If any statement in a PROCEDURE body is labelled, the declaration of that label must appear within the PROCEDURE body. A PROCEDURE body itself must not be labelled.

5.12 STREAM PROCEDURE Declarations

5.12.1 Syntax

```
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⟩)

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

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

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

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

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

2  ⟨label⟩ ::= ⟨identifier⟩

3  ⟨stream variable list⟩ ::= ⟨stream simple variable⟩|
                              ⟨stream variable list⟩,
                              ⟨stream simple variable⟩

3  ⟨stream simple variable⟩ ::= ⟨variable identifier⟩

1  ⟨variable identifier⟩ ::= ⟨identifier⟩
```

5.12.2 Example

```
STREAM PROCEDURE TRANCHAR (SORC, X, DEST, Y, LGTH); VALUE X, Y, LGTH;
     BEGIN SI ← SORC; DI ← DEST;
          SI ← SI + X; DI ← DI + Y;
          DS ← LGTH CHR
     END
```

5.12.3 Semantics

A STREAM PROCEDURE declaration defines a procedure which utilizes the B 5000 character mode to be associated with a procedure identifier. Since the B 5000 character mode is designed exclusively for the manipulation of bits, characters, and computer words, the language used to describe a STREAM PROCEDURE declaration differs from that of conventional procedures.

5.12.3.1 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 address) are assigned integral values equal to the absolute addresses of the corresponding actual parameters.

5.12.3.2 Stream Declarations

All stream simple variables in a stream block must be declared by a stream variable declaration. Therefore all stream simple variables are local to the stream block.

The LABEL declaration is a list of all labels in the stream block.

5.12.3.3 Compound Stream Tail

The stream block includes, in addition to the above two declarations, a stream statement or a series of stream statements. These statements are unique to the STREAM PROCEDURE declaration and may not be used outside the declaration. Stream statements and their uses are discussed in the following sections.

For brevity and clarity, the following notation has been adopted for discussing stream statements.

SI_w -- Word address portion of source index

DI_w -- Word address portion of destination index

SI_c -- Character designator portion of source index; $SI_c = 0$ for leftmost character of word, 7 for rightmost character

DI_c -- Character designator portion of destination index; $DI_c = 0$ for leftmost character of word, 7 for rightmost character.

SI_b -- Bit designator portion of source index; $SI_b = 0$ for leftmost bit of character, 5 for rightmost bit.

DI_b -- Bit designator portion of destination index; $DI_b = 0$ for leftmost bit of character, 5 for rightmost bit.

CI_w -- Word address portion of control index.

CI_s -- Syllable designator portion of control index; $CI_s = 0$ for leftmost syllable of word, 3 for rightmost syllable.

ri -- Repetitive indicator.

5.12.3.4 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 will be referenced throughout the succeeding discussion whenever applicable.

ADJUSTMENT CATEGORY I

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.

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.

ADJUSTMENT CATEGORY II

Source Index

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

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

Destination Index

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

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

5.12.4 Stream Statements

5.12.4.1 Syntax

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

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

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⟩ |
⟨dummy statement⟩

5.12.5 Stream Address Statement

5.12.5.1 Syntax

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

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

5.12.5.2 Examples

```
SI ← LOC Q1
DI ← DC
T2 ← DI
T3 ← CI
SI ← SOURCE
DI ← T2
SI ← SI + 3
DI ← DI - T4
```

5.12.5.3 Set Address Statement

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

The set address statement using the delimiter SC or DC (see paragraph 5.12.3.4, category II) assigns the value contained in the next eighteen bits of the applicable string to the source or destination index.

5.12.5.4 Store Address Statement

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

If DI is the stream address index of a store address statement, the following condition must be met. Before the statement is executed, if the destination index (DI) is pointing to the location of the indicated stream variable, DI_c and DI_b must both equal 0.

5.12.5.5 Recall Address Statement

The recall address statement causes the value of a stream variable to be assigned to the indicated index. If DI is the stream address index of a recall address statement, the following condition must be met. Before this statement is executed, if the destination index (DI) is pointing to the location of the indicated stream variable, then DI_c and DI_b must both equal 0.

5.12.5.6 Skip Address Statement

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

5.12.5.7 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.

5.12.6 Destination String Statement

5.12.6.1 Syntax

3 ⟨destination string statement⟩ ::= DS ← ⟨transfer part⟩

3 ⟨transfer part⟩ ::= ⟨source string transfer⟩ | ⟨literal transfer⟩

3 ⟨source string transfer⟩ ::= ⟨repetitive indicator⟩ ⟨transfer type⟩

3 ⟨repetitive indicator⟩ ::= ⟨repeat part⟩ | ⟨stream simple variable⟩

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

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

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

3 ⟨visible string character⟩ ::= . | [| (| < | ← | & | $ | * |) | ; | ≤ | - | / | , | % | = |] | # | @ | : | > | ≥ | + | 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 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

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

3 ⟨string bracket character⟩ ::= "

5.12.6.2 Examples

DS ← 3 WDS

DS ← T CHR

DS ← 6 OCT

DS ← X DEC

DS ← 3 ADD

DS ← N SUB

DS ← VARY ZON

DS ← 4 NUM

DS ← 7 LIT "HEADING"

DS ← 6 RESET

DS ← X SET

5.12.6.3 Transfer Words

The transfer words option (see paragraph 5.12.3.4, category I) 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:

$$SI_w \leftarrow SI_w + ri$$
$$DI_w \leftarrow DI_w + ri$$

5.12.6.4 Transfer Characters

The transfer characters option (see paragraph 5.12.3.4, category II) causes the number of characters specified by the repetitive indicator to be trans-

ferred 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)

5.12.6.5 Input Convert

The input convert option (paragraph 5.12.3.4) 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 of the rightmost character in the source field (10 = minus, otherwise plus). The value of the repetitive indicator must not be greater than 8.

The execution of this statement affects SI and DI as follows:

$SI_c \leftarrow SI_c + ri$ (overflow into SI_w can occur)
$DI_w \leftarrow DI_w + 1$

5.12.6.6 Output Convert

The output convert option (paragraph 5.12.3.4) causes one octal word in the source string to be transferred and converted to the number of destination characters specified by the repetitive indicator. The octal word is treated as an integer. The sign is placed in the zone bits of the rightmost destination character (00 = plus, 10 = minus). All other destination zone bits are set to zero. The value of the repetitive indicator must not be greater than 8. 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)

5.12.6.7 Transfer and Add

The transfer and add option (see paragraph 5.12.3.4, category II) 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 of their respective rightmost characters (10 = minus, otherwise plus). All other source zone bits are ignored. All other destination zone bits are set to zero. The sign of the result is placed in the zone bits of the rightmost destination character. If overflow occurs in the destination field, TOGGLE is set to TRUE, otherwise 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)

5.12.6.8 Transfer Character Portions
The transfer character portions option (see paragraph 5.12.3.4, category II) causes either the zone bits or 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 transferring zone bits the numeric portions of the destination characters are not affected. When transferring numeric bits the zone portions of the destination characters are set to zero. TOGGLE is set only when transferring numeric bits, as follows: If the zone bits of the rightmost source character are 10 (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)

5.12.6.9 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 rightmost string characters are ignored. The ex-

ecution of this statement affects DI only, as follows:

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

5.12.6.10 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:

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

5.12.6.11 Repetitive Indicator

The value of the repetitive indicator must never exceed 63.

5.12.7 Stream GO TO Statement

5.12.7.1 Syntax

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

5.12.7.2 Example

GO TO START

5.12.7.3 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.

5.12.7.4 Restriction

A stream GO TO statement must not cause transfer into nor out of a stream nest statement.

5.12.8 SKIP Bit Statement

5.12.8.1 Syntax

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

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

5.12.8.2 Examples

SKIP N SB

SKIP 12 DB

5.12.8.3 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)

5.12.9 Stream TALLY Statement

5.12.9.1 Syntax

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

5.12.9.2 Examples

TALLY ← ABLE

TALLY ← TALLY + 1

TALLY ← TALLY + BETA

GAMMA ← TALLY

5.12.9.3 Semantics

The stream TALLY statement provides a counting mechanism for stream procedures. TALLY has a value which is modulo 64; all overflows are lost.

5.12.10 Stream Nest Statements

5.12.10.1 Syntax

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⟩

5.12.10.2 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 L3; SI ← SI + 1);
TALLY ← TALLY + 1)

5.12.10.3 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 the compound nest. An additional statement, the JUMP OUT statement, is allowed in a compound nest. It may not be used elsewhere. 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 labelled statement. The JUMP OUT statement may be labelled.

5.12.10.4 Restrictions

A stream nest statement may be entered only at its beginning.

5.12.11 Stream RELEASE Statement

5.12.11.1 Syntax

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

5.12.11.2 Examples

    RELEASE (FILENAME1)

5.12.11.3 Semantics

The actual parameter corresponding to the formal parameter of a stream RELEASE statement must be a file identifier. This formal parameter must not be called by value. If the file 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 file 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 RELEASE statement.

5.12.12 Compound Stream Statement

5.12.12.1 Syntax

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

5.12.12.2 Example

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

5.12.12.3 Semantics

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

5.12.13 Conditional Stream Statement

5.12.13.1 Syntax

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

5.12.13.2 Examples

IF SC = "E" THEN GO TO BLAZES

IF 8 SC = DC THEN GO TO TUCSON

IF SB THEN SI ← SI + 1

IF TOGGLE THEN DS ← X ZON

5.12.13.3 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. Several kinds of tests are possible, as follows:

5.12.13.4 Source with Literal

The source with literal option (see paragraph 5.12.3.4, category II, for SI only) causes one source character to be compared with the character indicated in the test. If the relation is satisfied, TOGGLE is set to TRUE, otherwise it is set to FALSE.

5.12.13.5 Source with Destination

The source with destination option (see paragraph 5.12.3.4 category II) compares a specified number of source characters with the same number of destination characters. If the relation is satisfied, 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)

5.12.13.6 Source Bit

This test causes one source bit to be tested for 1. TOGGLE is set to TRUE if the result of the test is true, otherwise it is set to FALSE.

5.12.13.7 TOGGLE

This test is merely one for the value of TOGGLE. It does not change the TOGGLE value.

5.12.13.8 Source for Alpha

The source for alpha option (see paragraph 5.12.3.4, category II, for SI only) tests one source character. If it is a letter or a digit, TOGGLE is set to TRUE, otherwise it is set to FALSE.

5.12.14 Dummy Statement

5.12.14.1 Syntax

1 ⟨dummy statement⟩ ::= ⟨empty⟩

5.12.14.2 Examples

BOTTOM:

FINI:

5.12.14.3 Semantics

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

APPENDIX A

B 5000 INTERNAL CHARACTER CODES

In Order of Collating Sequence

(with Text References)

Character	6-Bit Code	Text References
blank	11 0000	(1.2, 2.4, 2.7, 4.7, 5.12.6)
.	01 1010	(1.2, 2.4, 2.6, 3.2, 5.7, 5.12.6)
[	01 1011	(1.2, 2.4, 3.1, 3.2, 3.6, 4.8.2, 4.8.3, 4.9, 5.2, 5.6, 5.7, 5.8, 5.11, 5.12.6)
(	01 1101	(1.2, 2.4, 3.2, 3.3, 3.4, 3.5, 3.6, 4.7, 4.8.2, 4.8.3, 4.8.4, 5.6, 5.7, 5.8, 5.10, 5.11, 5.12, 5.12.6, 5.12.10, 5.12.11)
<	01 1110	(1.2, 2.4, 3.5, 5.12.6)
←	01 1111	(1.2, 2.4, 4.2, 4.6, 5.3, 5.12.5, 5.12.6, 5.12.9)
&	01 1100	(1.2, 2.4, 5.12.6)
$	10 1010	(1.2, 5.12.6)
*	10 1011	(1.2, 2.4, 3.4, 4.9, 5.11, 5.12.6)
)	10 1101	(1.2, 2.4, 3.2, 3.3, 3.4, 3.5, 3.6, 4.7, 4.8.2, 4.8.3, 4.8.4, 5.6, 5.7, 5.8, 5.10, 5.11, 5.12, 5.12.6, 5.12.10, 5.12.11)
;	10 1110	(1.2, 2.4, 4.1, 5.11, 5.12, 5.12.6, 5.12.10)
≤	10 1111	(1.2, 2.4, 3.5, 5.12.6)
-	10 1100	(1.2, 2.4, 2.6, 3.4, 5.12.5, 5.12.6)
/	11 0001	(1.2, 2.4, 3.4, 5.7, 5.12.6)
,	11 1010	(1.2, 2.4, 3.1, 3.3, 4.6, 4.7, 4.8.2, 4.8.3, 4.8.4, 4.9, 5.1, 5.2, 5.3, 5.4, 5.5, 5.6, 5.7, 5.8, 5.10, 5.11, 5.12, 5.12.6)
%	11 1011	(1.2, 5.12.6)
=	11 1101	(1.2, 2.4, 3.5, 5.4, 5.12.6, 5.12.13)
]	11 1110	(1.2, 2.4, 3.1, 3.2, 3.6, 4.8.2, 4.8.3, 4.9, 5.2, 5.6, 5.7, 5.8, 5.11, 5.12.6)
"	11 1111	(1.2, 2.4, 2.7, 3.3, 4.7, 5.12.6, 5.12.13)
#	00 1010	(1.2, 2.4, 5.4, 5.12.6)
@	00 1011	(1.2, 2.4, 2.6, 5.12.6)
:	00 1101	(1.2, 2.4, 3.2, 4.0, 4.1, 4.5, 4.6, 5.2, 5.10, 5.12.4, 5.12.6, 5.12.10)
>	00 1110	(1.2, 2.4, 3.5, 5.12.6)
≥	00 1111	(1.2, 2.4, 3.5, 5.12.6)
+	01 0000	(1.2, 2.4, 2.6, 3.4, 5.12.5, 5.12.6, 5.12.9)

Character	6-Bit Code	Text References
A	01 0001	(1.2, 2.1, 5.7, 5.12.6)
B	01 0010	(1.2, 2.1, 5.12.6)
C	01 0011	(1.2, 2.1, 5.12.6)
D	01 0100	(1.2, 2.1, 5.7, 5.12.6)
E	01 0101	(1.2, 2.1, 5.7, 5.12.6)
F	01 0110	(1.2, 2.1, 5.7, 5.12.6)
G	01 0111	(1.2, 2.1, 5.12.6)
H	01 1000	(1.2, 2.1, 5.12.6)
I	01 1001	(1.2, 2.1, 5.7, 5.12.6)
×	10 0000	(1.2, 2.4, 3.4, 5.12.6)
J	10 0001	(1.2, 2.1, 5.12.6)
K	10 0010	(1.2, 2.1, 5.12.6)
L	10 0011	(1.2, 2.1, 5.7, 5.12.6)
M	10 0100	(1.2, 2.1, 5.12.6)
N	10 0101	(1.2, 2.1, 5.12.6)
O	10 0110	(1.2, 2.1, 5.7, 5.12.6)
P	10 0111	(1.2, 2.1, 5.12.6)
Q	10 1000	(1.2, 2.1, 5.12.6)
R	10 1001	(1.2, 2.1, 5.12.6)
≠	11 1100	(1.2, 2.4, 3.5, 5.12.6)
S	11 0010	(1.2, 2.1, 5.12.6)
T	11 0011	(1.2, 2.1, 5.12.6)
U	11 0100	(1.2, 2.1, 5.12.6)
V	11 0101	(1.2, 2.1, 5.12.6)
W	11 0110	(1.2, 2.1, 5.12.6)
X	11 0111	(1.2, 2.1, 5.7, 5.12.6)
Y	11 1000	(1.2, 2.1, 5.12.6)
Z	11 1001	(1.2, 2.1, 5.12.6)

Character	6-Bit Code	Text References
0	00 0000	(1.2, 2.2, 4.9, 5.6, 5.12.6, 5.12.13)
1	00 0001	(1.2, 2.2, 4.9, 5.6, 5.12.6, 5.12.13)
2	00 0010	(1.2, 2.2, 4.9, 5.6, 5.12.6)
3	00 0011	(1.2, 2.2, 4.9, 5.6, 5.12.6)
4	00 0100	(1.2, 2.2, 4.9, 5.12.6)
5	00 0101	(1.2, 2.2, 4.9, 5.12.6)
6	00 0110	(1.2, 2.2, 4.9, 5.12.6)
7	00 0111	(1.2, 2.2, 4.9, 5.12.6)
8	00 1000	(1.2, 2.2, 5.12.6)
9	00 1001	(1.2, 2.2, 5.12.6)
?	00 1100	(1.2)

INDEX

Metalinguistic Variables

The syntactical definition of each Extended ALGOL metalinguistic variable will be found in the section or subsection shown below. Where more than one reference is given, the succeeding references represent the same definition in a different context or a discussion of the semantics of the metalinguistic variable. References in parentheses indicate the use of the metalinguistic variable in the right-hand part of a syntactical definition.

⟨actual parameter⟩ 3.3, 4.7

⟨actual parameter list⟩ 3.3, 4.7

⟨actual parameter part⟩ 3.3, 4.7

⟨adding operator⟩ 3.4, 5.12.5

⟨arithmetic expression⟩ 3.4
(3.0, 3.1, 3.2, 4.2, 4.6, 4.8 2, 4.8.4, 4.9, 5.2, 5.8)

⟨arithmetic operator⟩ 2.4

⟨array declaration⟩ 5.2
(5.0)

⟨array identifier⟩ 3.1, 4.9
(3.3, 4.7, 5.2, 5.10)

⟨array identifier list⟩ 5.11

⟨array kind⟩ 5.2

⟨array list⟩ 5.2

⟨array segment⟩ 5.2

⟨array specification⟩ 5.11

⟨array specifier⟩ 5.11

⟨array specifier list⟩ 5.11

⟨assignment statement⟩ 4.2
(4.0)

⟨basic statement⟩ 4.0
(4.5)

⟨basic symbol⟩ 2.0

⟨bits in field⟩ 3.2

⟨block⟩ 4.1
(4.0, 4.5)

⟨block head⟩ 4.1

⟨blocking⟩ 5.6

⟨blocking option⟩ 5.6

⟨Boolean expression⟩ 3.5
(3.0, 3.4, 4.2, 4.5, 4.6, 4.8.2, 5.8)

⟨Boolean factor⟩ 3.5

⟨Boolean primary⟩ 3.5

⟨Boolean secondary⟩ 3.5

⟨Boolean term⟩ 3.5

⟨bound pair⟩ 5.2

⟨bound pair list⟩ 5.2

⟨bracket⟩ 2.4

⟨buffer part⟩ 5.6

⟨buffer size⟩ 5.6

⟨carriage control⟩ 4.8.3

⟨character⟩ 1.2

⟨compound nest⟩ 5.12.10

⟨compound statement⟩ 4.1
(4.0, 4.5)

⟨compound stream statement⟩ 5.12.12
(5.12.4)

⟨compound stream tail⟩ 5.12
(5.12.12)

⟨compound tail⟩ 4.1

⟨conditional statement⟩ 4.5
(4.0)

⟨conditional stream statement⟩ 5.12.13
(5.12.4)

⟨decimal fraction⟩ 2.6

⟨decimal number⟩ 2.6

⟨decimal places⟩ 5.7

⟨declaration⟩ 5.0
(4.1)

⟨declarator⟩ 2.4

⟨define declaration⟩ 5.4
(5.0)

⟨defined identifier⟩ 5.4
(4.7)

⟨definition⟩ 5.4

⟨definition list⟩ 5.4

⟨delimiter⟩ 2.4
(2.0)

⟨designational expression⟩ 3.6
(3.0, 4.3, 5.3)

⟨destination address part⟩ 5.12.5

⟨destination string statement⟩ 5.12.6
(5.12.4)

⟨diagnostic declaration⟩ 5.10
(5.0)

⟨digit⟩ 2.2
(2.0, 2.5, 2.6)

⟨disposition⟩ 5.6

⟨double space⟩ 4.8.3

⟨dummy statement⟩ 4.4
(4.0)

⟨dump indicator⟩ 5.10

⟨dump list⟩ 5.10

⟨dump list element⟩ 5.10

⟨dump part⟩ 5.10

⟨editing phrase⟩ 5.7

⟨editing phrase type⟩ 5.7

⟨editing segment⟩ 5.7

⟨editing specifications⟩ 5.7

⟨empty⟩ 1.2
(3.3, 4.4, 4.7, 4.8.3, 4.8.4, 5.2, 5.6, 5.7, 5.9, 5.11, 5.12, 5.12.6)

⟨end of file⟩ 5.6

⟨exponent part⟩ 2.6

⟨expression⟩ 3.0
(3.3, 4.7)

⟨expression list⟩ 4.8.2, 5.8

⟨expression part⟩ 4.8.2, 5.8

⟨factor⟩ 3.4

⟨field description⟩ 3.2

⟨field part⟩ 5.7

⟨field width⟩ 5.7

⟨file control part⟩ 5.6

⟨file declaration⟩ 5.6
(5.0)

⟨file identifier⟩ 5.6
(3.3, 4.7, 4.8.2, 4.8.3, 4.8.4, 5.10)

⟨file part⟩ 5.6

⟨fill statement⟩ 4.9
(4.0)

⟨for clause⟩ 4.6
(4.8.2, 5.8)

⟨for list⟩ 4.6

⟨for list element⟩ 4.6

⟨formal parameter⟩ 5.11
(5.12, 5.12.11)

⟨formal parameter list⟩ 5.11, 5.12

⟨formal parameter part⟩ 5.11

⟨format and list parameters⟩ 4.8.3

⟨format and list part⟩ 4.8.3

⟨format declaration⟩ 5.7
(5.0)

⟨format identifier⟩ 5.7
(3.3, 4.7, 4.8.2, 4.8.3)

⟨format part⟩ 5.7

⟨for statement⟩ 4.6
(4.0, 4.5)

⟨forward procedure declaration⟩ 5.9
(5.0)

⟨forward reference declaration⟩ 5.9

⟨forward switch declaration⟩ 5.9

⟨function designator⟩ 3.3
(3.2, 3.4, 3.5)

⟨go to statement⟩ 4.3
(4.0)

⟨identifier⟩ 2.5
(3.1, 3.3, 3.6, 4.9, 5.4, 5.5,
5.6, 5.7, 5.8, 5.11, 5.12)

⟨identifier list⟩ 5.11, 5.12

⟨if clause⟩ 3.4, 4.5
(3.5, 3.6)

⟨if statement⟩ 4.5

⟨illegitimate character⟩ 1.2

⟨implication⟩ 3.5

⟨initial value⟩ 4.9

⟨input convert⟩ 5.12.6

⟨input or output⟩ 5.7

⟨input parameters⟩ 4.8.2

⟨integer⟩ 2.6
(5.11)

⟨I-O part⟩ 5.6

⟨I-O statement⟩ 4.8
(4.0)

⟨I-O unit control⟩ 5.6

⟨jump out statement⟩ 5.12.10

⟨label⟩ 3.6, 5.5, 5.12
(4.0, 4.1, 4.5, 4.6, 5.6, 5.10,
5.12.4, 5.12.7, 5.12.10)

⟨label declaration⟩ 5.5, 5.12
(5.0)

⟨label list⟩ 5.5, 5.12

⟨left bit of field⟩ 3.2

⟨left part⟩ 4.2

⟨left part list⟩ 4.2

⟨letter⟩ 2.1
(2.0, 2.5, 2.7, 4.7)

⟨letter string⟩ 2.7, 4.7

⟨list⟩ 4.8.2, 5.8
(4.8.3)

⟨list declaration⟩ 5.8
(5.0)

⟨list identifier⟩ 5.8
(3.3, 4.7, 4.8.2)

⟨list part⟩ 5.8

⟨list segment⟩ 4.8.2, 5.8

⟨literal bits⟩ 5.12.6

⟨literal characters⟩ 5.12.6

⟨literal transfer⟩ 5.12.6

⟨local or own type⟩ 5.1, 5.2

⟨logical operator⟩ 2.4

⟨logical value⟩ 2.3
(2.0, 3.5)

⟨lower bound⟩ 5.2

⟨lower bound list⟩ 5.11

⟨monitor list⟩ 5.10

⟨monitor list element⟩ 5.10

⟨monitor part⟩ 5.10

⟨multiplying operator⟩ 3.4

⟨nest⟩ 5.12.10

⟨no space⟩ 4.8.3

⟨number⟩ 2.6
(4.9)

⟨number of buffers⟩ 5.6

⟨octal digit⟩ 4.9

⟨octal number⟩ 4.9

⟨operator⟩ 2.4

⟨output convert⟩ 5.12.6

⟨output parameters⟩ 4.8.3

⟨parameter delimiter⟩ 3.3, 4.7
(5.11, 5.12)

⟨partial word designator⟩ 3.2
(3.4, 3.5, 4.2)

⟨partial word operand⟩ 3.2

⟨primary⟩ 3.4

⟨procedure body⟩ 5.11

⟨procedure declaration⟩ 5.11
(5.0)

⟨procedure heading⟩ 5.11
(5.9)

⟨procedure identifier⟩ 3.3, 5.11
(4.2, 4.7, 5.10, 5.12)

⟨procedure statement⟩ 4.7
(4.0)

⟨procedure type⟩ 5.9

⟨program⟩ 4.1

⟨proper string⟩ 2.7, 5.12.6

⟨read statement⟩ 4.8.2
(4.8)

⟨recall address statement⟩ 5.12.5

⟨records per block⟩ 5.6

⟨relation⟩ 3.5

⟨relational operator⟩ 2.4, 3.5
(5.12.13)

⟨release statement⟩ 4.8.4
(4.8)

⟨repeat part⟩ 5.7, 5.12.6

⟨repetitive indicator⟩ 5.12.6
(5.12.8, 5.12.10, 5.12.13)

⟨row⟩ 4.9

⟨row designator⟩ 4.9

⟨save factor⟩ 5.6

⟨separator⟩ 2.4

⟨sequential operator⟩ 2.4

⟨set address statement⟩ 5.12.5

⟨simple arithmetic expression⟩ 3.4
(3.5)

⟨simple Boolean⟩ 3.5

⟨simple designational expression⟩ 3.6

⟨simple variable⟩ 3.1
(5.1, 5.6, 5.10)

⟨single space⟩ 1.2, 5.12.6
(2.4)

⟨skip address statement⟩ 5.12.5

⟨skip bit statement⟩ 5.12.8
(5.12.4)

⟨skip to channel⟩ 4.8.3

⟨skip to next page⟩ 4.8.3

⟨source address part⟩ 5.12.5

⟨source bit⟩ 5.12.13

⟨source for alpha⟩ 5.12.13

⟨source or destination bit⟩ 5.12.8

⟨source string transfer⟩ 5.12.6

⟨source with destination⟩ 5.12.13

⟨source with literal⟩ 5.12.13

⟨space⟩ 1.2
(2.7, 4.7)

⟨specification⟩ 5.11

⟨specification list⟩ 5.11

⟨specification part⟩ 5.11

⟨specificator⟩ 2.4

⟨specified lower bound⟩ 5.11

⟨specifier⟩ 5.11

⟨statement⟩ 4.0
(4.1, 4.5, 4.6, 5.11)

⟨store address statement⟩ 5.12.5

⟨stream address index⟩ 5.12.5

⟨stream address statement⟩ 5.12.5
(5.12.4)

⟨stream arithmetic expression⟩ 5.12.5

⟨stream block⟩ 5.12

⟨stream block head⟩ 5.12

⟨stream declaration⟩ 5.12

⟨stream formal parameter part⟩ 5.12

⟨stream go to statement⟩ 5.12.7
(5.12.4)

⟨stream if clause⟩ 5.12.13

⟨stream nest statement⟩ 5.12.10
(5.12.4)

⟨stream primary⟩ 5.12.5
(5.12.9)

⟨stream procedure declaration⟩ 5.12
(5.0)

⟨stream procedure heading⟩ 5.12

⟨stream release statement⟩ 5.12.11
(5.12.4)

⟨stream simple variable⟩ 5.12
(5.12.5, 5.12.6, 5.12.9)

⟨stream statement⟩ 5.12.4
(5.12, 5.12.10)

⟨stream tally statement⟩ 5.12.9
(5.12.4)

⟨stream variable declaration⟩ 5.12

⟨stream variable list⟩ 5.12

⟨string⟩ 2.7, 5.12.6
(3.4, 4.9, 5.7)

⟨string bracket character⟩ 1.2, 5.12.6
(2.7, 5.12.13)

⟨string character⟩ 1.2, 5.12.6
(2.7, 5.12.13)

⟨subscript expression⟩ 3.1
(3.6)

⟨subscript list⟩ 3.1

⟨subscripted variable⟩ 3.1
(5.10)

⟨switch declaration⟩ 5.3
(5.0)

⟨switch designator⟩ 3.6

⟨switch identifier⟩ 3.6
(3.3, 4.7, 5.3, 5.9, 5.10)

⟨switch list⟩ 5.3

⟨term⟩ 3.4

⟨test⟩ 5.12.13

⟨transfer and add⟩ 5.12.6

⟨transfer and convert⟩ 5.12.6

⟨transfer character portions⟩ 5.12.6

⟨transfer characters⟩ 5.12.6

⟨transfer part⟩ 5.12.6

⟨transfer type⟩ 5.12.6

⟨transfer words⟩ 5.12.6

⟨type⟩ 5.1
(5.2, 5.9, 5.11, 5.12)

⟨type declaration⟩ 5.1
(5.0)

⟨type list⟩ 5.1

⟨unconditional statement⟩ 4.0, 4.5

⟨unconditional stream statement⟩ 5.12.4
(5.12.13)

⟨unlabelled basic statement⟩ 4.0

⟨unlabelled block⟩ 4.1

⟨unlabelled compound statement⟩ 4.1

⟨unlabelled stream statement⟩ 5.12.4

⟨unsigned integer⟩ 2.6
(3.2, 4.8.3, 5.6, 5.7, 5.10, 5.12.5, 5.12.6)

⟨unsigned number⟩ 2.6
(3.4)

⟨upper bound⟩ 5.2

⟨value list⟩ 4.9

⟨value part⟩ 5.11, 5.12

⟨variable⟩ 3.1
(3.2, 3.4, 3.5, 4.2, 4.6)

⟨variable identifier⟩ 3.1, 5.12

⟨visible string character⟩ 1.2, 5.12.6

⟨word count⟩ 4.8.4

⟨write statement⟩ 4.8.3

Reserved Words

Reserved words in Extended ALGOL are discussed and used in the sections and subsections listed below. The use of parentheses indicates that these words do not appear in the left-hand part of a syntactical definition.

ABS (3.3.4)

ADD (2.4, 5.12.6)

ALPHA (2.4, 5.1, 5.12.13)

AND (2.4, 3.5)

ARCTAN (3.3.4)

ARRAY (2.4, 5.2, 5.11)

BEGIN (2.4, 4.1, 5.12, 5.12.12)

BOOLEAN (2.4, 5.1)

CHR (2.4, 5.12.6)

CI (2.4, 5.12.5)

COMMENT (2.4)

COS (3.3.4)

DB (2.4, 5.12.8)

DBL (2.4, 4.8.3)

DC (2.4, 5.12.5, 5.12.13)

DEC (2.4, 5.12.6)

DEFINE (2.4, 5.4)

DI (2.4, 5.12.5)

DIV (2.4, 3.4)

DO (2.4, 4.6)

DOUBLE (2.4, 5.1)

DS (2.4, 5.12.6)

DUMP (2.4, 5.10)

ELSE (2.4, 3.4, 3.6, 4.5)

END (2.4, 4.1, 5.12)

ENTIER (3.3.5)

EQV (2.4, 3.5)

EXP (3.3.4)

FALSE (2.3)

FILE (2.4, 5.6, 5.11)

FILL (2.4, 4.9)

FOR (2.4, 4.6)

FORMAT (2.4, 5.7, 5.11)

FORWARD (2.4, 5.9)

GO (2.4, 4.3, 5.12.7)

IF (2.4, 3.4, 4.5, 5.12.13)

IMP (2.4, 3.5)

IN (2.4, 5.6, 5.7)

INTEGER (2.4, 5.1)

JUMP (2.4, 5.12.10)

LABEL (2.4, 5.5, 5.11, 5.12)

LIST (2.4, 5.8, 5.11)

LIT (2.4, 5.12.6)

LN (3.3.4)

LOC (2.4, 5.12.5)

LOCAL (2.4, 5.12)

MOD (2.4, 3.4)

MONITOR (2.4, 5.10)

NO (2.4, 4.8.3)

NOT (2.4, 3.5)

NUM (2.4, 5.12.6)

OCT (2.4, 4.9, 5.12.6)

OR (2.4, 3.5)

OUT (2.4, 5.6, 5.7, 5.12.10)

OWN (2.4, 5.1, 5.2)

PAGE (2.4, 4.8.3)

PROCEDURE (2.4, 5.9, 5.11, 5.12)

READ (2.4, 4.8.2)

REAL (2.4, 5.1)

RELEASE (2.4, 4.8.4, 5.12.11)

RESET (2.4, 5.12.6)

REVERSE (2.4, 5.6)

SAVE (2.4, 5.2)

SB (2.4, 5.12.8, 5.12.13)

SC (2.4, 5.12.5, 5.12.13)

SET (2.4, 5.12.6)

SI (2.4, 5.12.5)

SIGN (3.3.4)

SIN (3.3.4)

SKIP (2.4, 5.12.8)

SQRT (3.3.4)

STEP (2.4, 4.6)

STREAM (2.4, 5.11, 5.12)

SUB (2.4, 5.12.6)

SWITCH (2.4, 5.3, 5.9, 5.11)

TALLY (2.4, 5.12.9)

THEN (2.4, 3.4, 4.5, 5.12.13)

TO (2.4, 4.3, 5.12.7)

TOGGLE (2.4, 5.12.13)

TRUE (2.3)

UNTIL (2.4, 4.6)

VALUE (2.4, 5.11, 5.12)

WDS (2.4, 5.12.6)

WHILE (2.4, 4.6)

WITH (2.4, 4.9)

WRITE (2.4, 4.8.3)

ZON (2.4, 5.12.6)

Burroughs Corporation
DETROIT, MICHIGAN 48232
Offices in Principal Cities
In Canada: Burroughs Business Machines Ltd., Toronto, Ontario

REV. A

LITHO IN U.S.A. 3-8-6 5000–21012