9

ECMA

EUROPEAN COMPUTER MANUFACTURERS ASSOCIATION

ECMA STANDARD

on

FORTRAN

April 1965

Free copies of this standard **ECMA-9** are available from

ECMA European Computer Manufacturers Association

114, rue du Rhône — 1204 Geneva (Switzerland)

ECMA

EUROPEAN COMPUTER MANUFACTURERS ASSOCIATION

ECMA STANDARD

on

FORTRAN

April 1965

BRIEF HISTORY

A Technical Committee of ECMA met for the first time in January 1964 to prepare a standard on FORTRAN.

Representatives of the following companies participated in the work of the committee:

A.E.I. Ltd

Compagnie des Machines Bull

N.V. Electrologica

Elliott Bros. (London) Ltd

IBM-WTEC

I.C.T., International Computers & Tabulators Ltd

I.T.T. Europe Inc.

Ing. C. Olivetti & Co. S.p.A.

SETI, Société Européenne pour le Traitement de l'Information

Sperry Rand International Corp.

The work has led to this Standard ECMA-8 adopted by the General Assembly on April 30, 1965. In addition, the Committee has collaborated with the following organizations:

American Standards Association (ASA)

International Organization for Standardization (ISO)

CONTENTS

* Blank in ECMA FORTRAN

1. INTRODUCTION

1.1. Purpose. This standard establishes the form for and the interpretation of programs expressed in the FORTRAN language for the purpose of promoting a high degree of interchangeability of such programs for use on a variety of automatic data processing systems. A processor shall conform to this standard provided it accepts and interprets as specified, at least those forms and relationships described herein.

Insofar as the interpretation of the form and relationships described are not affected, any statement of requirement could be replaced by a statement expressing that the standard does not provide an interpretation unless the requirement is met. Further, any statement of prohibition could be replaced by a statement expressing that the standard does not provide an interpretation when the prohibition is violated.

The ECMA FORTRAN has been adopted by the International Organization for Standardization (ISO) as intermediate level (referred to as Intermediate FORTRAN) between "FORTRAN" and "Basic FORTRAN" as will be described in the ISO Recommendation on FORTRAN. The ECMA FORTRAN is a subset of the full ISO FORTRAN. The FORTRANs are upwardly compatible from Basic to ECMA to ISO FORTRAN. The wording of ECMA FORTRAN has been rephrased so as to match the text of the ISO Recommendation.

Attached in Appendix 1 are those items which are available in ECMA FORTRAN that are not available in Basic. Appendix 2 is a list of those items available in ISO FORTRAN that are not available in ECMA FORTRAN.

The ISO FORTRAN and the ECMA FORTRAN use identical section numbering. Where ECMA FORTRAN has no language content counterpart to that defined in a particular section of ISO FORTRAN, only the section number is retained.

1.2. Scope. This standard establishes:

(1) The form of a program written in the FORTRAN language.
(2) The form of writing input data to be processed by such a program operating on automatic data processing systems.
(3) Rules for interpreting the meaning of such a program.
(4) The form of the output data resulting from the use of such a program on automatic data processing systems, provided that the rules of interpretation establish an interpretation.

This standard does not prescribe:

(1) The mechanism by which programs are transformed for use on a data processing system (the combination of this mechanism and data processing system is called a processor).
(2) The method of transcription of such programs or their input or output data to or from a data processing medium.
(3) The manual operations required for set-up and control

of the use of such programs on data processing equiment.

(4) The results when the rules for interpretation fail to establish an interpretation of such a program.

(5) The size or complexity of a program that will exceed the capacity of any specific data processing system or the capability of a particular processor.

(6) The range or precision of numerical quantities.

2. BASIC TERMINOLOGY

This section introduces some basic terminology and some concepts. A rigorous treatment of these is given in later sections. Certain assumptions concerning the meaning of grammatical forms and particular words are presented.

A program that can be used as a self-contained computing procedure is called an executable program (9.1.6).

An executable program consists of precisely one main program and possibly one or more subprograms (9.1.6).

A main program is a set of statements and comments not containing a FUNCTION or SUBROUTINE statement (9.1.5).

A procedure subprogram is similar to a main program but is headed by a FUNCTION or SUBROUTINE statement. A procedure subprogram is sometimes referred to as a subprogram (9.1.3).

The term "program unit" will refer to either a main program or subprogram (9.1.7).

Any program unit may reference an "external procedure" (Section 8).

An external procedure that is defined by FORTRAN statements is called a "procedure subprogram". External procedures also may be defined by other means. An external procedure may be an external function or an external subroutine. An external function defined by FORTRAN statements headed by a FUNCTION statement is called a "function subprogram". An external subroutine defined by FORTRAN statements headed by a SUBROUTINE statement is called a "subroutine subprogram".
(Sections 8 and 9).

Any program unit consists of statements and comments. A statement is divided into physical sections called lines, the first of which is called an initial line and the rest of which are called continuation lines (3.2).

There is a type of line called a comment that is not a statement and merely provides information for documentary purposes (3.2).

The statements in FORTRAN fall into two broad classes executable and nonexecutable. The executable statements specify the action of the program while the nonexecutable statements de-

scribe the use of the program, the characteristics of the operands, editing information, statement functions, or data arrangement (7.1, 7.2).

The syntactic elements of a statement are names and operators. Names are used to reference objects such as data or procedures. Operators including the imperative verbs, specify action upon named objects.

One class of name, the array name, deserves special mention. An array name and the size of the identified array must be defined in an array declarator (7.2.1.1). An array name qualified only by a subscript is used to identify a particular element of the array (5.1.3).

Data names and the arithmetic (or logical) operations may be connected into expressions. Evaluation of such an expression develops a value. This value is derived by performing the specified operations on the named data.

The identifiers used in FORTRAN are names and numbers. Data are named. Procedures are named. Statements are labelled with numbers. Input/output units are numbered (Sections 3, 6, 7).

At various places in this document there are statements with associated lists of entries. In all such cases the list is assumed to contain at least one entry unless an explicit exception is stated.
As an example, in the statement

$$\text{SUBROUTINE } s(a_1, a_2, \ldots, a_n)$$

it is assumed that at least one symbolic name is included in the list within parentheses. A list is a set of identifiable elements, each of which is separated from its successor by a comma. Further, in a sentence a plural form of a noun will be assumed to also specify the singular form of that noun as a special case when the context of the sentence does not prohibit this interpretation.

The term reference is used as a verb with special meaning as defined in Section 5.

3. PROGRAM FORM

Every program unit is constructed of characters grouped into lines and statements.

3.1. The Fortran Character Set. A program unit is written using the following characters: 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, O, 1, 2, 3, 4, 5, 6, 7, 8, 9, and:

Character	Name of Character
	Blank
=	Equals
+	Plus
-	Minus
*	Asterisk
/	Slash
(	Left Parenthesis
)	Right Parenthesis
,	Comma
.	Decimal Point

The order in which the characters are listed does not imply a collating sequence.

3.1.1. Digits. A digit is one of the ten characters: O, 1, 2, 3, 4, 5, 6, 7, 8, 9. Unless specified otherwise, a string of digits will be interpreted in the decimal base number system when a number system base interpretation is appropriate.

An octal digit is one of the eight characters: O, 1, 2, 3, 4, 5, 6, 7. These are only used in the STOP (7.1.2.7.1) and PAUSE (7.1.2.7.2) statements.

3.1.2 Letters. A letter is one of the twenty-six characters: 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.

3.1.3. Alphanumeric Characters. An alphanumeric character is a letter or a digit.

3.1.4 Special Characters. A special character is one of the ten characters: blank, equals, plus, minus, asterisk, slash, left, parenthesis, right parenthesis, comma and decimal point.

3.1.4.1. Blank Character. With the exception of the uses specified (3.2.2, 3.2.3, 3.2.4, 7.2.3.6, and 7.2.3.8) a blank character has no meaning and may be used freely to improve the appearance of the program subject to the restriction on continuation lines in 3.3.

3.2 Lines. A line is a string of 72 characters. All characters must be from the FORTRAN character set except as described in 7.2.3.8.

The character positions in a line are called columns and are consecutively numbered 1, 2, 3,...,72. The number indicates the

sequential position of a character in the line starting at the left and proceeding to the right.

3.2.1 Comment Line. The letter C in column 1 of a line designates that line as a comment line. A comment line must be immediately followed by an initial line, another comment line, or an end line.

A comment line does not affect the program in any way and is available as a convenience for the programmer.

3.2.2 End Line. An end line is a line with the character blank in columns 1 through 6, the characters E, N, and D, once each in that order, in columns 7 through 72, preceded by, interspersed with, or followed by the character blank. The end line indicates, to the processor, the end of the written description of a program unit (9.1.7). Every program unit must physically terminate with an end line.

3.2.3 Initial Line. An initial line is a line that is neither a comment line nor an end line and that contains the digit 0 or the character blank in column 6. Columns 1 through 5 contain the statement label or each contains the character blank.

3.2.4 Continuation Line. A continuation line is a line that contains any character other than the digit 0 or the character blank in column 6, and does not contain the character C in column 1.

A continuation line may only follow an initial line or another continuation line.

3.3 Statements. A statement consists of an initial line optionally followed by up to nine ordered continuation lines. The statement is written in columns 7 through 72 of the lines.* The order of the characters in the statement is columns 7 through 72 of the initial line followed, as applicable, by columns 7 through 72 of the first continuation line, columns 7 through 72 of the next continuation line, etc.

3.4 Statement Label. Optionally, a statement may be labelled so that it may be referred to in other statements. A statement label consists of from one to five digits. The value of the integer represented is not significant but must be greater than zero. The statement label may be placed anywhere in columns 1 through 5 of the initial line of the statement. The same statement label may not be given to more than one statement in a program unit. Leading zeros are not significant in differentiating statement labels.

3.5 Symbolic Names. A symbolic name consists of from one to six alphanumeric characters, the first of which must be alphabetic. See 10.1 through 10.1.10 for a discussion of classification

* Note case of statement embedded in logical IF statement See section 7.1.2.3.

of symbolic names and restrictions on their use.

3.6 Ordering of Characters. An ordering of characters is assumed within a program unit. Thus any meaningful collection of characters that constitutes names, lines, and statements exists as a totally ordered set. This ordering is imposed by the character position rule of 3.2 (which orders characters within lines) and the order in which lines are presented for processing.

4. DATA TYPES

Three different types of data are defined. These are integer, real, and logical. Each type has a different mathematical significance and many have different internal representation. Thus the data type has a significance in the interpretation of the associated operations with which a datum is involved. The data type of a function defines the type of the datum it supplies to the expression in which it appears.

4.1 Data Type Association. The name employed to identify a datum or function carries the data type association. The form of the string representing a constant defines both the value and the data type.

A symbolic name representing a function, variable, or array must have only a single data type association for each program unit. Once associated with a particular data type, a specific name implies that type for any differing usage of that symbolic name that requires a data type association throughout the program unit in which it is defined.

Data type may be established for a symbolic name by declaration in a type-statement (7.2.1.6) for the integer, real, and logical types. This specific declaration overrides the implied association available for integer and real (5.3).

4.2 Data Type Properties. The mathematical and the representation properties for each of the data types are defined in the following sections. For real and integer data, the value zero is considered neither positive nor negative.

4.2.1 Integer Type. An integer datum is always an exact representation of an integer value. It may assume positive, negative, and zero values. It may only assume integral values.

4.2.2 Real Type. A real datum is a processor approximation to the value of a real number. It may assume positive, negative and zero values.

4.2.3 (not used).

4.2.4 (not used).

4.2.5 Logical Type. A logical datum may assume only the truth values of true or false.

4.2.6 (not used).

4.3 Correspondence between storage and data. Each real logical and integer entity is counted as one storage unit.

5. DATA AND PROCEDURE IDENTIFICATION

Names are employed to reference or otherwise identify data and procedures.

The term reference is used to indicate an identification of a datum implying that the current value of the datum will be made available during the execution of the statement containing the reference. If the datum is identified but not necessarily made available, the datum is said to be named. One case of special interest in which the datum is named is that of assigning a value to a datum, thus defining or redefining the datum.

The term, reference, is used to indicate an identification of a procedure implying that the actions specified by the procedure will be made available.

A complete and rigorous discussion of reference and definition, including redefinition, is contained in Section 10.

5.1 Data and Procedure Names. A data name identifies a constant a variable, an array or array element, or a block (7.2.1.3). A procedure name identifies a function or a subroutine.

5.1.1 Constants. A constant is a datum that is defined by its literal occurrence and therefore, may not be redefined.

An integer or real constant is said to be signed when it is written immediately following a plus or minus. Also, for these types, an optionally signed constant is either a constant or a signed constant.

5.1.1.1 Integer constant. An integer constant is written as a non-empty string of digits. The constant is the digit string interpreted as a decimal numeral.

5.1.1.2 Real Constant. A basic real constant is written as an integer part, a decimal point, and a decimal fraction part in that order. Both the integer part and the decimal part are strings of digits; either one of these strings may be empty but not both. The constant is an approximation to the digit string interpreted as a decimal numeral.

A decimal exponent is written as the letter, E, followed by an optionally signed integer constant. A decimal exponent is a multiplier (applied to the constant written immediately preceding it) that is an approximation to the exponential form ten raised to the power indicated by the integer written following the E.

A real constant is indicated by writing a basic real constant, a basic real constant followed by a decimal exponent, or an integer constant followed by a decimal exponent.

5.1.1.3 (not used).

5.1.1.4 (not used).

5.1.1.5 Logical Constant. The logical constants, true and false are written .TRUE. and .FALSE. respectively.

5.1.1.6 (not used).

5.1.2 Variable. A variable is a datum that is identified by a symbolic name (3.5). Such a datum may be referenced and defined.

5.1.3 Array. An array is an ordered set of data of one, two or three dimensions. An array is identified by a symbolic name. Identification of the entire ordered set is achieved via use of the array name.

5.1.3.1 Array Element. An array element is one of the members of the set of data of an array. An array element is identified by immediately following the array name with a qualifier, called a subscript, which points to the particular element of the array.
An array element may be referenced and defined.

5.1.3.2 Subscript. A subscript is written as a parenthesized list of subscript expressions. Each subscript expression is separated by a comma from its successor, if there is a successor. The number of subscript expressions must correspond to the declared dimensionality (7.2.1.1), except in an EQUIVALENCE statement (7.2.1.4). Following evaluation of all of the subscript expressions, the array element successor function (7.2.1.1.) determines the identified array element.

5.1.3.3 Subscript Expressions. A subscript expression is written as one of the following constructs:

c*v+k
c*v-k
c*v
v+k
v-k
v
k

where c and k are integer constants and v is an integer variable reference. See Section 6 for a discussion of evaluation of expressions and 10.2.8 and 10.3 for requirements that apply to the use of a variable in a subscript.

5.1.4 Procedures. A procedure (Section 8) is identified by a symbolic name. A procedure is a statement function, an intrinsic function, a basic external function, an external function, or an external subroutine. Statement functions, intrinsic functions, basic external functions, and external functions are referred to as functions or function procedures; external subroutines as subroutines or subroutine procedures.

A function supplies a result to be used at the point of reference; a subroutine does not. Functions are referenced in a manner different from subroutines.

5.2 Function Reference. A function reference consists of the function name followed by an actual argument list enclosed in parentheses. If the list contains more than one argument, the arguments are separated by commas. The allowable forms of function arguments are given in Section 8.

See 10.2.1 for a discussion of requirements that apply to function references.

5.3 Type Rules for Data and Procedure Identifiers. The type of a constant is implicit in its name.

There is no type associated with a symbolic name that identifies a subroutine.

A symbolic name that identifies a variable, an array, or a statement function may have its type specified in a type-statement. In the absence of an explicit declaration, the type is implied by the first character of the name: I, J, K, L, M, and N imply type integer; any other letter implies type real.

A symbolic name that identifies an intrinsic function or a basic external function when it is used to identify this designated procedure, has a type associated with it as specified in Tables 3 and 4.

In the program unit in which an external function is referenced, its type definition is defined in the same manner as for a variable and an array. For a function subprogram, type is specified either implicitly by its name or explicitly in the FUNCTION statement.

The same type is associated with an array element as is associated with the array name.

5.4 Dummy Arguments. A dummy argument of an external procedure identifies a variable, array, subroutine, or external function.

When the use of an external function name is specified, the use of a dummy argument is permissible if an external function name will be associated with that dummy argument. (Section 8).

When the use of an external subroutine name is specified, the use of a dummy argument is permissible if an external subroutine name will be associated with that dummy argument.

When the use of a variable or array element reference is specified, the use of a dummy argument is permissible if a value of the same type will be made available through argument association.

Unless specified otherwise, when the use of a variable, array or array element name is specified, the use of a dummy argument is permissible provided that a proper association with an actual argument is made.

The process of argument association is discussed in Sections 8 and 10.

6. EXPRESSIONS

This section gives the formation and evaluation rules for arithmetic, relational, and logical expressions. A relational expression appears only within the context of logical expressions. An expression is formed from elements and operators. See 10.3 for a discussion of requirements that apply to the use of certain entities in expressions.

6.1 Arithmetic Expressions. An arithmetic expression is formed with arithmetic operators and arithmetic elements. Both the expression and its constituent elements identify values of one of the types integer or real. The arithmetic operators are:

Operator	Representing
+	Addition, positive value (zero + element)
-	Subtraction, negative value (zero-element)
*	Multiplication
/	Division
**	Exponentiation

The arithmetic elements are primary, factor, term, signed term, simple arithmetic expression, and arithmetic expression.

A primary is an arithmetic expression enclosed in parentheses, a constant, a variable reference, an array element reference, or a function reference.

A factor is a primary or a construct of the form

primary**primary

A term is a factor or a construct of one of the forms

term/factor
or
term*term

A signed term is a term immediately preceded by+or-.
A simple arithmetic expression is a term or two simple arithmetic expressions separated by a + or - .

An arithmetic expression is a simple arithmetic expression or a signed term or either of the preceding forms immediately followed by a + or - immediately followed by a simple arithmetic expression.

A primary of any type may be exponentiated by an integer primar and the resultant factor is of the same type as that of the element being exponentiated. A real primary may be exponentiated by a real primary, and the resultant factor is of type real. These are the only cases for which use of the exponentiation operator is defined.

By use of the arithmetic operators other than exponentiation, any admissible element may be combined with another admissible element of the same type, and the resultant element is of the

same type.

6.2 Relational Expressions. A relational expression consists of two arithmetic expressions separated by a relational operator and will have the value true or false as the relation is true or false, respectively. Both arithmetic expressions must be of the same type, either real or integer. The relational operators are:

Operator	Representing
.LT.	Less than
.LE.	Less than or equal to
.EQ.	Equal to
.NE.	Not equal to
.GT.	Greater than
.GE.	Greater than or equal to

6.3 Logical Expressions. A logical expression is formed with logical operators and logical elements and has the value true or false. The logical operators are:

Operator	Representing
.OR.	Logical disjunction
.AND.	Logical conjunction
.NOT.	Logical negation

The logical elements are logical primary, logical factor, logical term, and logical expression.

A logical primary is a logical expression enclosed in parentheses, a relational expression, a logical constant, a logical variable reference, a logical array element reference, or a logical function reference.

A logical factor is a logical primary or .NOT. followed by a logical primary.

A logical term is a logical factor or a construct of the form:

logical term .AND. logical term

A logical expression is a logical term or a construct of the form:

logical expression .OR. logical expression

6.4 Evaluation of Expressions. A part of an expression need be evaluated only if such action is necessary to establish the value of the expression. The rules for formation of expressions imply the binding strength of operators. It should be noted that the range of the subtraction operator is the term that immediately succeeds it. The evaluation may proceed according to any valid formation sequence (except as modified in the following paragraph).

When two elements are combined by an operator, the order of evaluation of the elements is optional. If mathematical use

of operators is associative, commutative, or both, full use of these facts may be made to revise orders of combination, provided only that integrity of parenthesized expressions is not violated. The value of an integer factor or term is the nearest integer whose magnitude does not exceed the magnitude of the mathematical value represented by that factor or term. The associative and commutative laws do not apply in the evaluation of integer terms containing division, hence the evaluation of such terms must effectively proceed from left to right.

Any use of an array element name requires the evaluation of its subscript. The evaluation of functions appearing in an expression may not validly alter the value of any other element within the expressions, assignment statement, or CALL statement in which the function reference appears. The type of the expression in which a function reference or subscript appears does not affect, nor is it affected by, the evaluation of the actual arguments or subscript.

No factor may be evaluated that requires a negative valued primary to be raised to a real exponent. No factor may be evaluated that requires raising a zero valued primary to a zero valued exponent.

No element may be evaluated whose value is not mathematically defined.

7. STATEMENTS

A statement may be classified as executable or nonexecutable. Executable statements specify actions; nonexecutable statements describe the characteristics and arrangement of data, editing information, statement functions, and classification of program units.

7.1 Executable Statements. There are three types of executable statements:

(1) Assignment statements
(2) Control statements.
(3) Input/output statements.

7.1.1 Assignment Statements. There are three types of assignment statements:

(1) Arithmetic assignment statement.
(2) Logical assignment statement.
(3) GO TO assignment statement.

7.1.1.1 Arithmetic Assignment Statement. An arithmetic assignment statement is of the form:

v = e

where v is a variable name or array element name of type other than logical and e is an arithmetic expression. Execution of this statement causes the evaluation of the expression e and the altering of v according to Table 1.

7.1.1.2 Logical Assignment Statement. A logical assignment

statement is of the form

v = e

where v is a logical variable name or a logical array element name and e is a logical expression. Execution of this statement causes the logical expression to be evaluated and its value to be assigned to the logical entity.

7.1.1.3 GO TO Assignment Statement. A GO TO assignment statement is of the form:

ASSIGN k TO i

where k is a statement label and i is an integer variable name. After execution of such a statement, subsequent execution of any assigned GO TO statement (Section 7.1.2.1.2) using that integer variable will cause the statement identified by the assigned statement label to be executed next, provided there has been no intervening redefinition (9.2) of the variable. The statement label must refer to an executable statement in the same program unit in which the ASSIGN statement appears.

Once having been mentioned in an ASSIGN statement, an integer variable may not be referenced in any statement other than an assigned GO TO statement until it has been redefined (Section 10.2.3).

TABLE 1. Rules for Assignment of e to v

If v Type Is	And e Type Is	The Assignment Rule Is *
Integer	Integer	Assign
Integer	Real	Fix & Assign
Real	Integer	Float & Assign
Real	Real	Assign

* Notes (1) Assign means transmit the resulting value, without change, to the entity.

(2) Fix means truncate any fractional part of the result and transform that value to the form of an integer datum.

(3) Float means transform the value to the form of a real datum.

7.1.2 Control Statements. There are eight types of control statements:

(1) GO TO statements.
(2) arithmetic IF statement.
(3) logical IF statement.
(4) CALL statement.
(5) RETURN statement.
(6) CONTINUE statement.
(7) STOP and PAUSE statements.
(8) DO statement.

The statement labels used in a control statement must be associated with executable statements within the same program unit in which the control statement appears.

7.1.2.1 GO TO Statements. There are three types of GO TO statements:

(1) Unconditional GO TO Statement.
(2) Assigned GO TO Statement.
(3) Computed GO TO Statement.

7.1.2.1.1 Unconditional GO TO Statement. An unconditional GO TO statement is of the form:

GO TO k

where k is a statement label.

Execution of this statement causes the statement identified by the statement label to be executed next.

7.1.2.1.2 Assigned GO TO Statement. An assigned GO TO statement is of the form:

GO TO i, (k_1, k_2..., k_n)

where i is an integer variable reference, and the k's are statement labels.

At the time of execution of an assigned GO TO statement, the current value of i must have been assigned by the previous execution of an ASSIGN statement to be one of the statement label in the parenthesized list, and such an execution causes the statement identified by that statement label to be executed nex

7.1.2.1.3 Computed GO TO Statement. A computed GO TO statement is of the form:

GO TO (k_1, k_2...,k_n),i

where the k's are statement labels and i is an integer variable reference. See 10.2.8 and 10.3 for a discussion of requirements that apply to the use of a variable in a computed GO TO statement.

Execution of this statement causes the statement identified by the statement label k_j to be executed next, where j is the valu of i at the time of the execution. This statement is defined only for values such that $1 \leq j \leq n$.

7.1.2.2 Arithmetic IF Statement. An arithmetic IF Statement is of the form:

IF (e) k_1, k_2, k_3

where e is any arithmetic expression of type integer or real and the k's are statement labels.

The arithmetic IF is a three-way branch. Execution of this statement causes evaluation of the expression e following whic the statement identified by the statement label k_1, k_2, or k_3 is executed next as the value of e is less than zero, zero, or greater than zero, respectively.

7.1.2.3 Logical IF Statement. A logical IF statement is of the form:

IF (e) S

where e is a logical expression and S is any executable statement except a DO statement or another logical IF statement. Upon execution of this statement, the logical expression e is evaluated. If the value of e is false, statement S is executed as though it were a CONTINUE statement. If the value of e is true, statement S is executed.

7.1.2.4 CALL Statement. A CALL statement is of one of the forms:

CALL s (a_1, a_2,...,a_n)

or

CALL s

where s is the name of a subroutine and the a's are actual arguments (8.4.2).

The inception of execution of a CALL statement references the designated subroutine. Return of control from the designated subroutine completes execution of the CALL statement.

7.1.2.5 RETURN Statement. A RETURN statement is of the form:

RETURN

A RETURN statement marks the logical end of a procedure subprogram and, thus, may only appear in a procedure subprogram.

Execution of this statement when it appears in a subroutine subprogram causes return of control to the current calling program unit.

Execution of this statement when it appears in a function subprogram causes return of control to the current calling program unit. At this time the value of the function (8.3.1) is made available.

7.1.2.6 CONTINUE Statement. A CONTINUE statement is of the form:

CONTINUE

Execution of this statement causes continuation of normal execution sequence.

7.1.2.7 The STOP and PAUSE statements. The STOP and PAUSE statements are program control statements.

7.1.2.7.1 STOP Statement. A STOP statement is of one of the forms:

STOP n

or

STOP

where n is an octal digit string of length from one to five.

Execution of this statement causes termination of execution of the executable program.

7.1.2.7.2 PAUSE Statement. A PAUSE statement is of one of the forms:

PAUSE n
or
PAUSE

where n is an octal digit string of length from one to five.

The inception of execution of this statement causes a cessation of execution of this executable program. Execution must be resumable. At the time of cessation of execution the octal digit string is accessible. The decision to resume execution is not under control of the program, but if execution is resumed without otherwise changing the state of the processor, the completion of the PAUSE statement causes continuation of normal execution sequence.

7.1.2.8 DO Statement. A DO statement is of one of the forms:

DO n i = m_1, m_2, m_3

or

DO n i = m_1, m_2

where

(1) n is the statement label of an executable statement. This statement, called the terminal statement of the associated DO, must physically follow and be in the same program unit as that DO statement. The terminal statement may not be a GO TO of any form, arithmetic IF, RETURN, STOP, PAUSE or DO statement, nor a logical IF containing any of these forms.

(2) i is an integer variable name; this variable is called the control variable.

(3) m_1, called the initial parameter; m_2, called the terminal parameter; and m_3, called the incrementation parameter, are each either an integer constant or integer variable reference. If the second form of the DO statement is used so that m_3 is not explicitly stated, a value of one is implied for the incrementation parameter. At time of execution of the DO statement, m_1, m_2, and m_3 must be greater than zero.

Associated with each DO statement is a range that is defined to be those executable statements from and including the first executable statement following the DO, to and including the terminal statement associated with the DO. A special situation occurs when the range of a DO contains another DO statement. In this case, the range of the contained DO must be a subset of the range of the containing DO.

A DO statement is used to define a loop. The action succeeding execution of a DO statement is described by the following five steps:

1. The control variable is assigned the value represented by the initial parameter. This value must be less than or equal to the value represented by the terminal parameter.

2. The range of the DO is executed.

3. If control reaches the terminal statement, and after execution of the terminal statement, the control variable of the most recently executed DO statement associated with the terminal statement is incremented by the value represented by the associated incrementation parameter.

4. If the value of the control variable after incrementation is less than or equal to the value represented by the associated terminal parameter, the action as described starting at step 2 is repeated with the understanding that the range in question is that of the DO, the control variable of which was most recently incremented. If the value of the control variable is greater than the value represented by its associated terminal parameter, the DO is said to have been satisfied and the control variable becomes undefined.

5. At this point, if there were one or more other DO statements referring to the terminal statement in question, the control variable of the next most recently executed DO statement is incremented by the value represented by its associated incrementation parameter and the action as described in step 4 is repeated until all DO statements referring to the particular termination statement are satisfied, at which time the first executable statement following the terminal statement is executed.
In the remainder of this section (7.1.2.8) a logical IF statement containing a GO TO or arithmetic IF statement form is regarded as a GO TO or arithmetic IF statement respectively.

Upon exiting from the range of a DO by execution of a GO TO statement or an arithmetic IF statement, that is, other than by satisfying the DO, the control variable of the DO is defined and is equal to the most recent value attained as defined in the foregoing.

A GO TO statement or an arithmetic IF statement may not cause control to pass into the range of a DO from outside its range. When a procedure reference occurs in the range of a DO, the actions of that procedure are considered to be temporarily within that range, i.e., during the execution of that reference.

The control variable of a DO may not be redefined during the execution of the range of that DO. The parameters must remain constant during execution of the range of that DO.

If a statement is the terminal statement of more than one DO

statement, the statement label of that terminal statement may not be used in any GO TO or arithmetic IF statement that occurs anywhere but in the range of the most deeply contained DO with that terminal statement.

7.1.3 Input/Output Statements. There are two types of input/output statements:

(1) READ and WRITE statements.
(2) Auxiliary input/output statements.

The first type consists of the statements that cause transfer of records of sequential files to and from internal storage, respectively. The second type consists of the BACKSPACE and REWIND statements that provide for positioning of such an external file, and ENDFILE, which provides for demarcation of such an external file.

In the following descriptions, u and f identify input/output units and format specifications, respectively. An input/output unit is identified by an integer value and u may be either an integer constant or an integer variable reference whose value then identifies the unit. The format specification is described in 7.2.3. The statement label of a FORMAT statement is represented by f. The identified statement must appear in the same program unit as the input/output statement.

A particular unit has a single sequential file associated with it. The most general case of such a unit has the following properties:

(1) If the unit contains one or more records, those records exist as a totally ordered set.

(2) There exists a unique position of the unit called its initial point. If a unit contains no records, that unit is positioned at its initial point. If the unit is at its initial point and contains records, the first record of the unit is defined as the next record.

(3) If a unit is not positioned at its initial point, there exists a unique preceding record associated with that position. The least of any records in the ordering described by (1) following this preceding record is defined as the next record of that position.

(4) Upon completion of execution of a WRITE or ENDFILE statement, there exist no records following the records created by that statement.

(5) When the next record is transmitted, the position of the unit is changed so that this next record becomes the preceding record.

If a unit does not provide for some of the properties given in the preceding, certain statements that will be defined may not refer to that unit. The use of such a statement is not defined for that unit.

7.1.3.1 READ and WRITE Statements. The READ and WRITE statements specify transfer of information. Each such statement may include a list of the names of variables, arrays and array elements. The named elements are assigned values on input and have their values transferred on output.

Records may be formatted or unformatted. A formatted record consists of a string of the characters. The transfer of such a record requires that a format specification be referenced to supply the number of records transferred by the execution of a formatted READ or WRITE is dependent upon the list and referenced format specification (7.2.3.4). An unformatted record consists of a string of representations of values. When an unformatted or formatted READ statement is executed, the required records on the identified unit must be, respectively, unformatted or formatted records.

7.1.3.1.1 Input/Output Lists. The input list specifies the names of the variables and array elements to which values are assigned on input. The output list specifies the references to variables and array elements whose values are transmitted. The input and output lists are of the same form.

Lists are formed in the following manner. A simple list is a variable name, an array element name, or an array name, or two simple lists separated by a comma.

A list is a simple list, a simple list enclosed in parentheses, a DO-implied list, or two lists separated by a comma.

A DO-implied list is a list followed by a comma and a DO-implied specification, all enclosed in parentheses.

A DO-implied specification is of one of the forms:

$$i = m_1, m_2, m_3$$

or

$$i = m_1, m_2$$

The elements i, m_1, m_2, and m_3, are as defined for the DO statement (7.1.2.8). The range of DO-implied specification is the list of the DO-implied list and, for input lists, i, m_1, m_2, and m_3, may appear, within that range, only in subscripts.

A variable name or array element name specifies itself. An array name specifies all of the array element names defined by the array declarator, and they are specified in the order given by the array element successor function (7.2.1.1.1).

The elements of a list are specified in the order of their occurrence from left to right. The elements of a list in a DO-implied list are specified for each cycle of the implied DO.

7.1.3.1.2 Formatted READ. A formatted READ statement is of one of the forms:

```
READ (u,f) k
   or
READ (u, f)
```

where k is a list.

Execution of this statement causes the input of the next records from the unit identified by u. The information is scanned and converted as specified by the format specification identified by f. The resulting values are assigned to the elements specified by the list. See however 7.2.3.4.

7.1.3.1.3 Formatted WRITE. A formatted WRITE statement is of one of the forms:

```
WRITE (u,f) k
     or
WRITE (u,f)
```

where k is a list.

Execution of this statement creates the next records on the unit identified by u. The list specifies a sequence of values. These are converted and positioned as specified by the format specification identified by f. See however 7.2.3.4.

7.1.3.1.4 Unformatted READ. An unformatted READ statement is of one of the forms:

```
READ (u) k
     or
READ (u)
```

where k is a list.

Execution of this statement causes the input of the next record from the unit identified by u,and, if there is a list, these values are assigned to the sequence of elements specified by the list. The sequence of values required by the list may not exceed the sequence of values from the unformatted record.

7.1.3.1.5 Unformatted WRITE. An unformatted WRITE statement is of the form:

```
WRITE (u) k
```

where k is a list.

Execution of this statement creates the next record on the unit identified by u of the sequence of values specified by the list.

7.1.3.2 Auxiliary Input/Output Statements. There are three types of auxiliary input/output statements:

(1) REWIND statement.
(2) BACKSPACE statement.
(3) ENDFILE statement.

7.1.3.2.1 REWIND Statement. A REWIND statement is of the form:

```
REWIND u
```

Execution of this statement causes the unit identified by u to be positioned at its initial point.

7.1.3.2.2 BACKSPACE Statement. A BACKSPACE statement is of the form:

```
BACKSPACE u
```

If the unit identified by u is positioned at its initial point, execution of this statement has no effect. Otherwise, the execution of this statement results in the positioning of the unit identified by u so that what had been the preceding record prior to that execution becomes the next record.

7.1.3.2.3 ENDFILE Statement. An ENDFILE statement is of the form:

ENDFILE u

Execution of this statement causes the recording of an endfile record on the unit identified by u. The endfile record is an unique record signifying a demarcation of a sequential file. Action is undefined when an endfile record is encountered during execution of a READ statement.

7.1.3.3 Printing of Formatted Record. When formatted records are prepared for printing, the first character of the record is not printed.

The first character of such a record determines vertical spacing as follows:

Character	Vertical Spacing Before Printing
Blank	One line
0	Two lines
1	To first line of next page
+	No advance

7.2 Nonexecutable Statements. There are four types of nonexecutable statements:

(1) Specification statements.
(2) FORMAT statement.
(3) Function defining statements.
(4) Subprogram statements.

See 10.1.2 for a discussion of restrictions on appearances of symbolic names in such statements.

The function defining statements are the statements defining the statement function and discussed in 8.1.1. The subprogram statements are FUNCTION statements or SUBROUTINE statements, which are discussed in 8.3.1 and 8.4.1 respectively.

7.2.1 Specification Statements. There are five types of specification statements:

(1) DIMENSION statement.
(2) COMMON statement.
(3) EQUIVALENCE statement.
(4) EXTERNAL statement.
(5) Type-statements.

7.2.1.1 Array-Declarator. An array declarator specifies an array used in a program unit.

The array declarator indicates the symbolic name, the number of dimensions (one, two, or three), and the size of each of the dimensions. The array declarator statement may be a type state-

ment, DIMENSION, or COMMON statement.

An array declarator has the form:

v (i)

where:

(1) v, called the declarator name, is a symbolic name,
(2) (i), called the declarator subscript, is composed of 1,2, or 3 expressions, each of which may be an integer constant or an integer variable name. Each expression is separated by a comma from its successor if there are more than one of them. In the case where i contains no integer variable, i is called the constant declarator subscript.

The appearance of a declarator subscript in a declarator statement serves to inform the processor that the declarator name is an array name. The number of subscript expressions specified for the array indicates its dimensionality. The magnitude of the values given for the subscript expressions indicates the maximum value that the subscript may attain in any array element name.

No array element name may contain a subscript that, during execution of the executable program, assumes a value less than one or larger than the maximum length specified in the array declarator.

7.2.1.1.1 Array Element Successor Function and Value of a Subscript. For a given dimensionality subscript declarator, and subscript, the value of a subscript pointing to an array element and the maximum value a subscript may attain is indicated in Table 2. A subscript expression must be greater than zero.

The value of the array element successor function is obtained by adding one to the entry in the subscript value column. Any array element whose subscript has this value is the successor to the original element. The last element of the array is the one whose subscript value is the maximum subscript value and has no successor element.

TABLE 2. Value of a Subscript

Dimensionality	Subscript Declarator	Subscript	Subscript Value	Maximum Subscript Value
1	(A)	(a)	a	A
2	(A, B)	(a, b)	a+A ·(b - 1)	A·B
3	(A, B, C)	(a, b, c)	a+A ·(b - 1) + A·B·(c - 1)	A·B·C

Notes. (1) a, b, and c are subscript expressions.
(2) A, B, and C are dimensions.

7.2.1.1.2 Adjustable Dimension. If any of the entries in a declarator subscript is an integer variable name, the array is called an adjustable array, and the variable names are called adjustable dimensions. Such an array may only appear in a procedure subprogram. The dummy argument list of the subprograms must contain the array name and the integer variable names that represent the adjustable dimensions.
The values of the actual arguments that represent array dimensions in the argument list of the reference must be defined (10.2) prior to calling the subprogram and may not be redefined or undefined during execution of the subprogram. The maximum size of the actual array may not be exceeded. For every array appearing in an executable program (9.1.6), there must be at least one constant array declarator associated through subprogram references.

In a subprogram, a symbolic name that appears in a COMMON statement may not identify an adjustable array.

7.2.1.2 DIMENSION Statement. A DIMENSION statement is of the form:

$$\text{DIMENSION } v_1(i_1),\ v_2(i_2),\ldots,v_n(i_n)$$

where each v(i) is an array declarator.

7.2.1.3. COMMON Statement. A COMMON statement is of the form:

$$\text{COMMON } /\ x_1/\ a_1/\ldots/\ x_n/\ a_n$$

where each a is a nonempty list of variable names, array names, or array declarators (no dummy arguments are permitted) and each x is a symbolic name or is empty. If x_1 is empty, the first two slashes are optional. Each x is a block name, a name that bears no relationship to any variable or array having the same name. This holds true for any such variable or array in the same or any other program unit. See 10.1.1 for a discussion of restrictions on uses of block names.

In any given COMMON statement, the entities occurring between block name x and the next block name (or the end of the statement if no block name follows) are declared to be in common block x. All entities from the beginning of the statement until the appearance of a block name, or all entities in the statement if no block name appears, are declared to be in blank or unlabelled common. Alternatively, the appearance of two slashes with no block name between them declares the entities that follow to be in blank common.

A given common block name may occur more than once in a COMMON statement or in a program unit. The processor will string together in a given common block all entities so assigned in the order of their appearance (10.1.2). The first element of an array will follow the immediately preceding entity, if one exists, and the last element of an array will immediately precede the next entity, if one exists.

The size of a common block in a program unit is the sum of the

storage required for the elements introduced through COMMON and EQUIVALENCE statements. The sizes of labelled common blocks with the same label in the program units that comprise an executable program must be the same. The sizes of blank common in the various program units that are to be executed together need not be the same. Size is measured in terms of storage units (7.2.1.3.1).

7.2.1.3.1 Correspondence of Common Blocks. If all of the program units of an executable program that contain any definition of a common block of a particular name define that block such that:

(1) There is identity in type for all entities deined in the corresponding position from the beginning of that block, and
(2) If the block is labelled and the same number of entities is defined for the block;
then the values in the corresponding positions (counted by the number of preceding storage units) are the same quantity in the executable program.

For common blocks with the same number of storage units (4.3) or blank common:

(1) In all program units which have defined the identical type to a given position (counted by the number of preceding storage units) references to that position refer to the same quantity.

(2) A correct reference is made to a particular position assuming a given type if the most recent value assignment to that position was of the same type.

7.2.1.4 EQUIVALENCE Statement. An EQUIVALENCE statement is of the form:

EQUIVALENCE (k_1), (k_2),...,(k_n)

in which each k is a list of the form:

a_1, a_2,...,a_m.

Each a is either a variable name or an array element name (not a dummy argument), the subscript of which contains only constants, and m is greater than or equal to two. The number of subscript expressions of an array element name must correspond in number to the dimensionality of the array declarator or must be one (the array element successor function defines a relation by which an array can be made equivalent to a one dimensional array of the same length).

The EQUIVALENCE statement is used to permit the sharing of storage by two or more entities. Each element in a given list is assigned the same storage (or part of the same storage) by the processor. The EQUIVALENCE statement should not be used to equate mathematically two or more entities. If a two storage unit entity is equivalenced to a one storage unit entity, the latter will share space with the first storage unit of the former.

The assignment of storage to variables and arrays declared direc-

ly in a COMMON statement is determined solely by consideration of their type and the COMMON and array declarator statements. Entities so declared are always assigned unique storage, contiguous in the order declared in the COMMON statement.

The effect of an EQUIVALENCE statement upon common assignment may be the lengthening of a common block; the only such lengthening permitted is that which extends a common block beyond the last assignment for that block made directly by a COMMON statement.

When two variables or array elements share storage because of the effects of EQUIVALENCE statements, the symbolic names of the variables or arrays in question may not both appear in COMMON statements in the same program unit.

Information contained in 7.2.1.1.1,7.2.1.3.1, and the present section suffices to describe the possibilities of additional cases of sharing of storage between array elements and entities of common blocks. It is incorrect to cause either directly or indirectly a single storage unit to contain more than one element of the same array.

7.2.1.5 EXTERNAL Statement. An EXTERNAL statement is of the form:

$$\text{EXTERNAL } v_1, v_2, \ldots, v_n$$

where each v is an external procedure name.

Appearance of a name in an EXTERNAL statement declares that name to be an external procedure name. If an external procedure name is used as an argument to another external procedure, it must appear in an EXTERNAL statement in the program unit in which it is so used.

7.2.1.6 Type-statements. A type-statement is of the form:

$$t\ v_1, v_2, \ldots, v_n$$

where t is INTEGER, REAL, OR LOGICAL, and each v is a variable name, an array name, a function name, or an array declarator.

A type-statement is used to override or confirm the implicit typing, to declare entities to be of type logical, and may supply dimension information.

The appearance of a symbolic name in a type-statement serves to inform the processor that it is of the specified data type for all appearances in the program unit. See, however, the restriction in 8.3.1 second paragraph.

7.2.2 (not used).

7.2.3 Format Statement. FORMAT statements are used in conjunction with the input/output of formatted records to provide conversion and editing information between the internal representation and the external character strings.

A FORMAT statement is of the form:

$$\text{FORMAT } (q_1 t_1 z_1 t_2 z_2 \ldots t_n z_n q_2)$$

where:

(1) $(q_1 t_1 z_1 t_2 z_2 \ldots t_n z_n q_2)$ is the format specification.

(2) Each q is a series of slashes or is empty.

(3) Each t is a field descriptor or group of field descriptors.

(4) Each z is a field separator.

(5) n may be zero.

A FORMAT statement must be labelled.

7.2.3.1 Field Descriptors. The format field descriptors are of the forms:

srFw.d
srEw.d
rIw
rLw
$nHh_1 h_2 \ldots h_n$
nX

where:

(1) The letters F, E, I, L, H, and X indicate the manner of conversion and editing between the internal and external representations and are called the conversion codes.

(2) w and n are nonzero integer constants representing the width of the field in the external character string.

(3) d is an integer constant representing the number of digits in the fractional part of the external character string.

(4) r, the repeat count, is an optional nonzero integer constant indicating the number of times to repeat the succeeding basic field descriptor.

(5) s is optional and represents a scale factor designator.

(6) Each h is one of the characters capable of representation by the processor.

For all descriptors, the field width must be specified. For descriptors of the form w.d , the d must be specified, even if it is zero. Further, w must be greater than or equal to d.

The phrase basic field descriptor will be used to signify the field descriptor unmodified by s or r.

The internal representation of external fields will correspond to the internal representation of the corresponding type constants (4.2 and 5.1.1).

7.2.3.2 Field Separators. The format field separators are the slash and the comma. A series of slashes is also a field separator. The field descriptors or groups of field descriptors are separated by a field separator.

The slash is used not only to separate field descriptors, but to specify demarcation of formatted records. A formatted record is a string of characters. The lengths of the strings for a given external medium are dependent upon both the processor and the external medium.

The processing of the number of characters that can be contained in a record by an external medium does not of itself cause the introduction or inception of processing of the next record.

7.2.3.3 Repeat Specifications. Repetition of the field descriptors (except nH and nX) is accomplished by using the repeat count. If the input/output list warrants, the specified conversion will be interpreted repetitively up to the specified number of times.

Repetition of a group of field descriptors or field separators is accomplished by enclosing them within parentheses and optionally preceding the left parenthesis with an integer constant called the group repeat count indicating the number of times to interpret the enclosed grouping. If no group repeat count is specified, a group repeat count of one is assumed. This form of grouping is called a basic group.

A further grouping may be formed by enclosing field descriptors, field separators, or basic groups within parentheses. Again, a group repeat count may be specified. The parentheses belonging to the format specification are not considered as group delineating parentheses.

7.2.3.4 Format Control Interaction with an Input/output List. The inception of execution of a formatted READ or formatted WRITE statement initiates format control. Each action of format control depends on information jointly provided respectively by the next element of the input/output list, if one exists, and the next field descriptor obtained from the format specification. If there is an input/output list, at least one field descriptor other than nH or nX must exist.

When a READ statement is executed under format control one record is read when the format control is initiated, and thereafter additional records are read only as the format specification demands. Such action may not require more characters of a record that it contains.

When a WRITE statement is executed under format control, writing of a record occurs each time the format specification demands that a new record be started. Termination of format control causes writing of the current record.

Except for the effects of repeat counts, the format specification is interpreted from left to right.

To each I, F, E or L basic descriptor interpreted in a format specification, there corresponds one element specified by the input/output list. To each H or X basic descriptor there is no corresponding element specified by the input/output list, and the format control communicates information directly with the record. Whenever a slash is encountered, the format specification demands that a new record start or the preceding record terminate. During a READ operation, any unprocessed characters of the current record will be skipped at the time of termination of format control or when a slash is encountered.

Whenever the format control encounters an I, F, E, or L basic descriptor in a format specification, it determines if there is a corresponding element specified by the input/output list. If there is such an element, it transmits appropriately converted information between the element and the record and proceeds. If there is no corresponding element, the format control terminates.

If, however, the format control proceeds to the last outer right parenthesis of the format specification, a test is made to determine if another list element is specified. If not, control terminates. However, if another list element is specified, the format control demands a new record start and control reverts to that group repeat specification terminated by the last preceding right parenthesis, or if none exists, then to the first left parenthesis of the format specification. Note, this action of itself has no effect on the scale factor.

7.2.3.5 Scale Factor. A scale factor designator is defined for use with the F and E conversions and is of the form:

nP

where n, the scale factor, is an integer constant or minus followed by an integer constant.

When the format control is initiated, a scale factor of zero is established. Once a scale factor has been established, it applies to all subsequently interpreted F and E field descriptors, until another scale factor is encountered, and then that scale factor is established.

7.2.3.5.1 Scale Factor Effects. The scale factor n affects the appropriate conversions in the following manner:

(1) For F and E input conversions (provided no exponent exists in the external field) and F output conversions, the scale factor effect is as follows:

externally represented number equals internally represented number times the quantity ten raised to the nth power.

(2) For F and E input, the scale factor has no effect if there is an exponent in the external field.

(3) For E output, the basic real constant part of the output quantity is multiplied by 10^n and the exponent is reduced by n.

7.2.3.6 Numeric Conversions. The numeric field descriptors I, F, and E are used to specify input/output of integer and real data.

(1) With all numeric input conversions, leading blanks are not significant and other blanks are zero. Plus signs may be omitted. A field of all blanks is considered to be zero.

(2) With the F and E input conversions, a decimal point appearing in the input field overrides the decimal point specification supplied by the field descriptor.

(3) With all output conversions, the output field is right justified. If the number of characters produced by the conversion is smaller than the field width, leading blanks will be inserted in the output field.

(4) With all output conversions, the external representation of a negative value must be signed; a positive value may be signed.

(5) The number of characters produced by an output conversion must not exceed the field width.

7.2.3.6.1 Integer Conversion. The numeric field descriptor Iw indicates that the external field occupies w positions as an integer. The value of the list item appears, or is to appear, internally as an integer datum.

In the external input field, the character string must be in the form of an integer constant or signed integer constant (5.1.1.1),except for the interpretation of blanks (7.2.3.6).

The external output field consists of blanks, if necessary, followed by a minus if the value of the internal datum is negative, or an optional plus otherwise, followed by the magnitude of the internal value converted to an integer constant.

7.2.3.6.2 Real Conversions. There are two conversions available for use with real data: F and E.

The numeric field descriptor Fw.d indicates that the external field occupies w positions, the fractional part of which consists of d digits. The value of the list item appears, or is to appear, internally as a real datum.

The basic form of the external input field consists of an optional sign, followed by a string of digits optionally containing a decimal point. The basic form may be followed by an exponent of one of the following forms:

(1) Signed integer constant.
(2) E followed by an integer constant.
(3) E followed by a signed integer constant.

The external output field consists of blanks, if necessary, followed by a minus if the internal value is negative, or an optional plus otherwise, followed by string of digits containing a decimal point representing the magnitude of the internal value, as modified by the established scale factor, rounded to d fractional digits.

The numeric field descriptor Ew.d indicates that the external field occupies w positions, the fractional part of which consists of d digits. The value of the list item appears, or is to appear, internally as a real datum.

The form of the external input field is the same as for the F conversion.

The standard form of the external output field for a scale factor of zero is[1]

$$\xi\ 0.x_1 \ldots x_d E$$

1) ξ signifies no character position or minus in that position.

where:

(1) $x_1 \ldots x_d$ are the d most significant rounded digits of the value of the data to be output.

(2) Y is of one of the forms:

$$E \pm y_1 y_2$$

or

$$\pm y_1 y_2 y_3$$

and has the significance of a decimal exponent (an alternative for the plus in the first of these forms is the character blank).

(3) The digit 0 in the aforementioned standard form may optionally be replaced by no character position.

(4) Each y is a digit.

The scale factor n controls the decimal normalization between the number part and the exponent part such that:

(1) If $n \leq 0$, there will be exactly -n leading zeros and d+n significant digits after the decimal point.

(2) If $n > 0$, there will be exactly n significant digits to the left of the decimal point and d-n+1 to the right of the decimal point.

7.2.3.6.3 (not used).

7.2.3.6.4 (not used).

7.2.3.7 Logical Conversion. The logical field descriptor Lw indicates that the external field occupies w positions as a string of information as defined below. The list item appears, or is to appear, internally as a logical datum.

The external input field must consist of optional blanks followed by a T or F followed by optional characters, for true and false, respectively.

The external output field consists of w-1 blanks followed by a T or F as the value of the internal datum is true or false, respectively.

7.2.3.8 Hollerith Field Descriptor. Hollerith information may be transmitted by means of the field descriptor nH.

The nH descriptor causes Hollerith information to be read into, or written from, the n characters (including blanks) following the nH descriptor in the format specification itself.

7.2.3.9 Blank Field Descriptor. The field descriptor for blanks is nX. On input, n characters of the external input record are skipped. On output, n blanks are inserted in the external output record.

7.2.3.10 (not used).

8. PROCEDURES AND SUBPROGRAMS

There are four categories of procedures: statement functions, intrinsic functions, external functions, and external subroutines. The first three categories are referred to collectively as functions or function procedures, the last as subroutines or subroutine procedures. Function subprograms and subroutine subprograms are classified as procedure subprograms. Type rules for function procedures are given in 5.3.

8.1 Statement Functions. A statement function is defined internally to the program unit in which it is referenced. It is defined by a single statement similar in form to an arithmetic or logical assignment statement.

In a given program unit, all statement function definitions must precede the first executable statement of the program unit and must follow the specification statements, if any. The name of a statement function must not appear in an EXTERNAL statement, nor as a variable name or an array name in the same program unit.

8.1.1 Defining Statement Functions. A statement function is defined by a statement of the form:

$$f(a_1, a_2, \ldots, a_n) = e$$

where f is the function name, e is an expression and the relationship between f and e must conform to the assignment rules in 7.1.1.1 and 7.1.1.2. The a's are distinct variable names, called the dummy arguments of the function. Since these are dummy arguments, their names, which serve only to indicate type, number, and order of arguments, may be the same as variable names of the same type appearing elsewhere in the program unit.

Aside from the dummy arguments, the expression e may only contain:

(1) Constants.
(2) Variable references.
(3) Intrinsic function references.
(4) References to previously defined statement functions.
(5) External function references.

8.1.2 Referencing Statement Functions. A statement function is referenced by using its reference (5.2) as a primary in an arithmetic or logical expression. The actual arguments, which constitute the argument list, must agree in order, number, and type with the corresponding dummy arguments. An actual argument in a statement function reference may be any expression of the same type as the corresponding dummy argument.

Execution of a statement function reference results in an association (10.2.2) of actual argument values with the corresponding dummy arguments in the expression of the function definition, and an evaluation of the expression. Following this, the resultant value is made available to the expression that con-

tained the function reference.

8.2 Intrinsic Functions and Their Reference. The symbolic names of the intrinsic functions (see Table 3) are predefined to the processor and have a special meaning and type if the name satisfies the conditions of 10.1.7.

An intrinsic function is referenced by using its reference as a primary in an arithmetic or logical expression. The actual arguments, which constitute the argument list, must agree in type, number, and order with the specification in Table 3 and may be any expression of the specified type. The intrinsic functions AMOD, MOD, SIGN, and ISIGN are not defined when the value of the second argument is zero.

Execution of an intrinsic function reference results in the actions specified in Table 3 based on the values of the actual arguments. Following this, the resultant value is made available to the expression that contained the function reference.

8.3 External Functions.An external function is defined externally to the program unit that references it. An external function defined by FORTRAN statements headed by a FUNCTION statement is called a function subprogram.

8.3.1 Defining Function Subprograms. A FUNCTION statement is of the form:

$$t \quad \text{FUNCTION} \quad f\ (a_1,a_2,\ldots,a_n)$$

where:

(1) t is either INTEGER, REAL or LOGICAL, or is empty.

(2) f is the symbolic name of the function to be defined.

(3) The a's, called the dummy arguments, are each either a variable name, an array name, or an external procedure name.

Function subprograms are constructed as specified in 9.1.3 with the following restrictions:

(1) The symbolic name of the function must also appear as a variable name in the defining subprogram. During every execution of the subprogram, this variable must be defined and, once defined, may be referenced or redefined. The value of the variable at the time of execution of any RETURN statement in this subprogram is called the value of the function.

(2) The symbolic name of the function must not appear in any nonexecutable statement in this program unit, except as the symbolic name of the function in the FUNCTION statement.

(3) The symbolic names of the dummy arguments may not appear in an EQUIVALENCE or COMMON statement in the function subprogram.

(4) The function subprogram may define or redefine one or more of its arguments so as to effectively return results in addition to the value of the function.

(5) The function subprogram may contain any statements except

SUBROUTINE, another FUNCTION statement, or any statement that directly or indirectly references the function being defined.

(6) The function subprogram must contain at least one RETURN statement.

8.3.2 Referencing External Functions. An external function is referenced by using its reference (5.2) as a primary in an arithmetic or logical expression. The actual arguments, which constitute the argument list, must agree in order, number, and type with the corresponding dummy arguments in the defining program unit. An actual argument in an external function reference may be one of the following:

(1) A variable name.
(2) An array element name.
(3) An array name.
(4) Any other expression.
(5) The name of an external procedure.

If an actual argument is an external function name or a subroutine name, then the corresponding dummy argument must be used as an external function name or a subroutine name, respectively.

TABLE 3. Intrinsic Functions

Intrinsic Function	Definition	Number of Arguments	Symbolic Name	Type of:	
				Argument	Function
Absolute Value	$\|a\|$	1	ABS IABS	Real Integer	Real Integer
Truncation	Sign of a times largest integer $\leq \|a\|$	1	AINT INT	Real Real	Real Integer
Remindering * (see note below)	$a_1 \pmod{a_2}$	2	AMOD MOD	Real Integer	Real Integer
Choosing Largest Value	Max $(a_1, a_2 \ldots)$	≥ 2	AMAXO AMAX1 MAXO MAX1	Integer Real Integer Real	Real Real Integer Integer
Choosing Smallest Value	Min$(a_1, a_2 \ldots)$	≥ 2	AMINO AMIN1 MINO MIN1	Integer Real Integer Real	Real Real Integer Integer
Float	Conversion from integer to real	1	FLOAT	Integer	Real
Fix	Conversion from real to integer	1	IFIX	Real	Integer
Transfer of Sign	Sign of a_2 times $\|a_1\|$	2	SIGN ISIGN	Real Integer	Real Integer
Positive Difference	a_1-Min(a_1, a_2)	2	DIM IDIM	Real Integer	Real Integer

* The function MOD or AMOD (a_1, a_2) is defined as $a_1 - [a_1/a_2]\, a_3$, where $[x]$ is the integer whose magnitude does not exceed the magnitude of x and whose sign is the same as x.

If an actual argument corresponds to a dummy argument that is defined or redefined in the referenced subprogram, the actual argument must be a variable name, an array element name, or an array name. Execution of an external function reference as described in the foregoing, results in an association (10.2.2) of actual arguments with all appearances of dummy arguments in executable statements, function definition statements, and as adjustable dimensions in the defining subprogram. If the actual argument is as specified in item (4) in the foregoing, this association is by value rather than by name. Following these associations, execution of the first executable statement of the defining subprogram is undertaken. An actual argument which is an array element name containing variables in the subscript could in every case be replaced by the same argument with a constant subscript containing the same values as would be derived by computing the variable subscript just before the association of arguments takes place.

If a dummy argument of an external function is an array name, the corresponding actual argument must be an array name or array element name (10.1.3).

If a function reference causes a dummy argument in the referenced function to become associated with another dummy argument in the same function or with an entity in common, a definition of either within the function is prohibited.

Unless it is a dummy argument, an external function is also referenced (in that it must be defined) by the appearance of its symbolic name in an EXTERNAL statement.

8.3.3 Basic External Functions. FORTRAN processors must supply the external functions listed in Table 4. Referencing of these functions is accomplished as described in (8.3.2). Arguments for which the result of these functions is not mathematically defined or is of type other than that specified are improper.

8.4 Subroutine. An external subroutine is defined externally to the program unit that references it. An external subroutine defined by FORTRAN statements headed by a SUBROUTINE statement is called a subroutine subprogram.

TABLE 4. Basic External Functions.

Basic External Function	Definition	Number of Arguments	Symbolic Name	Type of:	
				Argument	Function
Exponential	e^a	1	EXP	Real	Real
Natural Logarithm	$\log_e(a)$	1	ALOG	Real	Real
Common Logarithm	$\log_{10}(a)$	1	ALOG10	Real	Real
Trigonometric Sine	sin (a)	1	SIN	Real	Real
Trigonometric Cosine	cos (a)	1	COS	Real	Real
Hyperbolic Tangent	tanh (a)	1	TANH	Real	Real
Square Root	$(a)^{1/2}$	1	SQRT	Real	Real
Arctangent	arctan(a)	1	ATAN	Real	Real
	$\arctan(a_1/a_2)$	2	ATAN2	Real	Real

8.4.1 Defining Subroutine Subprograms. A SUBROUTINE statement is of one of the forms:

SUBROUTINE s (a_1, a_2,...,a_n)

or

SUBROUTINE s

where:

(1) s is the symbolic name of the subroutine to be defined.
(2) The a's, called the dummy arguments, are each either a variable name, an array name, or an external procedure name.

Subroutine subprograms are constructed as specified in 9.1.3 with the following restrictions:

(1) The symbolic name of the subroutine must not appear in any statement in this subprogram except as the symbolic name of the subroutine in the SUBROUTINE statement itself.

(2) The symbolic names of the dummy arguments may not appear in an EQUIVALENCE or COMMON statement in the subprogram.

(3) The subroutine subprogram may define or redefine one or more of its arguments so as to effectively return results.

(4) The subroutine subprogram may contain any statements except FUNCTION, another SUBROUTINE statement, or any statement that directly or indirectly references the subroutine being defined.

(5) The subroutine subprogram must contain at least one RETURN statement.

8.4.2 Referencing Subroutines. A subroutine is referenced by a CALL statement (7.1.2.4). The actual arguments, which constitute the argument list, must agree in order, number, and type with the corresponding dummy arguments in the defining program. An actual argument in a subroutine reference may be one of the following:

(1) A variable name.
(2) An array element name.
(3) An array name.
(4) any other expression.
(5) The name of an external procedure.

If an actual argument is an external function name or a subroutine name, the corresponding dummy argument must be used as an external function name or a subroutine name, respectively.

If an actual argument corresponds to a dummy argument that is defined or redefined in the referenced subprogram, the actual argument must be a variable name, an array element name or an array name.

Execution of a subroutine reference as described in the foregoing results in an association of actual arguments with all appearances of dummy arguments in executable statements, function definition statements, and as adjustable dimensions in the defining subprogram. If the actual argument is as specified in item (4) in the foregoing, this association is by value rather than by name. Following these associations, execution of the first executable statement of the defining subprogram is undertaken.

An actual argument which is an array element name containing variables in the subscript could in every case be replaced by the same argument with a constant subscript containing the same values as would be derived by computing the variable subscript just before the association of arguments takes place.

If a dummy argument of an external function is an array name, the corresponding actual argument must be an array name or array element name (10.1.3).

If a subroutine reference causes a dummy argument in the referenced subroutine to become associated with another dummy argument in the same subroutine or with an entity in common, a definition of either entity within the subroutine is prohibited.

Unless it is a dummy argument, a subroutine is also referenced

(in that it must be defined) by the appearance of its symbolic name in an EXTERNAL statement.

8.5 (not used).

9. PROGRAMS

An executable program is a collection of statements, comment lines, and end lines that completely (except for input data values and their effects) describe a computing procedure.

9.1 Program Components. Programs consist of program parts, program bodies, and subprogram statements.

9.1.1 Program Part. A program part is a collection of executable statements, FORMAT statements, and statement function definitions. A program part must contain at least one executable statement, however it need not contain any statements from those of the latter two classes of statements. The statement function definitions must precede the first executable statement.

9.1.2 Program Body. A program body is a collection of specification statements, FORMAT statements or both, or neither, followed by a program part, followed by an end line.

9.1.3 Subprogram. A subprogram consists of a SUBROUTINE or FUNCTION statement followed by a program body.

9.1.4 (not used)

9.1.5 Main Program. A main program consists of a program body.

9.1.6 Executable Program. An executable program consists of a main program plus any number of subprograms, external procedures, or both.

9.1.7 Program Unit. A program unit is a main program or a subprogram.

9.2 Normal Execution Sequence. When an executable program begins operation, execution commences with the execution of the first executable statement of the main program. A subprogram, when referenced, starts execution with execution of the first executable statement of that subprogram. Unless a statement is a GO TO, arithmetic IF, RETURN, or STOP statement or the terminal statement of a DO, completion of execution of that statement causes execution of the next following executable statement. The sequence of execution following execution of any of these statements is described in Section 7. A program part may not (in the sense of 1.1) contain an executable statement that can never be executed.

10. INTRA- AND INTERPROGRAM RELATIONSHIPS.

10.1 Symbolic Names. A symbolic name has been defined to consist of from one to six alphanumeric characters, the first of which must be alphabetic. Sequences of characters that are format field descriptors or uniquely identify certain statement types,

e.g., GO TO, READ, FORMAT, etc., are not symbolic names in such occurrences nor do they form the first characters of symbolic names in these cases. In a program unit, a symbolic name (perhaps qualified by a subscript) must identify an element of one (and usually only one) of the following classes:

Class I	An array and the elements of that array.
Class II	A variable.
Class III	A statement function.
Class IV	An intrinsic function.
Class V	An external function.
Class VI	A subroutine.
Class VII	An external procedure which cannot be classified as either a subroutine or an external function in the program unit in question.
Class VIII	A block name.

10.1.1 Restrictions on Class. A symbolic name in Class VIII in a program unit may also be in any one of the Classes I, II, or III in that program unit.

In the program unit in which a symbolic name in Class V appears immediately following the word FUNCTION in a FUNCTION statement, that name must also be in Class II.

Once a symbolic name is used in Class V, VI, VII, or VIII in any unit of an executable program, no other program unit of that executable program may use that name to identify an entity of these classes other than the one originally identified. In the totality of the program units that make up an executable program, a Class VII name must be associated with a Class V or VI name. Class VII can only exist locally in program units.

In a program unit, no symbolic name can be in more than one class except as noted in the foregoing. There are no restrictions on uses of symbolic names in different program units of an executable program other than those noted in the foregoing.

10.1.2 Implications of Mentions in Specification Statements. A symbolic name is in Class I if and only if it appears as a declarator name. Only one such appearance for a symbolic name in a program unit is permitted.

A symbolic name that appears in a COMMON statement (other than as a block name) is either in Class I, or in Class II but not Class V.(8.3.1). Only one such appearance for a symbolic name in a program unit is permitted.

A symbolic name that appears in an EQUIVALENCE statement is either in Class I, or in Class II but not Class V.(8.3.1).

A symbolic name that appears in a type-statement cannot be in Class VI or Class VII. Only one such appearance for a symbolic name in a program unit is permitted.

A symbolic name that appears in an EXTERNAL statement is in either Class V, Class VI, or Class VII. Only one such ap-

pearance for a symbolic name in a program unit is permitted.

10.1.3 Array and Array Element. In a program unit, any appearance of a symbolic name that identifies an array must be immediately followed by a subscript, except for the following cases:

(1) In the list of an input/output statement.
(2) In a list of dummy arguments.
(3) In the list of actual arguments in a reference to an external procedure.
(4) In a COMMON statement.
(5) In a type-statement.

Only when an actual argument of an external procedure reference is an array name or an array element name may the corresponding dummy argument be an array name. If the actual argument is an array name, the length of the dummy argument array must be no greater than the length of the actual argument array. If the actual argument is an array element name, the length of the dummy argument array must be less than or equal to the length of the actual argument array plus one minus the value of the subscript of the array element.

10.1.4 External Procedures. The only case when a symbolic name is in Class VII occurs when that name appears only in an EXTERNAL statement and as an actual argument to an external procedure in a program unit.

Only when an actual argument of an external procedure reference is an external procedure name may the corresponding dummy argument be an external procedure name.

In the execution of an executable program, a procedure subprogram may not be referenced twice without the execution of a RETURN statement in that procedure having intervened.

10.1.5 Subroutine. A symbolic name is in Class VI if it appears:

(1) Immediately following the word SUBROUTINE in a SUBROUTINE statement.

(2) Immediately following the word CALL in a CALL statement.

10.1.6 Statement Function. A symbolic name is in Class III in a program unit if and only if it meets all three of the following conditions:

(1) It does not appear in an EXTERNAL statement nor is it in Class I.

(2) Every appearance of the name, except in a type-statement, is immediately followed by a left parenthesis.

(3) A function defining statement (8.1.1) is present for that symbolic name.

10.1.7 Intrinsic Function. A symbolic name is in Class IV in a program unit if and only if it meets all four of the following conditions:

(1) It does not appear in an EXTERNAL statement nor is it in

Class I or Class III.

(2) The symbolic name appears in the name column of the table in Section 8.2.

(3) The symbolic name does not appear in a type-statement of type different from the intrinsic type specified in the table.

(4) Every appearance of the symbolic name (except in a type-statement as described in the foregoing) is immediately followed by an actual argument list enclosed in parentheses.

The use of an intrinsic function in a program unit of an executable program does not preclude the use of the same symbolic name to identify some other entity in a different program unit of that executable program.

10.1.8 External Function. A symbolic name is in Class V if it:
(1) Appears immediately following the word FUNCTION in a FUNCTION statement.

(2) Is not in Class I, Class III, Class IV or Class VI and appears immediately followed by a left parenthesis on every occurrence except in a type-statement, in an EXTERNAL statement, or as an actual argument. There must be at least one such appearance in the program unit in which it is so used.

10.1.9 Variable. In a program unit, a symbolic name is in Class II if it meets all three of the following conditions:

(1) It is not in Class VI or Class VII.
(2) It is never immediately followed by a left parenthesis unless it is immediately preceded by the word FUNCTION in a FUNCTION statement.

(3) It occurs other than in a Class VIII appearance.

10.1.10 Block Name. A symbolic name is in Class VIII if and only if it is used as a block name in a COMMON statement.

10.2 Definition. There are two levels of definition of numeric values, first level definition and second level definition. The concept of definition on the first level applies to array elements and variables; that of second level definition to integer variables only. These concepts are defined in terms of progressions of execution; and thus, an executable program, complete and in execution, is assumed in what follows.

There are two other varieties of definition that should be noted. The first, effected by a GO TO assignment and referring to an integer variable being defined with other than an integer value, is discussed in 7.1.1.3 and 7.1.2.1.2; the second, which refers to when an external procedure may be referenced, will be discussed in the next section.

In what follows, otherwise unqualified use of the terms definition and undefinition (or their alternate forms) as

applied to variables and array elements will imply modification by the phrase on the first level.

10.2.1 Definition of Procedures. If an executable program contains information describing an external procedure, such an external procedure with the applicable symbolic name is defined for use in that executable program. An external function reference or subroutine reference (as the case may be) to that symbolic name may then appear in the executable program, provided that number of arguments agrees between definition and reference. In addition, for an external function, the type of function must agree between definition and reference. Other restrictions on agreements are contained in 8.3.1, 8.3.2, 8.4.1, 8.4.2, 10.1.3, and 10.1.4.

The basic external functions listed in (8.3.3) are always defined and may be referenced subject to the restrictions alluded to in the foregoing.

A symbolic name in Class III or Class IV is defined for such use.

10.2.2 Associations That Effect Definition. Entities may become associated by:

(1) COMMON association.
(2) EQUIVALENCE association.
(3) Argument substitution.

Multiple association to one or more entities can be the result of combinations of the foregoing. Any definition or undefinition of one of a set of associated entities effects the definition or undefinition of each entity of the entire set.

For purposes of definition, in a program unit there is no association between any two entities both of which appear in COMMON statements. Further, there is no other association for common and equivalenced entities other than those stated in 7.2.1.3.1 and 7.2.1.4.

If an actual argument of an external procedure reference is an array name, an array element name, or a variable name, then the discussions in 10.1.3 and 10.2.1 allow an association of dummy arguments with the actual arguments only between the time of execution of the first executable statement of the procedure and the inception of execution of the next encountered RETURN statement of that procedure. Note specifically that this association can be carried through more than one level of external procedure reference.

In what follows, variables or array elements associated by the information in 7.2.1.3.1 and 7.2.1.4 will be equivalent if and only if they are of the same type.

If an entity of a given type becomes defined, then all associated entities of different type become undefined at the same time, while all associated entities of the same type become defined unless otherwise noted.

Association by argument substitution is only valid in the case of identity of type, so the rule in this case is that an entity created by argument substitution is defined at time of entry if and only if the actual argument was defined. If an entity created by argument substitution becomes defined or undefined (while the association exists) during execution of a subprogram, then the corresponding actual entities in all calling program units becomes defined or undefined accordingly.

10.2.3 Events That Effect Definition. Any entity is undefined at the time of the first execution of the first executable statement of the main program. Redefinition of a defined entity is always permissible except for certain integer variables (7.1.2.8, 7.1.3.1.1., and 7.2.1.1.2) or certain entities in subprogram (6.4., 8.3.2., and 8.4.2).

Variables and array elements become defined or redefined as follows:

(1) Completion of execution of an arithmetic or logical assignment statement causes definition of the entity that precedes the equals.

(2) As execution of an input statement proceeds, each entity, which is assigned a value of its corresponding type from the input medium is defined at the time of such association. Only at the completion of execution of the statement do associated entities of the same type become defined.

(3) Completion of execution of a DO statement causes definition of the control variable.

(4) Inception of execution of action specified by a DO-implied list causes definition of the control variable.

Variables and array elements become undefined as follows:

(1) At the time a DO is satisfied, the control variable becomes undefined.

(2) Completion of execution of an ASSIGN statement causes undefinition of the integer variable in the statement.

(3) Certain entities in function subprograms (10.2.9) become undefined.

(4) Completion of execution of action specified by a DO-implied list causes undefinition of the control variable.

(5) When an associated entity of different type becomes defined.

(6) When an associated entity of the same type becomes undefined.

10.2.4 Entities in Blank Common

Entities in blank common and those entities associated with them, once defined by any of the rules previously mentioned, remain defined until they become undefined.

name appears as a block name in the program unit. If a main program or referenced subprogram contains a labelled common block name, any entity in the block (and its associates) once defined remain defined until they become undefined.

If a subprogram contains a labelled common block name that is not contained in any program unit currently referencing the subprogram directly or indirectly, the execution of a RETURN statement in the subprogram causes undefinition of all entities in the block (and their associates) except for initially defined entities that have maintained their initial definitions.

10.2.6 Entities Not in Common. An entity not in common is initially undefined.

Such entities once defined by any of the rules previously mentioned, remain defined until they become undefined.

If such an entity is in a subprogram, the completion of execution of a RETURN statement in that subprogram causes all such entities and their associates at that time to become undefined. In this respect, it should be noted that the association between dummy arguments and actual arguments is terminated at the inception of execution of the RETURN statement.

10.2.7 Basic Block. In a program unit, a basic block is a group of one or more executable statements defined as follows. The following statements are block terminal statements:

(1) DO statement.
(2) CALL statement.
(3) GO TO statement of all types.
(4) Arithmetic IF statement.
(5) STOP statement.
(6) RETURN statement.
(7) The first executable statement, if it exists, preceding a statement whose label is mentioned in a GO TO or arithmetic IF statement.

(8) An arithmetic statement in which an integer variable precedes the equals.
(9) A READ statement with an integer variable in the list.
(10) A logical IF containing any of the admissible forms given in the foregoing.

The following statements are block initial statements:

(1) The first executable statement of a program unit.
(2) The first executable statement, if it exists, following a block terminal statement.

Every block initial statement defines a basic block. If that initial statement is also a block terminal statement, the basic block consists of that one statement. Otherwise, the basic block consists of the initial statement and all executable statements that follow until a block terminal statement is encountered. The terminal statement is included in the basic block.

10.2.7.1 Last Executable Statement. In a program unit the last executable statement (which cannot be part of a logical IF) must be one of the following statements: GO TO statement, arithmetic IF statement, STOP statement or RETURN statement.

10.2.8 Second Level Definition. Integer variables must be defined on the second level when used in subscripts and computed GO TO statements.

Redefinition of an integer entity causes all associated variables to be undefined for use on the second level during this execution of this program unit until the associated integer variable is explicitly redefined.

Except as just noted, an integer variable is defined on the second level upon execution of the initial statement of a basic block only if both of the following conditions apply:

(1) The variable is used in a subscript or in a computed GO TO in the basic block in question.

(2) The variable is defined on the first level at the time of execution of the initial statement in question.

This definition persists until one of the following happens:

(1) Completion of execution of the terminal statement of the basic block in question.

(2) The variable in question becomes undefined or receives a new definition on the first level.

At this time, the variable becomes undefined on the second level.

In addition, the occurrence of an integer variable in the list of an input statement in which that integer variable appears following in a subscript causes that variable to be defined on the second level. This definition persists until one of the following happens:

(1) Completion of execution of the terminal statement of the basic block containing the input statement.

(2) The variable becomes undefined or receives a new definition on the first level.

An integer variable defined as the control variable of a DO-implied list is defined on the second level over the range of that DO-implied list and only over that range.

10.2.9 Certain Entities in Function Subprograms. If a function subprogram is referenced more than once with an identical argument list in a single statement, the execution of that subprogram must yield identical results for those cases mentioned, no matter what the order of evaluation of the statement.

If a statement contains a factor that may not be evaluated (6.4), and if this factor contains a function reference, then

all entities that might be defined in that reference become undefined at the completion of evaluation of the expression containing the factor.

10.3 Definition Requirements for Use of Entities. Any variable referenced in a subscript or a computed GO TO must be defined on the second level at the time of this use.

Any variable, array element, or function referenced as a primary in an expression and any subroutine referenced by a CALL statement must be defined at the time of this use. In the case where an actual argument in the argument list of an external procedure reference is a variable name or an array element name, this in itself is not a requirement that the entity be defined at the time of the procedure reference; however, when such an argument is an external procedure name, it must be defined.

Any variable used as an initial value, terminal value, or incrementation value of a DO statement or a DO-implied list must be defined at the time of this use.

Any variable used to identify an input/output unit must be defined at the time of this use.

At the time of execution of a RETURN statement in a function subprogram, the value (8.3.1) of that function must be defined.

At the time of execution of an output statement, every entity whose value is to be transferred to the output medium must be defined unless the output is under control of a format specification and the corresponding conversion code is A. If the output is under control of a format specification, a correct association of conversion code with type of entity is required unless the conversion code is A. The following are the correct associations: I with integer; E and F with real and L with logical.

APPENDIX 1

ITEMS INCLUDED IN ECMA FORTRAN WHICH HAVE BEEN EXCLUDED FROM BASIC FORTRAN

1. Up to nine continuation cards (5 in Basic)
2. Six Character identifiers (5 in Basic)
3. Logical data
4. Logical IF
5. Relational and Logical expressions
6. Type statements
7. EXTERNAL statements
8. 3-dimensional arrays (2 in Basic)
9. Adjustable dimensions
10. If dummy argument is an array, actual argument may be an array element
11. DIMENSION information may appear in COMMON statements
12. Named COMMON blocks
13. GO TO assignment statement and assigned GO TO
14. 5 octal digits in STOP and PAUSE (4 in Basic)
15. Print Carriage Control
16. Scale Factor
17. Input of numbers with exponent when the Field Description is Fw.d.
18. Embedded and trailing blanks are treated as zeros during numeric input conversion
19. Second level of parentheses in FORMAT statements
20. Side effects in FUNCTION
21. The functions AINT, INT, AMOD, MOD, AMAX0, AMAX1, MAX0, MAX1, AMIN0, AMIN1, MIN0, MIN1, DIM, IDIM, ATAN2, ALOG10.
22. COMMON, EQUIVALENCE, DIMENSION do not have to be ordered.

APPENDIX 2

ITEMS OF ISO FORTRAN WHICH HAVE BEEN EXCLUDED FROM ECMA FORTRAN

1. Up to 19 continuation cards (9 in ECMA)
2. Currency sign
3. COMPLEX data
4. Double length data
5. COMPLEX and DOUBLE functions
6. Hollerith data type
7. Extended range of DO statement
8. FORMAT input at run time
9. G, A, and D field descriptors
10. Data initialization statement and Block Data program.